Оглавление

Урок от 18.09.18

Введение в программирование

Часть 1. Программирование на языке СИ

Создание языка Си
Си был создан в период с 1970 г. по 1972 г. сотрудниками фирмы Bell Labs Денисом Ритчи и Кеном Томпсоном во время написания ОС Unix. Языку Си предшествовали два языка BCPL и B. Автором языка B был Кен Томпсон. В 1973г. Денис Ритчи и Кен Томпсон переписали ядро системы Unix на языке Си и отошли от принятого стандарта использовать язык Ассемблер для написания операционных систем. Сначала фактическим стандартом языка Си была версия для операционной системы UNIX, которая была впервые описана в 1978г. С 1983 года велась работа по стандартизации языка Си. В 1989г. Американский институт национальных стандартов (American National Standards Institute – ANSI) одобрил стандарт языка Си. Эту версию языка Си обычно называют стандартом C89. В 1999 году появился новый стандарт языка Си – C99.

Алфавит языка Си
В основе языка Си лежат неделимые элементарные частицы — символы алфавита языка программирования, которые составляют таблицу кодов ASCII.
Из символов строятся элементарные смысловые понятия (слова).
Алфавит Си включает:
1. Строчные и прописные буквы латинского алфавита.
2. Цифры от 0 до 9.
3. Символ ”_” (подчеркивания).
4. Набор специальных символов
   “ {} , | [] + - % / \ ; ‘ : ? < > = ! & # ~ ^ . *
5. Прочие символы.

Типы слов:
1. Ключевые слова.
2. Идентификаторы (имена переменных).
3. Символы (знаки) операций.
4. Разделители.
5. Литералы (константы) — фиксированные значения, которые программа не может изменять. Способ представления литерала зависит от его типа (например 'a' – символьная константа).

Из слов строятся выражения (предложения), команды.
В Си признаком конца команды является “**;**”.
Из команд строятся программные модули.
В Си нет подпрограмм, есть только функции. Язык Си не является блочно-стуктурированным, так как не позволяет объявлять одну функцию внутри других, но считается структурированным.

Для образования ключевых слов и идентификаторов используются латинские символы, знак подчеркивания и цифры.
Идентификаторы могут начинаться с символа и _ (знак подчеркивания).

Одинаковые прописные и строчные буквы считаются *разными символами.*

Типы данных:

1. Простые.
2. Составные.

1. **Простые данные:**
   1.1. **Числа**: целые, вещественные.
   1.2. **Символы** — символ=1 байт=8 бит. Набор символов <=256.
   1.3. **Адреса** (ссылка, указатель) — целое число без знака.

2. **Составные данные:**

2.1. Массивы данных. Одномерный массив — последовательность чисел. Двумерный массив — таблица чисел. Существует понятие массив массивов.

2.2. Файлы:
Файл — логическая совокупность данных + физическое место хранения. Файл состоит из символов, чисел, массивов,….

2.3. Структуры, объединения, перечисления:
Структура — совокупность логически связанных данных разных типов. Существуют массивы структур. Все простые типы могут быть использованы для организации составных данных.

Целочисленный литерал — служит для записи целочисленных значений и является соответствующей последовательностью цифр (возможно со знаком "-").
Ц.л., начинающийся с 0, воспринимается как восьмеричное целое.
Ц.л., начинающийся с 0x или 0X, воспринимается как шестнадцатиричное целое. Непосредственно за литералом может располагаться один из (или два) специальных суффикса: U (u) и L (l).

Вещественный литерал — служит для отображения вещественных значений в обычной десятичной или научной нотациях. Непосредственно за литералом могут располагаться один из двух специальных суффиксов: F (f) и L(l).

Символьный литерал — служит для вывода соответствующего значения ASCII кода и представляет собой последовательность из одной или нескольких литер, заключенных в кавычки. Например, литера Z может быть представлена литералом 'Z', '\132', '\x5A'.
Рассмотрим список литер, которые используются в качестве служебных символов или не имеют графического представления.

\0	\x00	Нулевая литера
\a	\x07	Сигнал
\b	\x08	Возврат на шаг
\f	\x0C	Перевод страницы
\n	\x0A	Перевод строки
\r	\x0D	Возврат каретки
\t	\x09	Горизонтальная табуляция
\v	\x0B	Вертикальная табуляция
\\	\x5C	Обратная косая черта
\'	\x27	‘
\"	\x22	“
\?	\x3F	?

Строковый литерал — является последовательностью литер, заключенных в двойные кавычки.
'x' — символ x, "x" — два символа (x и \0), где \0 — терминатор (конец строки).

Пример простой программы на языке Си

```
#include<stdio.h>
int main()
{
int a,b;
a=10;
b=2*a;
printf("b=%d\n",b);
return 0;
}
```

Программу на языке Си можно условно разделить на две части:
1. декларативную (описание);
2. императивную (исполняемую).

Часть I – директива препроцессора

#include<stdio.h>
#include – включить;
<stdio.h> – объект включения (прототипы функций стандартного ввода, вывода).
Компилятор получает директиву, ищет текст, берет копию и присоединяет ее к тексту программы.

Часть II – исполняемая.

main() – имя главной функции, т.е. выполнение программы начинается с этой функции. У функции есть имя и тело. В теле функции тоже может быть две части – декларативная и императивная.

Декларативная – int a,b;
Императивная – остальная часть тела функции.

Рассмотрим функцию вывода на стандартный терминал:
printf("I",II);
I – строка формата (в кавычках).
II – объекты вывода (через запятые).
%d – спецификатор вывода (целочисленное значение).
\n – управляющая последовательность, перевод курсора на начало следующей строки.

Замечание: Возможно задание в спецификаторе ширины поля:
%nd %n.mf %ne %ns

Пример: /* Элементы программы */

```
#include<stdio.h>
# include<windows.h>       // CharToOem()
int main()
{
int iVar=56;
float fVar=45.567F;
char cStr[]="Русский текст!!!";
CharToOem(cStr,cStr);
printf("iVar=%5d\tfVar=%5.2f\tfVar=%6e\n",iVar,fVar,fVar);
printf("%25s\n",cStr);
printf("%-25s\n",cStr);
printf("%25.6s\n",cStr);
printf("%-25.6s\n",cStr);
return 0;
}
```

/* Нотация Windows*/

Рассмотрим элементы "венгерской нотации" (венгерский программист фирмы Microsoft Чарльз Симони) — имена переменных начинаются со строчной буквы или букв, описывающих тип данных переменных (префикс).

Префикс	Соответствующий тип данных
b	BOOL (целое число)
c	Character (символ)
i	Integer (целое число)
p	Указатель
s	Строка
v	Void
w	Word

При работе с компилятором Visual C++ существует проблема вывода на экран букв русского алфавита. Для решения этой проблемы обычно используют функцию CharToOem(), прототип которой описан в <windows.h>

Пример: /*Использование русского текста при выводе*/

```
#include<stdio.h>
```

```
#include<windows.h>
int main()
{
	char str[20];
	CharToOem("Русский текст!!!",str);
	printf("%s\n",str);
	char str1[20]="Второй вариант !!!";
	CharToOem(str1,str);
	printf("%s\n",str);
	return 0;
}
```

Типы данных. Операции над данными

Стандартные типы данных: В языке Си существует пять элементарных типов данных. Это

1. символ – **char**;
2. целое число – **int**;
3. число с плавающей точкой – **float**;
4. число с плавающей точкой удвоенной точности – **double**;
5. переменная, не имеющая значений – **void**.

Замечание: Стандарт С99 добавляет еще три основных типа:

1. _Bool;
2. _Complex;
3. _Imaginary.

Все другие типы данных в Си создаются на основе элементарных типов.

Название типа	Нижняя граница	Верхняя граница	Точность	Байт
bool	False	True	No	1
char	-128	127	No	1
short	-32768	32767	No	2
int	-2 147 483 648	2 147 483 647	No	4
long	-2 147 483 648	2 147 483 647	No	4
float	$3.4*10^{-38}$	$3.4*10^{38}$	7	4
double	$1.7*10^{-308}$	$1.7*10^{308}$	15	8

Беззнаковые целые типы:

Название	Нижняя граница	Верхняя граница	Размер (байт)
unsigned char	0	255	1
unsigned short	0	65535	2
unsigned int	0	4 294 967 295	4
unsigned long	0	4 294 967 295	4

Неявное преобразование типов:

Тип данных	Старшинство
long double	Высший
double	
float	
long	
int	
short	
char	Низший

Спецификатор типов:

Спецификатор	Форма вывода
%d	десятичное число со знаком
%i	десятичное число со знаком

%u	беззнаковое целое десятичное число
%f	вещественное число с плавающей точкой
%e	экспоненциальная форма
%c	символ.
%s	строка символов
%o	восьмеричное число
%x	шестнадцатиричное число
%p	указатель
%ld	long int
%lf	long double

Пример: /* Программа определения номера кода и размеров типов */

```
#include<stdio.h>
int main()
{
        char ch;
        unsigned char ch1;
        int iVar1=90;
        int iVar2=045;
        int iVar3=0x45;
        printf(" iVar1=%d, iVar2=%d, iVar3=%d\n",   iVar1,iVar2,iVar3);
        printf("Input symbol \n");
        scanf("%c",&ch);
        ch1=ch;
        printf("Kod %c = %d %d.\n",ch,ch,ch1);
        printf("char = %d\n",sizeof(char));
        printf("short = %d\n",sizeof(short));
        printf("int = %d\n",sizeof(int));
        printf("long = %d\n",sizeof(long));
        printf("float = %d\n",sizeof(float));
        printf("double = %d\n",sizeof(double));
        printf("long double = %d\n",sizeof(long double));
printf("dec=%d oct=%o hex=%x char=%c\n",iVar1,iVar1,iVar1,iVar1);
printf("z= %c 132=%c x5a=%c\n",'Z','\132','x5a');
printf("\'Z\'= %c \'\\132\'=%c \'x5a\'=%c\n','Z','\132','\x5a');
        // dec=90 oct=132 hex=5a char=Z
return 0;
}
```

Если на запрос программы вести русскую букву "я", то код этой буквы будет для типа char – -17, а для типа unsigned char – 239. Для объяснения этого результата необходимо рассмотреть представление числа в дополнительном обратном коде.

Классификация операций

Классификация операций идет по числу операндов.

Группа унарных операций

Знак операции	Операция
-	Арифметическое отрицание
~	Поразрядное логическое отрицание (дополнение)
!	Логическое отрицание
*	Разадресация
&	Адрес
+	Унарный плюс
Sizeof	Размер
++ --	Операции единичного приращения

Операции разадресации и адреса используются для работы с переменными типа указатель.
Операция ”адрес” (”&”) дает адрес своего операнда.
Операция разадресации (”*”) осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина переменной, на которую указывает операнд.
Операция * — операция обращения по адресу (не знаем адрес, но обращаемся к нему).

Пример:

```
int x, y =10, *address;
address=&x;
*address=y;
printf(“y=%d x=%d”,y,x);
```

Операции единичного приращения:

++ – приращение положительно (инкремент).

-- – приращение отрицательно (декремент).

Операции увеличения (++) и уменьшения (--) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значение операнда на единицу.

Операнд должен быть:

1. Целого типа.
2. Плавающего типа.
3. Адрес (указатель).

Для целого и плавающего типа увеличение или уменьшение происходит на "1".Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует.

Существует две формы:

1. Префиксная — сначала приращение, потом действие.
2. Постфиксная — сначала действие, потом приращение.

Пример:

```
#include<stdio.h>
#include<windows.h>
int main()
{
int a,b,c,d,r,s;
float g,f,h;
char *pChar;
int *pInt;
a=b=1;
s=a+b++;
printf("a=%d b=%d s=%d\n",a,b,s);
c=d=3;
r=c+ ++d;
printf("c=%d d=%d r=%d\n",c,d,r);
c=d=3;
r=c+++d;
printf("c=%d d=%d r=%d\n",c,d,r);
c=d=3;
r=c+++d++;
printf("c=%d d=%d r=%d\n",c,d,r);
c=d=3;
r= --c+d++;
printf("c=%d d=%d r=%d\n",c,d,r);
g=3.5;
f=4.2;
h=++g+--f;
printf("g=%f f=%f h=%f\n",g,f,h);  // c=4.5 f=3.2 h=7.7
char cChar='a';
pChar=&cChar;
printf("cChar=%c, pChar=%p cChar=%c pChar=%p\n",cChar,&cChar,*pChar,pChar);
cChar++;
pChar=&cChar;
printf("cChar=%c cChar=%p \n",cChar,&cChar);

// Поразрядное логическое отрицание (дополнение)
        unsigned char cVar1=140,cVar2;
        cVar2=~cVar1;
        printf("cVar1=%d\tcVar2=%d\n",cVar1,cVar2); // 140 115
        printf("cVar1=%c\t\tcVar2=%c\n",cVar1,cVar2); // M s
        return 0;
}
```

Урок от 24.09.18

Группа бинарных операций

Знак операции	Операция
*	Умножение
/	Деление
%	Остаток от деления
+	Сложение
-	Вычитание
<<	Сдвиг влево
>>	Сдвиг вправо
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
= = (=)	Логическое равно
!=	Не равно
&	Поразрядное логическое И
|	Поразрядное логическое ИЛИ
^	Поразрядное исключающее ИЛИ
&&	Логическое И
||	Логическое ИЛИ
,	Последовательное вычисление

Операция ”остаток от деления” (%) имеет результатом остаток от деления первого операнда на второй. Знак результата зависит от конкретной реализации. В Visual C++ знак результата совпадает со знаком первого операнда.

Пример:/* Остаток от деления */

```
int iVar1=49, iVar2=10,iVar3,iVar4,iVar5,iVar6;
iVar3=iVar1%iVar2;
iVar4=-iVar1%-iVar2;
iVar5=iVar1%-iVar2;
iVar6=-iVar1%iVar2;
printf("++=%d\t--=%d\t+-=%d\t-+=%d\n",iVar3,iVar4,iVar5,iVar6);
```

Операции сдвига сдвигают операнд влево (<<) или вправо (>>) на число битов, заданное вторым операндом. Оба операнда должны быть целыми величинами.

Замечание: В языке Си ”истина” считается любое число, не равное 0.

Пример:

```
bool bBool1,bBool2,bBool3,bBool4,bBool5;
bBool1=-31;
bBool2=0;
bBool3=!bBool1;
bBool4=true;
bBool5=false;
printf("bBool1=%d\tbBool2=%d\tbBool3=%d\n",bBool1,bBool2,bBool3);
printf("bBool4=%d\tbBool5=%d\n",bBool4,bBool5);
```

Операция поразрядного логического **И (&)** сравнивает каждый бит первого операнда с соответствующим битом второго операнда по правилу:

0	1	0	1
0	0	1	1
0	0	0	1

Операция поразрядного логического **ИЛИ (|)** сравнивает каждый бит первого операнда с соответствующим битом второго операнда по правилу:

0	1	0	1
0	0	1	1
0	1	1	1

Операция поразрядного исключающего **ИЛИ** (^)сравнивает каждый бит первого операнда с соответствующим битом второго операнда по правилу:

0	1	0	1
0	0	1	1
0	1	1	0

Логические операции:

Операция **&&** (логическое **И**) (значение обоих операндов - истина):

0	1	0	1
0	0	1	1
0	0	0	1

Операция || (логическое **ИЛИ**)(значение хотя бы одного операнда -истина):

0	1	0	1
0	0	1	1
0	1	1	1

Операция **!** (**НЕ)** (истина, если была ложь и наоборот).

Операция последовательного вычисления (,) используется обычно для вычисления двух или более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычисляет два операнда последовательно слева направо.

Пример:

```
for(i=0, j=10; i<MAX && 2*j<MAX; i++)
```

```
#include<stdio.h>
int main()
{
        int iVar1=8, iVar2;
        printf("Right %d\t%d\t%d\n",iVar1>>1,iVar1>>2,iVar1>>3);        // сдвиг вправо
        printf("Left %d\t%d\t%d\n",iVar1<<1,iVar1<<2,iVar1<<3);         // сдвиг влево
        iVar1=-8;
        printf("Right %d\t%d\t%d\n",iVar1>>1,iVar1>>2,iVar1>>3);        // сдвиг вправо
        printf("Left %d\t%d\t%d\n",iVar1<<1,iVar1<<2,iVar1<<3);         // сдвиг влево
        printf("Input iVar1,  iVar2\t");
        scanf("%d%d",&iVar1,&iVar2);
        printf("iVar1&&iVar2=%d\n",iVar1&&iVar2);                       // логическое И
        printf("iVar1||iVar2=%d\n",iVar1||iVar2);                       // логическое ИЛИ
        printf("!iVar1=%d\n",!iVar1);                                   // отрицание НЕ
        return 0;
}
```

Поразрядные логические операции

```
        int iVar1=0x5f, iVar2=0xa9;
        int iResult;
        iResult=iVar1&iVar2;            // поразрядное логическое И
        printf("iResult=%xh\n",iResult);  //  9h
        iResult=iVar1|iVar2;            // поразрядное логическое ИЛИ
        printf("iResult=%xh\n",iResult); // ffh
        iResult=iVar1^iVar2;            // поразрядное искл. ИЛИ
        printf("iResult=%xh\n",iResult); // f6h
        int iVar3=0x25,iVar4=0xa4;
```

```
iVar3+=iVar4;
printf("iVar3=%oo\tiVar3=%dd\tiVar3=%Xh\t",iVar3,iVar3,iVar3);
// 311 201 c9
```

Операции присваивания

Знак операции	Название операции
++	Увеличение (инкремент)
--	Уменьшение (декремент)
=	Простое присваивание
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Целочисленное деление с присваиванием
<<=	Сдвиг влево с присваиванием
>>=	Сдвиг вправо с присваиванием
&=	Поразрядное логическое И с присваиванием
|=	Поразрядное логическое ИЛИ с присваиванием
^=	Поразрядное исключающее ИЛИ с присваиванием

Примеры:
a+=20;
d%=3.0;
x*=3*y+12;

Терарная операция.

Терарная операция включает в себя выражение с тремя операндами и имеет формат:

операнд1?орперанд2:операнд3;

Операнд1 – арифметическое или логическое выражение-условие.
Операнд2 – результат по выполнению условию.
Операнд3 – результат по не выполнению условию.

Пример:

```
int x, y=-5;
x=(y<=0)?-y:y;
printf("x=%d\n",x);                    // x=5
```

Если y<=0 => x=-y
y>0 =>x=y

Приоритеты операций и порядок выполнения

Приоритет	Знак операции	Тип операции
1	**() [] . -> sizeof()**	Круглые и квадратные скобки, точка, стрелка
2 (*)	**- ~ ! * & ++ --**	Унарные
3	*** / %**	Мультипликативные
4	**+ -**	Аддитивные
5	**<< >>**	Побитовый сдвиг
6	**< > <= >=**	Неравенство
7	**== !=**	Равенство
8	**&**	Поразрядное И
9	^	Поразрядное исключающее ИЛИ
10	|	Поразрядное ИЛИ
11	**&&**	Логическое И
12	||	Логическое ИЛИ
13	**?:**	Условное выражение

14 (*)	= *= /= %= += -= <<= >>=	Простое и составное присваивание
15	,	Последовательное вычисление

Примеры:

y=x/2+a*5%10; // y=(x/2)+((a*5)%10);

a=b+++c/5; // a=(b++)+(c/5);

Преобразование типов:

Рассмотрим три правила преобразования типов.

1. Если операция выполняется над данными двух различных типов, обе величины приводятся к “высшему” из двух типов.
2. Применение ключевого слова “unsigned” повышает ранг соответствующего типа.
3. В операторе присваивания конечный результат вычисления приводится к типу переменной, которой должно быть присвоено это значение.
   Существует возможность точно указать тип данных, к которому необходимо привести некоторую величину. Этот способ называется ”приведением типов” и состоит в следующем: перед данной величиной в круглых скобках записывается имя требуемого типа.

   Пример:
   int I Var;
   iVar=1.6+1.7; // 3 (автоматическое преобразование типов)
   iVar=(int)1.6+(int)1.7; // 2
   (float)9 // 9.0

Урок от 1.10.18

Операторы языка Си

Все операторы условно могут быть разделены на категории:

1. Условные операторы (**if, switch**);
2. Операторы цикла (**for, while, do**);
3. Операторы переходов (**break, continue, return, goto**);
4. Операторы “выражение” ,”,”, “пустой оператор”, “составной оператор”.

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор языка Си может быть помечен меткой, состоящей из имени и следующего за ним двоеточия.

Все операторы языка Си, кроме составных операторов, заканчиваются точкой с запятой (;).

I. Оператор ”выражение”

Этот оператор заключается:

a. в вычислении выражения и присваивания;
b. в вызове функций.

Пример:

```
++i;
fltVar=cos(a*5);
swap(&x,&y);
```

II. Пустой оператор

Состоит только из точки с запятой. При выполнении ничего не происходит. Используется в операторах **do, for, while, if** в случаях, когда тело оператора не требуется, хотя по синтаксису требуется хотя бы один оператор. А также, когда необходимо пометить фигурную скобку меткой. Синтаксис требует, чтобы после метки обязательно следовал оператор. Фигурная скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

Пример:

```
void main()
        {
        …
        {
        goto end:;
        …
        }
        end:;
        }
```

III. Составной оператор

Формат:

```
{[объявление]
        …
оператор
[оператор]
        …
  }
```

В конце составного оператора точка с запятой не ставится. Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов.

IV. Оператор **if.**

```
if (выражение)
        оператор1
        [else оператор2]
```

Пример:

```
if  (i<j)
   i++;
else   {
    j=i-3;
    i++;
        }
```

Вычисление оператора **if** начинается с вычисления **выражения**.

Если **выражение** истинно (не ноль), то выполняется **оператор1,** иначе **оператор2**.
Если **выражение** ложно и отсутствует фраза **else**, то выполняется следующий за **if оператор**.
Допускается использование вложенных операторов **if**
Пример:

```
int main()
        {
         int     iVar1=2, iVar2=7, iVar3=3;
                 if (iVar1>iVar2)
                 {
                  if ( iVar2<iVar3)
                          iVar3=iVar2;
                 }
                 else
                          iVar3=iVar1;
                 printf("iVar3=%d\n",iVar3);
        }
```

```
int main()
        {
         int     iVar1=2, iVar2=7, iVar3=3;
                 if (iVar1>iVar2)

                  if ( iVar2<iVar3)
                          iVar3=iVar2;

                 else
                          iVar3=iVar1;
                 printf("iVar3=%d\n",iVar3);
        }
```

V.Оператор **switch**

предназначен для организации выбора из множества вариантов.
Формат:

```
switch (выражение){
    [case константное_выражение:]
      …
    [список операторов]
    [case константное_выражение:]
      …
    [список операторов]
      …
    [default: список оператор]
    }
```

Выражение, следующее за ключевым словом **switch**, может быть произвольным выражением, значение которого должно быть целым.
Константное выражение вычисляется во время компиляции. Оно не может содержать переменные или вызовы функций. Обычно используются целые или символьные константы.
Замечание: Программист должен сам позаботиться о выходе из **case**, иначе все выполняется до конца.
Пример:

```
int     a=2;
switch(a) {
        case 1: func1();
        case 2: func2();
        case 3: func3();
        case 4: func4();
        default: printf("Good bye\n");
        }
```

Выполнение будет идти от **func2()** и до конца.

VI.Оператор **break**
Обеспечивает прекращение выполнения самого внутреннего из объемлющих его операторов **switch, do, for** и **while.** После выполнения оператора **break** управление передается оператору, следующему за прерванным.
Пример:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{

    char cSign;
    float x, y = 2, z = 5;
    printf("Enter the operator sign: (+ - * / ) :");
    scanf_s("%c", &cSign, 1);
    switch (cSign) {
```

```
        case '-':x = y - z;
                 break;
        case '+':x = y + z;
                 break;
        case '*':x = y * z;
                 break;
        case '/':x = y / z;
                 break;
        default: printf("Unknown operation.");
                 x = 0;
        }
        printf("x=%f\n", x);
        system("pause");
        return 0;
}
```

VII. Оператор **for**

Оператор цикла **for** имеет следующий формат:

for([выражение_инициализация];[условное выражение];[выражение_итерация])
оператор

Выражение_инициализация используется для установки начального значения переменных, управляющих циклом.
Условное выражение — выражение, определяющее условие, при котором оператор цикла будет исполняться.
Выражение_итерация определяет изменение переменных, управляющих циклом после каждого выполнения цикла.

Схема выполнения оператора **for**:

1. вычисляется выражение_инициализация;
2. вычисляется условное выражение;
3. если значение условного выражения не равно нулю, выполняется оператор;
4. вычисляется выражение_итерация;
5. вновь вычисляется условное выражение;
6. если условное выражение равно нулю, управление передается на оператор, следующий за оператором **for**.

Замечание: Проверка условия всегда выполняется в начале цикла. Это значит, что цикл может ни разу не выполниться, если условное выражение сразу будет ложным.

Пример:

```
#include<stdio.h>
#include<stdlib.h>
#include<windows.h>
int main()
{


        char cStr[] = "Квадрат числа ";
        int i;
        CharToOem(cStr, cStr);
        for (i = 0; i <= 10; i++)
                printf("%s%2d=%3d\n", cStr, i, i*i);

    system("pause");
    return 0;
    }
```

Результат — квадрат чисел от 0 до 10.
Проверить работу программы, если изменить 7-ую строку на:

```
        for( i=0; i<=10; i++);      // добавили ;
```

VIII. Оператор **while**

Формат:

while(выражение)
оператор

Схема выполнения:

1. вычисляется выражение;
2. если выражение ложно (=0), то тело оператора **while** не выполняется, а управление передается на следующий за **while** оператор;
3. если выражение истинно (не равно нулю), то тело оператора **while** выполняется;
4. процесс повторяется с шага 1.

Пример:

```
int i=0;
while (i < 10)
{
	printf("i = %d\n", i);
	i++;
}
```

IX. Оператор **do**

Используется в тех случаях, когда тело цикла должно выполняться хотя бы один раз.

Формат:

```
do
	{
	оператор;
	}
	while (выражение);
```

Схема выполнения оператора **do**:

1. выполняется оператор;
2. вычисляется выражение. Если выражение не равно нулю, то выполнение продолжается с шага 1.

X. Оператор **continue**

Этот оператор работает подобно оператору break, но в отличие от break прерывает выполнение тела цикла и передает управление на следующую итерацию.

Формат: continue

Пример: /* вывод четных чисел*/

```
#include<stdio.h>
#include<stdlib.h>
int main()
	{
		int iVar;
		for (iVar = 0; iVar < 100; iVar++)
		{
			if (iVar % 2) continue;
			printf("%d\n", iVar);
		}

		system("pause");
	return 0;
	}
```

Оператор **return**

Оператор завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию. Управление передается в вызывающую функцию в точку, непосредственно следующую за вызовом.

Формат: **return [выражение];**

Значение выражения возвращается в вызывающую функцию в качестве значения вызываемой функции.
Если выражение опущено, то возвращаемое функцией значение не определено.
Если функция не возвращает значения, ее следует объявить с типом void.
То есть, имеем два случая использования:

1. надо немедленно выйти из функции;
2. функция должна возвращать значение.

Пример:

```
int sum(int iVar1, int iVar2)
```

```
        {
        return (iVar1+iVar2);
        }
```

Пример:

```
    void print(char x)
        {
            if (x==0)
            { printf("Это плохой аргумент\n",);
            return;
            }
            printf("Введите аргумент %c\n",x);
        }
```

Оператор **return** используется для выхода из функции в случае, если аргумент равен нулю.

XI. Оператор **goto**

Рекомендуется не использовать.

Формат:

```
    goto    имя;
            …
            имя: оператор;
```

Метка — просто идентификатор.

Программы – примеры использования операторов языка Си

```
/* Метод дихотомии */
#include<stdio.h>
#include<stdlib.h>
double fun(double);
int main()
{
    double s, a = 0., b = 1.8, dt = 1.5, e = 1E-6;
    while (dt > e)
    {
        s = (a + b) / 2;
        if (fun(a)*fun(s) < 0)
            b = s;
        else
            a = s;
        dt = b - a;
        printf("dt=%f\ta=%f\tb=%f\n", dt, a, b);
    }
    printf("x=%f\tdt=%f\n", a, dt);
    printf("s=%10.9lf\n", fun((b + a) / 2));

        system("pause");
    return 0;
    }

double fun(double c)
{
    return (c*c - 3 * c + 1);
}
```

```
#include<stdio.h>
#include<stdlib.h>
sum(int, int);

int  main()
{
    int s, a = 10, b = 20;
    s = sum(a, b);
    printf("s=%d\n", s);
    int i = 0, j = 0, j1 = 0;
```

```
	while (j < 100)
	{
		j += ++i;
		printf("i=%d\tj=%d\n", i, j);
		if (j > 50)
		{
			j1 = 1;
			goto m;
		}
	}
m:;
	printf("j1=%d\n", j1);

		system("pause");
	return 0;
}

int sum(int iVar1, int iVar2)
{
	return (iVar1 + iVar2);
}
```

Урок от 8.10.18

Структурированные типы данных

1. **Массивы.**
2. **Структуры.**
3. **Перечисления.**
4. **Смеси.**

Массивы

Определение: Массивы — логически связанные группы элементов одинакового типа.
Ко всему массиву целиком можно обращаться по имени. Также можно выбирать любой элемент массива. Число элементов массива (размер массива) задается при его объявлении и в дальнейшем не меняется (массив в Си является статическим).
Элементы массива нумеруются (индексируются) от 0 до размер массива-1.
Массивы определяются также как и переменные.

Пример:
int a[100]; // массив из 100 элементов целого типа a[0], …,a[99]
char b[30]; // 30 элементов типа char
float c[42]; // 42 элемента типа float

Инициализация массивов
Начальное значение массиву можно присвоить, указав список значений.
Пример:

```
int iVect[] = { 1,2,3,4 };
char cStr[] = { 'm','a','p','\0' };
char cStr1[] = "map";
float fVect[3] = { 0.5, 1.7, 3.8 };
```

соответственно **4, 3** и **3**. В четвертом случае размерность массива указана явно и равна **3**.

Пример: // Проверка

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    char cStr[] = { 'm','a','p','\0' };
    char cStr1[] = "map";
    printf("cStr=%d\tcStr1=%d\n", sizeof(cStr), sizeof(cStr1));

            system("pause");
    return 0;
}
```

Если размер явно указан, то задание большего числа элементов в списке инициализации будет ошибкой.

Если в списке элементов инициализации недостает элементов, всем остальным элементам массива присваивается значение **0**.

Пример:

```
int iVect1[3]={1,2,3,4};   // ошибка
int iVect2[5]={1,2,3} ;    // равнозначно {1,2,3,0,0}
```

Замечание: Такой вид присваивания массиву ошибочен

```
int iVect2[5];
. . .
iVect2={1,2,3,4,5};
```

Многомерные массивы

Двухмерный массив представляется как одномерный массив, элементы которого являются тоже массивами, то есть как массив массивов.

Пример:
char cArr[10][20];

Элементы двухмерного массива хранятся по строкам (то есть быстрее всего изменяется крайний правый индекс).

a[5][9] — десятый элемент шестой строки.

В языке Си существует сильная взаимосвязь между указателями и массивами.
Пусть дано:
int a[5]; // массив из 5 элементов типа int.
int *y; // y – указатель на тип int.

Тогда:
Так как имя массива есть адрес его нулевого элемента, то оператор **y=&a[0];** эквивалентен **y=a;**
Оператор
y=&a[2]; присваивает переменной **y** адрес третьего элемента массива.
Если указатель **y** указывает на очередной элемент массива **a**, то **y=y+1;** указывает на следующий элемент массива.
В нашем примере **y** указывает теперь на элемент **a[3]**.
Соответственно, через оператор * и указатель **y**, можно получить значение элемента массива.
Так, следующие два оператора:
int c=a[3];
int c=*y;
эквивалентны.
Пусть **y=&a[0];** //(**y=a;**)
Элемент
a[i] можно представить как ***(a+i),** или ***(y+i),** или **y[i].**

Таким образом, любой массив и индексное выражение в массиве можно представить посредством указателей.

Замечание: Есть различие между указателем и именем массива.
Указатель — это переменная и **y=a;** или **y++**; – допустимые операции.
Имя массива — константа. Выражения вида **a=y; a++;** – использовать нельзя, так как a – константа и не может быть изменена.

Замечание: Если **a** адрес, то нельзя использовать оператор
y=&a;
(так как **a** – адрес).

/* Если указатели адресуют элементы одного массива, то их можно сравнивать **(>, <, ==, !=).**

Нельзя сравнивать, либо применять в арифметических операциях указатели на разные массивы. (**NULL** вместо нуля). */

Указатели на элементы одного массива также можно вычитать. Тогда результатом будет число элементов массива, расположенных между уменьшаемым и вычитаемым объектами.

```
#include<stdio.h>
#include <windows.h>
#include<stdlib.h>
int  main()
{
        int iArr[10];
        int *iPtr1, *iPtr2, iVar;
        char cStr[] = "Введите элементы массива ";
        CharToOem(cStr, cStr);
        printf("%s\n", cStr);
        for (int i = 0; i < 10; i++)
```

```
		scanf_s("%d", &iArr[i]);

	for (int i = 0; i < 10; i++)
		printf("iArr[%d]=%2d\n", i, iArr[i]);

	// Работа с указателями
	iPtr1 = iArr;
	iPtr2 = &iArr[0];
	printf("iArr=%p\tiPtr1=%p\tiPtr2=%p\n", iArr, iPtr1, iPtr2);
	printf("iArr[2]=%d\t*(iArr+2)=%d\n", iArr[2], *(iArr + 2));
	printf("iPtr1[2]=%d\t*(iPtr1+2)=%d\n", iPtr1[2], *(iPtr1 + 2));

	iPtr1 = &iArr[2];
	iPtr2 = &iArr[6];
	iVar = iPtr2 - iPtr1;
	printf("iVar=%d\n", iVar);

	// Операторы с ошибками
//		iArr++;
//		iArr=iPtr1;
//		iPtr1=&iArr;


		system("pause");
	return 0;
}
```

Определение массива с инициализацией:

Пример:// Инициализация массивов

```
#include<stdio.h>
#include <windows.h>
#include<stdlib.h>
int  main()
{
	int i, j;
	int a[2][3] = { 1,3,5,7,9,11 };
	int b[2][3] = { {1,3,5},{7,9,11} };
	int c[2][3] = { 1,3,5,7 };
	int d[2][3] = { {1,3},{5,7} };
	//	int e[2][3]={{1,3},{5,7},{9,11}};

	for (i = 0; i < 2; i++) {
		printf("\n");
		for (j = 0; j < 3; j++)
			printf(" a[%d][%d]=%d", i, j, a[i][j]);
	}


	for (i = 0; i < 2; i++) {
		printf("\n");
		for (j = 0; j < 3; j++)
			printf(" b[%d][%d]=%d", i, j, b[i][j]);
	}


	for (i = 0; i < 2; i++) {
		printf("\n");
		for (j = 0; j < 3; j++)
			printf(" c[%d][%d]=%d", i, j, c[i][j]);
	}

	for (i = 0; i < 2; i++) {
		printf("\n");
```

```
            for (j = 0; j < 3; j++)
                printf(" d[%d][%d]=%d", i, j, d[i][j]);
        }
        printf("\n");

            system("pause");
        return 0;
    }
```

Пример: /* Умножение двух матриц */

$$C = A \cdot B; \quad c_{ij} = \sum_{k=1}^{K} a_{ik} \cdot b_{kj}; \quad i = 1,\cdots,3; \quad j = 1,\cdots,2; \quad k = 1,\cdots,4;$$

$$A = \begin{pmatrix} 1 & 2 & 5 & 3 \\ 2 & 4 & 0 & 1 \\ 3 & -1 & 2 & 4 \end{pmatrix}; \quad B = \begin{pmatrix} 2 & -1 \\ 3 & 2 \\ 3 & 4 \\ 2 & -2 \end{pmatrix}; \quad C = A \cdot B = \begin{pmatrix} 29 & 17 \\ 18 & 4 \\ 17 & -5 \end{pmatrix}.$$

```
#include<stdio.h>
#include <windows.h>
#include<stdlib.h>
int  main()
{
        int A[3][4] = { {1,2,5,3},
                        {2,4,0,1},
                        {3,-1,2,4}
        };
        int B[4][2] = { {2,-1},
                        {3,2},
                        {3,4},
                        {2,-2}
        };
        int C[3][2];
        int i, j, k;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 2; j++) {
                C[i][j] = 0;
                for (k = 0; k < 4; k++)
                    C[i][j] += A[i][k] * B[k][j];
            }

        for (i = 0; i < 3; i++) {
            for (j = 0; j < 2; j++) {
                printf("C[%d][%d]=%d\t", i, j, C[i][j]);
            }
            printf("\n");
        }


            system("pause");
        return 0;
    }

// массивы и указатели

        int array[5]={1,2,3,4,5};
        int *iPtr;
        iPtr=array;         // iPtr=&array[0];
        printf("\n\n*iPtr=%d array[0]=%d\n",*iPtr,array[0]);
```

```
//  доступ к данным
printf("\narray[3]=%d *(array+3)=%d iPtr[3]=%d *(iPtr+3)=%d\n",array[3],*(array+3),iPtr[3],*(iPtr+3));
```

```
#include<stdio.h>
#include <windows.h>
#include<stdlib.h>
int  main()
{
                    //определение указателей
                    int i = 100;
                    int *iPtr;
                    iPtr = &i;
                    printf("iPtr=%p adr &i=%p\n", iPtr, &i);
                    printf("*iPtr=%d i=%d\n", *iPtr, i);
                    int *iPtr1, *iPtr2, *a;

                    // унарные операции
                    int *iPtr0, s = 5;
                    iPtr0 = &s;
                    printf("iPtr0=%p\n", iPtr0);
                    iPtr0++;
                    printf("iPtr0=%p\n", iPtr0);
                    iPtr0--;
                    printf("iPtr0=%p\n", iPtr0);

                    // бинарные операции
                    int i1, j1;
                    a = iPtr0;
                    iPtr1 = a + 4;
                    iPtr2 = a + 9;
                    i1 = iPtr1 - iPtr2;
                    j1 = iPtr2 - iPtr1;
                    printf("i1=%d j1=%d\n", i1, j1);

                    // операторы сравнения
                    int *iPtr3, *iPtr4, *z, z0 = 10;
                    z = &z0;
                    iPtr3 = z + 9;
                    iPtr4 = z + 7;
                    if (iPtr3 > iPtr4) *z = 4;
                    printf("z0=%d\n", *z);


                    system("pause");
          return 0;
}
```

Строковые литералы, массивы и указатели

Строковые литералы — последовательность символов, заключенная в двойные кавычки. В строковом литерале на один символ больше, чем используется при записи. Он всегда заканчивается нулевым символом **'\0'**.

Пример:

```
sizeof("Бор"); // значение 4
```

Строковый литерал можно присвоить переменной типа **char***
Но, изменение литерала через такой указатель является ошибкой.

Пример:

```
#include<stdio.h>
#include <windows.h>
#include<stdlib.h>
int  main()
```

```
{
		char cStr[] = "Mars";
		cStr[2] = 'p';	// ошибки нет
		printf("cStr=%s\n", cStr);
		char* pStr = "Mars";	//
		// pStr[2] = 'p';// ошибка при выполнении программы
		printf("pStr=%s\n", pStr);
		system("pause");
	return 0;
}
```

Урок от 22.10.18

Объявление объектов и типов.

Все переменные в Си должны быть объявлены. Объявление происходит с помощью ключевого слова – спецификатора типа, идентификатора и модификаторов.

Модификатор	Определяемый тип
*	указатель
[]	массив
()	функция

Можно использовать одновременно более одного модификатора, что позволяет создавать множество сложных описателей типов.

Замечание: Некоторые комбинации недопустимы, например:

1. элементами массива не могут быть функции:
2. функции не могут возвращать функции.

При интерпретации сложных описателей квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед звездочкой (слева от идентификатора).

Квадратные или круглые скобки имеют один и тот же приоритет и связываются слева направо.

Спецификатор типа рассматривается на последнем шаге, когда описатель уже полностью проинтерпретирован. Можно использовать круглые скобки, чтобы изменить порядок интерпретации.

Для интерпретации можно использовать простое правило: «изнутри наружу», состоящее из 4-х шагов.

1. Начать интерпретацию с идентификатора и посмотреть вправо, есть ли квадратные или круглые скобки.
2. Если они есть, то проинтерпретировать эту часть описателя и затем посмотреть налево в поисках звездочки.
3. Если на любой стадии справа встретится закрывающая круглая скобка, то вначале необходимо применить все эти правила внутри круглых скобок, а затем продолжить интерпретацию.
4. Интерпретировать спецификатор типа.

Пример:

int *(*comp[10])()
6 5 3 1 2 4

В примере объявляется переменная **comp (1)** как массив (**2**) из 10 указателей (**3**) на функции (**4**), возвращающие указатели (**5**) на целые значения (**6**).

```
/* Что напечатает программа ? */
#include <stdio.h>
int main()
{
 char note[]="You are welcome!";
 char *ptr;

 ptr=note;
 puts(ptr);
```

```
 puts(++ptr;
 note[7]='\0';
 puts(note);
 puts(++ptr);
return 0;
}
```

// Примеры объявления с использованием модификаторов

```
int board [8][8];
int **ptr;
int *risk[10];
int (*wis)[10];
int (*oop)[3][4];
char *fun();
char (*fun)();
char (*fun())[3];
```

Пример: /* Многоуровневые ссылки */

```
#include <stdio.h>
char *m[]={"January","February","March","April","May","June","July",
"August","September","October","November","December"};
char **mp[]={m+11,m+10,m+9,m+8,m+7,m+6,m+5,m+4,m+3,m+2,m+1,m};
char ***mpp[]={mp,mp+9,mp+6,mp+3};
char ****mppp=mpp;

int main()
{
        int i;
        printf("%s\n",***++mppp);
        printf("%s\n",***--mppp);
        for(i=0;i<12;i++)
                printf("%s%c",mppp[0][0][i-11],(i==6)?'\n':' ');
        printf("\n");
        for(i=0;i<12;i++)
                printf("%s%c",mppp[1][2][11-i],(i==5)?'\n':' ');
        printf("\n");
        printf("%s\n",*mppp[3][1]);
        printf("%s\n",**mppp[1]+3);
        printf("%s\n",mppp[2][-1][3]+2);
        for(i=0;i<8;i++)
                printf("%s\n",mppp[3][-2][-9]+i);
        return 0;
}
```

Результаты выполнения программы:

Пример:

```
#include <stdio.h>
char *c[]={"TEST1 ","NOP ","FILE ","FIRST "};
```

```
char **cp[]={c+3,c+2,c+1,c};
char ***cpp=cp;

int main()
{
        printf("%s ",**++cpp);
        printf("%s ",*--*++cpp+3);
        printf("%s",*cpp[-2]+3);
        printf("%s\n",cpp[-1][-1]+1);
        return 0;
}
```

Результаты выполнения программы:

Пример:/* нахождение определителя матрицы */

```
#include<stdio.h>
#include<windows.h>
#define n 4

void main()
{
        float a[n][n],koef,det=1;
        int i,j,k;
        char cStr[]="Введите матрицу n на n";
        CharToOem(cStr,cStr);
        printf("%s\n",cStr);
        for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                        scanf("%f",&a[i][j]);

                // Прямой ход

                for(i=0;i<n-1;i++){             //      идем по строкам
                        for(j=i+1;j<n;j++){     //по соотв. столбцу
                                koef=a[j][i]/a[i][i];
                                for(k=i;k<n;k++)        //по столбцам
                                        a[j][k]-=a[i][k]*koef;}}

                // Вывод результатов
                for(i=0;i<n;i++){
                        for(j=0;j<n;j++)
                                printf("a[%d][%d]=%5.2f\t",i,j,a[i][j]);
                                printf("\n");}
                for(i=0;i<n;i++)
                        det*=a[i][i];
                printf("det=%5.2f\n",det);
}
```

Пример: /* решение СЛАУ методом Гаусса */

```
#include<stdio.h>
#include<windows.h>
```

```
#define n 4

void main()
{
	float a[n][n+1],x[n],koef,det=1,y=0;
	int i,j,k;
	char cStr[]="Введите матрицу n на n+1";
	CharToOem(cStr,cStr);
	printf("%s\n",cStr);
	for(i=0;i<n;i++)
		for(j=0;j<n+1;j++)
			scanf("%f",&a[i][j]);

		// Прямой ход метода Гаусса

		for(i=0;i<n;i++){				// идем по строкам
			for(j=i+1;j<n;j++){	//  по соотв. столбцу
				koef=a[j][i]/a[i][i];
				for(k=i;k<n+1;k++) // по столбцам
					a[j][k]-=a[i][k]*koef;}}

		// Вывод результатов


		for(i=0;i<n;i++){
			for(j=0;j<n+1;j++)
				printf("a[%d][%d]=%5.2f\t",i,j,a[i][j]);
				printf("\n");}
		for(i=0;i<n;i++)
			det*=a[i][i];
		printf("det=%f\n",det);

		// Обратный ход.
		x[n-1]=a[n-1][n]/a[n-1][n-1];
		for(i=n-2;i>=0;i--){
			y=0;
			for(j=n-1;j>i;j--)
				y+=a[i][j]*x[j];
				x[i]=(a[i][n]-y)/a[i][i];}
		for(i=0;i<n;i++)
			printf("x[%d]=%5.2f\n",i+1,x[i]);
}
```

Структуры

Структура – тип данных, который объединяет несколько переменных, в общем случае разных типов. Переменные, которые объединены структурой, называются элементами, полями, членами структуры.

Пример: /* объявления структуры */

```
struct student{char name[30];
               int course;
               char group;
               int scholarship;};
```

Объявление структуры является оператором, то есть в конце должна быть “;”.
Таким образом определяется шаблон структуры.
struct – тип структуры, **student** – имя структуры. Все вместе это определяет шаблон структуры, память не выделяется.
Определим переменные типа **struct student:**

struct student stud1,stud2;

Компилятор автоматически выделяет под них место в памяти компьютера. Под каждую переменную выделяется непрерывный участок памяти.
Задание шаблона и объявление переменных можно производить в одном операторе:

```
struct student{char name[30];
               int course;
               char group;
               int scholarship;} stud1, stud2;
```

Доступ к конкретному элементу структуры осуществляется с помощью операции “точка”.

Пример:

```
strcpy(stud1.name,”Степанов А.В.”);
printf(“%s\n”,stud2.group);
```

Массивы структур

1. Задаем шаблон структуры.
2. Объявляем массив.

Пример:

struct student studcourse[200];

Доступ к полю name 101- го элемента массива: **studcourse[100].name**

studcourse[100].name[7] – 8 - й элемент (буква) поля name 101-го элемента массива studcourse.
Если объявлены две переменные типа структуры с одним и тем же шаблоном, то можно использовать операцию присваивания:

```
stud1=stud2;
```

Замечание: нельзя использовать операцию присваивания переменных типа структуры, шаблоны которых описаны под разными именами.
Пример:

```
second=first;          // неправильно
second.a=first.a;              // правильно
```

Можно создать указатель на структуру:

```
struct student *pStru; // pStru – переменная типа указатель на структуру student
```

Операция стрелка (->) употребляется вместо операции точка (.), когда хотят получить значение элемента структуры с применением переменной типа указатель.

Пример: /* комплексные числа */

```
#include <stdio.h>
#include <stdlib.h>

	int main()
	{
		struct complex {
			float x;
			float y;
		} z;
		struct complex *p;	// объявление указателя
		p = &z;
		(*p).x = 1;
		p->y = 2;
		printf("Re z = %f\tIm z = %f\n", z.x, z.y);


	system("pause");
	return 0;
}
```

В качестве элементов структуры можно использовать массивы, структуры и массивы структур.

Пример: /* использование в качестве элемента структуры другой структуры */

```
struct address{
	char city[30];
	char street[30];
	int house;
		};
struct fulladdress{
	struct address addr;	//
	int room;
	char name[30];
}AddrStud;
…
AddrStud.addr.house=15; // присвоение значения house структуры address переменной AddrStud
```

Пример:

```
struct date{
	int day;
	int month;
	int year;
	};
struct student{
	char name[20];
	char surname[20];
```

```
        struct date birthday;
        int group;
        int age;
                };
struct student students[21];
        …
students[4].group=119;
students[4].birthday.day=3;
students[4].birthday.month=8;
students[4].birthday.year=1988;
```

Инициализация

```
struct date boy={3,8,1988};
```

```
/* Что напечатает программа?  */
#include <stdio.h>
#include <stdlib.h>

struct house {
      float ds;
      int rooms;
      int stories;
      char *address;
};

      int main()
      {
            struct house fr = { 25000, 6, 5, "Baker street" };
            struct house *sign;

            sign = &fr;
            printf("%d %d\n", fr.rooms, sign->stories);
            printf("%s\n", fr.address);
            printf("%c %c\n", sign->address[7], fr.address[10]);



      system("pause");
      return 0;
}
```

Объединения (union)

Существует еще один тип данных для размещения в памяти нескольких переменных разного типа. Объявляется объединение так же, как и структура.

```
union u {
            int iVar;
            char cStr;
            long int lVar;
```

};

Это шаблон объединения.
Пример:
union u alfa,beta; // можно с шаблоном

В отличие от структуры для переменной типа union места в памяти выделяется ровно столько, сколько надо элементу объединения, имеющему наибольший размер в байтах.

Остальные переменные будут располагаться в том же месте памяти. Синтаксис использования элементов объединения такой же, как и для структуры:
u.cStr=’5’;
Для объединений также разрешена операция ->, если мы обращаемся к объединению с помощью указателя.

```
#include <stdio.h>
#include <stdlib.h>

	int main()
	{
		union u {
			int iVar;
			char cStr;
			long int lVar;
			double fVar;
		};
		union u alfa; //
		union u *ptr;
		ptr = &alfa;
		ptr->iVar = 5;
		printf("d=%d\n", sizeof(alfa));
		printf("%p\tiVar=%d\n", &alfa.iVar, alfa.iVar);
		ptr->fVar = 5.6;
		printf("%p\tfVar=%f\n", &alfa.fVar, alfa.fVar);

	system("pause");
	return 0;
}
```

Перечисления

Перечислимый тип (enumeration).
Этот тип – множество поименованных целых констант.
Перечислимый тип определяет все допустимые значения, которые могут иметь переменные этого типа.
Формат:
enum имя_типа {список_названий} список переменных;

Пример:

enum seasons {win,spr,sum,aut};
enum seasons s;

Каждое из имен win,spr,sum,aut представляет собой целую величину. По умолчанию они соответственно равны 0,1,2,3.
Оператор

printf(“%d\t%d”,win,aut);

выдаст числа 0 и 3.
Во время объявления типа можно одному или нескольким именам присвоить другие значения:

enum value {one=1,two,three,ten=10,thausand=1000,next};

Если распечатать
printf(“%d\t%d\t%d\t%d\t%d\t%d\n”,one,two,three,ten,thausand,next);
то на экране появятся числа 1 2 3 10 1000 1001, то есть каждый следующий символ увеличивается на 1 по сравнению с предыдущим, если нет другого присвоения.
С переменными перечислимого типа можно производить следующие операции:

1. Присваивать переменную типа ***enum*** другой переменной того же типа.
2. Провести сравнение с целью выяснения равенства или неравенства.
3. Арифметические операции с константами типа ***enum*** (i=win-aut);

Нельзя использовать арифметические операции и операции ++ и -- для переменных типа ***enum***.
Основная причина использования перечислимого типа – это улучшение читаемости программ.

```
#include <stdio.h>
#include <stdlib.h>

	int main()
	{
		enum seasons { win, spr, sum, aut } s, s1;
		printf("%d\t%d\t%d\t%d\t", win, spr, sum, aut);
		s = win;
		s1 = aut;
		int i = spr;
		//		sum=3;		//ошибка
		printf("\ns=%d\ts1=%d\ti=%d\n", s, s1, i);


	system("pause");
	return 0;
}
```

1. **<u>Пример</u>: /* Вычисление e x */**

$$f(x) = e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Написать программу вычисления экспоненты с точностью 1E-6.

2. **<u>Числа Фибоначчи. Золотое сечение.</u>**

Числа Фибоначчи (Леонардо Пизанский [1180 - 1240]) – числовая последовательность [1228], каждый член которой равен сумме двух его предыдущих членов.
Рекуррентная формула: $a_{n+1} = a_n + a_{n-1}, a_1 = a_2 = 1.$
Золотое сечение – гармоническое деление отрезка длины a точкой x на две части в соотношении: (**) $a : x = x : (a - x)$. Решение этого уравнения: (*)

$x = (\sqrt{5} - 1)\, a/2 \approx 0.62a$. Решение также приближенно равно a_{n+1}/a_n, где a_i – числа последовательности Фибоначчи.
Написать программу, в которой запрашивается количество членов последовательности и далее находится значение Золотого сечения. Вывести решение (*) при $a = 1$. Проверить для (**).
Замечание: для использования математических функций необходимо использовать math.h

3. **Комплексные числа**
Используя структуры, написать сложение, вычитание, умножение и деление комплексных чисел.

Урок от 29.10.18

Функции

Функция – это независимая совокупность объявлений и операторов, предназначенных для выполнения определенной задачи. В языке Си нет процедур, подпрограмм, есть только функции. Каждая функция должна иметь имя, которое используется для вызова функции. Имя одной из функций – main(), зарезервировано. Эта функция должна присутствовать в каждой программе, с нее начинается выполнение программы, хотя она необязательно должна следовать первой в программе.
При вызове функции ей могут быть переданы данные посредством аргументов функции. Функция может возвращать и не возвращать значение. Возвращаемое значение и есть основной результат выполнения функции, который при выполнении программы подставляется на место вызова функции.
С использованием функций в языке Си связаны три понятия:

1. Объявление функции.
2. Определение функции.
3. Вызов функции.

Объявление функции (прототип) задает имя функции, типы и число формальных параметров, тип значения, возвращаемого функцией, и класс памяти.
Объявление функции, в отличие от определения функции, является оператором, то есть за заголовком функции следует точка с запятой. Заголовок функции может заканчиваться списком имен. Это имена других функций, возвращаемых значение того же типа, который задан спецификатором типа для первого описателя функции. Компилятор использует прототип функции для сравнения типов фактических параметров в вызове функции с типами формальных параметров. Прототипы библиотечных функций находятся в файлах включения, поставляемых в составе системы программирования и включаемых в программу с помощью директивы препроцессора #include.

Определение функции задает имя функции, типы и число формальных параметров, тип значения, возвращаемого функцией, объявления и операторы, которые определяют действие функции (тело функции).
Формат определения функции:
[спецификатор класса памяти][спецификатор типа] имя([список формальных параметров])
тело функции
Необязательный спецификатор класса памяти задает класс памяти, который может быть или static или extern. Если спецификатор не задан, то по умолчанию extern.
Спецификатор типа может задавать любой основной тип. Если тип не задан, то по умолчанию принимается int.
Имя функции может быть задано со звездочкой, что означает, что функция возвращает указатель. Функция не может возвращать массив или функцию, но может возвращать указатели на любой тип, включая массивы и функции. Функция возвращает значение, если ее выполнение заканчивается выполнением оператора return в теле функции следующего формата:
return выражение;
Выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения в определении (объявлении) функции, и управление передается в точку вызова.
Если оператор return не задан или не содержит выражения, то функция не возвращает никакого значения. В этом случае спецификатор типа возвращаемого значения должен быть задан ключевым словом void, означающим отсутствие возвращаемого значения.
Формальные параметры – это переменные, которые принимают значения, передаваемые функции во время вызова.
Список формальных параметров – это последовательность объявлений формальных параметров, разделенных запятыми. Список может заканчиваться запятой и многоточием. Это означает, что число аргументов функции переменно. Если формальный параметр представлен в списке, но не объявлен, то предполагается, что параметр имеет тип int.
Все переменные, объявленные в теле функции, имеют класс памяти auto, если они не объявлены иначе, и являются локальными.
Параметры функции передаются по значению и могут рассматриваться как локальные переменные. Они инициализируются значениями переданных аргументов и теряются при выходе из функции. Поэтому в теле функции нельзя изменить значение параметра, так как функция работает с копиями аргументов. Однако, если в качестве параметра передать указатель переменной, можно использовать операцию разадресации этой переменной для осуществления доступа и изменения ее значения.
Вызов функции имеет следующий формат:
выражение([список выражений])
Выражение вычисляется как адрес функции. Список выражений представляет собой список фактических аргументов, передаваемых в функцию. Фактический аргумент может быть величиной основного типа,

структурой, перечислением, смесью или указателем. Массивы и функции не могут быть переданы как параметры, но указатели на эти объекты могут быть переданы.
Выполнение вызова функции происходит следующим образом:

1. Вычисляются значения в списке выражений. Если в прототипе функции вместо списка формальных параметров задано ключевое слово void, это значит, что в функцию не передается никаких аргументов и в определении функции не должно быть формальных параметров.
2. Происходит замена формальных параметров на фактические (последовательно).
3. Управление передается на первый оператор функции.
4. Выполнение оператора return в теле функции возвращает управление и, возможно, значение в вызывающую функцию. Если оператор return не задан, то управление возвращается после выполнения последнего оператора тела функции. При этом возвращаемое значение не определено.

Выражение перед скобками вычисляется как адрес функции. Это означает, что функцию можно вызвать через указатель на функцию.

Пример:

```
#include <stdio.h>
#include <stdlib.h>

int(*fun)(int, int), sum(int, int), razn(int, int);

int main()
{
	int a = 10, b = 20, c;
	c = sum(a, b);
	printf("c=%d\tsum=%d\trazn=%d\n", c, sum(a, b), razn(a, b));
	fun = sum;
	printf("*fun=%d\n", (*fun)(a, b));
	fun = razn;
	printf("*fun=%d\n", (*fun)(a, b));

	system("pause");
	return 0;
}

int sum(int x, int y)
{
	return (x + y);
}
int razn(int x, int y)
{
	return (y - x);
}
```

Пример:/* передача аргументов по значению и по адресу*/

```
#include <stdio.h>
#include <stdlib.h>

void swap(int x, int y);
void swap1(int *x, int *y);

int main()
{
	int a = 10, b = 20;
	swap(a, b);
	printf("a=%d\tb=%d\n", a, b);
	swap1(&a, &b);
	printf("a=%d\tb=%d\n", a, b);

	system("pause");
	return 0;
}

void swap(int x, int y)
{
	int temp = x;
```

```
	x = y;
	y = temp;
	printf("x=%d\ty=%d\n", x, y);
}
void swap1(int *x, int *y)
{
	int temp = *x;
	*x = *y;
	*y = temp;
	printf("x=%d\ty=%d\n", *x, *y);
}
```

Для передачи в функцию в качестве фактического параметра массива достаточно указать адрес его начала.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#define n 5
	float Sum(float[]);

	int main()
	{
		float aVar[] = { 1.5,4.5,5.0,10.0,15.5 };
		float rez;
		rez = Sum(aVar) / (float)n;
		printf("Sum=%f\n", rez);

		system("pause");
		return 0;
	}

	float Sum(float aFVar[])
	{
		float f = 0;
		for (int i = 0; i < n; i++)
			f += aFVar[i];
		return f;
	}
```

Замечание: Компилятору не требуется размер массива, подлежащего обработки.

Замечание: В функции можно передавать многомерные массивы.

Допустимые варианты:

```
float Det(float b[][n]);

det_a=Det(d);		// вычисление определителя матрицы системы
float Det(float c[][n])


float Det(float b[n][n]);

det_a=Det(d);		// вычисление определителя матрицы системы
float Det(float c[n][n])
```

```
/* Передача в функцию массива структур*/

#include <stdio.h>
#include <stdlib.h>
#define n 2
struct st {
	int a;
```

```
	int b;
	int c;
};
float middle(struct st[]);


int main()
{
	struct st s1[n];
	s1[0].a = 5;
	s1[0].b = 4;
	s1[0].c = 3;
	s1[1].a = 5;
	s1[1].b = 4;
	s1[1].c = 5;
	float bn = middle(s1);
	printf("sr=%f\n", bn);

	system("pause");
	return 0;
}

float middle(struct st d[])
{
	float sr = (d[0].a + d[0].b + d[0].c + d[1].a + d[1].b + d[1].c) / 6.0;
	return sr;
}
```

Практическое занятие

1. Написать программу решения СЛАУ методом Крамера.
2. Написать программу нахождения обратной матрицы (с проверкой).

Урок от 12.11.18

Область действия переменных и классы памяти

С точки зрения действия переменных различают два типа переменных:

1. локальные
2. глобальные

Локальные переменные – это переменные, объявленные внутри блока, в частности внутри функции. Локальная переменная доступна внутри блока, в котором она объявлена, то есть область действия локальной переменной – блок. Локальные переменные хранятся в стеке (динамически изменяющейся области памяти).
Глобальные переменные – переменные, объявленные вне какой-либо функции. В отличие от локальной переменной глобальные переменные могут быть использованы в любом месте программы, но перед их первым использованием они должны быть объявлены. Область действия – вся программа. Глобальные переменные хранятся в отдельной фиксированной области памяти, созданной компилятором специально для этого.

Модификаторы типа – const, volatile

Переменная, к которой в объявлении применен модификатор const, не может изменять своего значения.
Пример:
const int a=10;
Модификатор volatile указывает компилятору на то, что значение переменной может также измениться независимо от программы, т.е. вследствие воздействия чего-либо, не являющегося оператором программы. Оба модификатора могут применяться совместно. Это означает, что значение переменной может измениться только под воздействием извне.

Классы памяти (для управления механизмом использования памяти)

Каждая переменная принадлежит к одному из четырех классов памяти, которые описываются следующими ключевыми словами:

1. auto – автоматическая;
2. register – регистровая;
3. extern – внешняя;
4. static – статическая.

Тип памяти указывается модификатором – ключевым словом, стоящим перед спецификацией типа переменной.

Пример:

```
register int plus;
static int sum;
```

Если ключевого слова перед спецификацией типа локальной переменной при ее объявлении нет, то по умолчанию она принадлежит классу auto. (Поэтому практически никогда это ключевое слово не используется).

Автоматическая переменная имеет локальную область действия. Они известны только внутри блока, в котором они определены. При выходе из блока автоматические переменные пропадают, эта область памяти освобождается. Автоматические переменные хранятся в оперативной памяти.

Регистровые переменные хранятся в регистрах процессора. Доступ к переменным, хранящимся в регистровой памяти, гораздо быстрее, чем к тем, которые хранятся в оперативной памяти. Во всем остальном регистровые переменные аналогичны автоматическим переменным. Если доступных регистров нет, то переменная становится просто автоматической переменной.

В языке Си при компоновке программы (редактировании связей) к переменной может применяться одно из трех связываний: внутреннее, внешнее или не относящееся ни к одному из этих типов (то есть редактирование связей к ней не применяется). В общем случае к именам функций и глобальных переменных применяется внешнее связывание. Это означает, что после компоновки они будут доступны во всех файлах, составляющих программу. К объектам, объявленным со спецификатором static и видимым на уровне файла, применяется внутреннее связывание. После компоновки они будут доступны только внутри файла, в котором они объявлены. К локальным переменным связывание не применяется и они доступны только внутри своих блоков.

Спецификатор extern указывает на то, что к объекту применяется внешнее связывание, поэтому он будет доступен во всей программе. Один и тот же объект может быть объявлен неоднократно в разных местах, но определен может быть только один раз.

Внешняя переменная extern относится к глобальным переменным. Она может быть объявлена как вне, так и внутри тела функции.

Пример:

```
int f()
{
        extern int j; /* объявление внешней переменной внутри функции */
}
```

Появление ключевого слова extern связано с модульностью языка Си, то есть возможностью составлять многофайловую программу с возможностью раздельной компиляции каждого файла.

Когда в одном из файлов опишем вне тела функции глобальную переменную

```
float global;
```

то для нее выделяется место в памяти в разделе глобальных переменных и констант. Если мы используем эту глобальную переменную в другом файле, то при раздельной компиляции без дополнительного объявления переменной компилятор не будет знать, что это за переменная. Использование объявления

```
extern float global;
```

сообщает, что такая переменная будет описана в другом файле.

Пример: /* Использование спецификатора *extern/**

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
	extern int a, b;
	printf("a=%d\tb=%d\n", a, b);
	system("pause");
	return 0;
}

// описание глобальных переменных a и b
int a = 10, b = 20;
```

Отметим, что глобальные переменные a и b объявлены после main().

Статические переменные вводятся ключевым словом static. Область видимости локальных статических переменных такая же, как у автоматических, но значение переменных сохраняется от одного вызова функции до другого. Локальные статические переменные инициализируются нулем, если не указан другой инициализатор.

Пример: /* описание с инициализацией */

```
static int count=10;
```

Можно описать глобальную (внешнюю) статическую переменную, то есть описать переменную типа static вне любой функции.

Отличие внешней переменной от внешней статической переменной состоит в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, в то время как внешняя статическая переменная только функциями того файла, где она описана, причем только после ее определения.

Все глобальные переменные, и статические и нестатические, инициализируются нулем, если не предусмотрено другой инициализации. Локальные переменные, если нет явной инициализации, инициализируются произвольным числом.

Рассмотрим таблицу, в которой приведены область действия и продолжительность существования переменных разных классов памяти.

Класс памяти	**Ключевое слово**	**Время существования**	**Область действия**
Автоматический	auto	временно	блок
Регистровый	register	временно	блок
Статический локальный	static	постоянно	блок
Статический глобальный	static	постоянно	файл
Внешний	extern	постоянно	программа

Замечание: В программе может быть описано несколько переменных с одним и тем же именем, но в разных блоках.

Замечание: Инициализация статических переменных происходит один раз, во время первого вызова функции. При последующих вызовах повторной инициализации не происходит.

Пример: /*использование статических переменных*/

```
#include <stdio.h>
#include<windows.h>
#include <stdlib.h>
float sr(float);

int main()
{
	float x = 1.;
	char sStr[] = "Введите значение x";
	CharToOem(sStr, sStr);
	while (x != 0)
	{
		printf("%s\n", sStr);
		scanf_s("%f", &x);
		if (x != 0)
			printf("res=%f\n", sr(x));
	}
	system("pause");
	return 0;
}

float sr(float y)
{
	static float res = 0.0;
	static int count = 0;
	count++;
	res += y;
	return res/(float)count;
}
```

Передача параметров из операционной среды

Функция main() может быть определена с параметрами, которые передаются из внешнего окружения, например из командной строки. Принято, что два первых параметра функции main(), имеют имена argc и argv, хотя это и не диктуется языком Си.
Параметр argc определяет общее число параметров, передаваемых функции main(), и объявляется как int. Параметр argv объявляется как массив указателей, каждый элемент которого ссылается на строковое представление аргумента, передаваемое функции main(). Если функции main() передается третий параметр, этот параметр принято называть envp. Параметр envp объявляется аналогично параметру argv как массив указателей на строковые величины, определяющие операционную среду, в которой выполняется программа.

Аргумент argv[0] всегда содержит имя командной строки, поэтому значение argc на 1 превышает количество передаваемых аргументов. Аргументы командной строки могут задавать, например, режимы работы, имена файлов и другие данные для программы и отделяются друг от друга пробелами. (Если пробел должен быть представлен в аргументе, аргумент заключается в кавычки).
Полный заголовок функции main() имеет вид:
int main(int argc, char *argv[], char * envp[])

Пример: /* программа печати количества, значений фактических аргументов, передаваемых функции main(), и переменных среды */

```
#include <stdio.h>
#include<windows.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *envp[])
{
	char cStr1[] = "Количество передаваемых аргументов";
	char cStr2[] = "Имя программы и передаваемые аргументы";
```

```
        char cStr3[] = "Переменные среды";
        CharToOem(cStr1, cStr1);
        CharToOem(cStr2, cStr2);
        CharToOem(cStr3, cStr3);
        printf("%s - %d\n", cStr1, argc);
        while (*argv)
               printf("%s\n", *argv++);
        while (*envp)
               printf("%s\n", *envp++);

        system("pause");
        return 0;
}
```

Функция int main(), как и другие функции, может возвращать целое значение. Оно передается в вызывающий процесс. Возврат нулевого значения говорит об успешном окончании работы, другие значения говорят об аварийном окончании работы.

Препроцессор и его директивы

Препроцессор – это процессор, манипулирующий исходным текстом Си программы на первой стадии ее компиляции.
Управление работой препроцессора производится посредством директив препроцессора. Обработанный препроцессором исходный текст поступает компилятору и он уже не содержит директив препроцессора. Директивы препроцессора отмечаются специальным маркером – знаком номера (#). Между знаком номера и первой буквой директивы могут находиться пробелы.

Директивы препроцессора могут появиться в любом месте исходного файла, но они применимы только к остатку исходного файла. Директивы дают команду препроцессору заменить тот или иной идентификатор некоторым значением, выполнить операцию макроопределения, включить текст некоторого файла в данный файл, “подавить” компиляцию части программы посредством условной компиляции.

Директива **#define**
Эта директива имеет два формата:

1. #define идентификатор текст_подстановки
2. #define идентификатор(список параметров) текст_подстановки

Первый формат определяет простую подстановку. То есть, в тексте исходного файла, следующем за директивой #define, все вхождения идентификатора заменяются на текст подстановки, определенный в директиве справа от идентификатора.

Текст подстановки может быть опущен. В этом случае удаляются все вхождения идентификатора из текста исходной программы.

Символические константы можно определять в любом месте исходного кода. Однако чтобы переопределить их (изменить значение), необходимо отменить предыдущее определение. Для удаления символической константы используют директиву #undef:

Пример:

```
#define PI 3.1415926
```

Пример:

```
#define n 10
 …
#undef n
…
#define n12
```

Второй формат позволяет определить макрокоманды с аргументами. Если после идентификатора следует список параметров, то в исходном тексте, следующем за директивой #define, каждое вхождение выражения идентификатор (список параметров) заменяется на текст подстановки с заменой формальных параметров фактическими.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#define MIN(x,y) ((x)<(y)?(x):(y))    /* минимум из x и y */

int main()
{
   float a = -15, b = 10;
   printf("min=%f\n", MIN(a, b));
   system("pause");
   return 0;
}
```

Замечание:

- параметры макроопределения заключают в круглые скобки;
- макроопределение не заканчивается точкой с запятой.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#define v(x) (x*x*x)

int main() {
       float a = 2;
       printf("V=%f\n", v(a+3));
       system("pause");
       return 0;
}
```

Директива **#include**

Эта директива имеет формат:

1. #include<спецификатор маршрута>
2. #include “спецификатор маршрута”

Спецификатор маршрута – это имя файла с предшествующей спецификацией каталога. Директива #include добавляет содержимое заданного файла включения к другому файлу. Включаемые файлы также могут содержать директивы препроцессора. Препроцессор выполняет директивы из нового файла, а затем продолжает обработку текста исходного файла.

Если имя файла заключено в скобки, то файл берется из некоторого стандартного каталога, а текущий каталог не просматривается. Если имя файла дано в кавычках, то препроцессор ищет файл в текущем каталоге. Если его там нет, то ищется в стандартном каталоге.

Директивы условной компиляции

Это директивы **#if, #elif, #else, #endif**

Эти директивы позволяют отменить компиляцию отдельных частей исходного файла посредством проверки константных выражений или идентификаторов, и имеют следующий формат:

```
#if константное выражение_1
        [текст]
[#elif константное выражение_2
        текст]
[#elif константное выражение_3
        текст]
        …
[#else
        текст]
#endif
```

Каждой директиве #if в исходном файле должна соответствовать закрывающая директива #endif. Между ними может быть несколько директив #elif и не более одной #else.
Директива #else, если она есть, должна быть последней директивой перед директивой #endif.
Препроцессор выбирает одно из вхождений текста, на основе вычисления константного выражения. Выбирается то, где будет найдено выражение со значением истина (не ноль). Если все выражения ложны

или отсутствует директива #elif, то текст берется после директивы #else. Если эта директива отсутствует, то никакой текст не выбирается.

Константное выражение – это выражение из различных констант, исключая константы типа enum (перечисление).

Константы, определенные препроцессором
Препроцессор самостоятельно определяет пять констант. От обычных (определенных программистом) они отличаются наличием пары символов подчеркивания в начале и конце их имени.
__DATE__ - дата компиляции;
__FILE__ - имя компилируемого файла;
__LINE__ - номер текущей строки исходного текста программы;
__STDC__ - равна 1, если компилятор работает по стандарту ANSI для языка C;
__TIME__ - время компиляции.
Если эти константы встречаются в тексте программы, то заменяются на соответствующие строки или числа.

```c
#include <stdlib.h>
#include <stdio.h>

#define PR printf("\n")

int main() {
	printf(__DATE__); PR;
	printf("%d", __LINE__); PR;
	printf(__FILE__); PR;
	printf(__TIME__); PR;

	system("pause");
	return 0;
}
```

Урок от 19.11.18

Файлы данных и функции ввода – вывода.

Под файлом далее будем понимать поименованную часть памяти. В функциях, работающих с файлами, язык Си “рассматривает” файл как структуру. Эта структура описана в файле stdio.h. Наименование шаблона файла – FILE.

Поток – это файл данных или физическое устройство (принтер, дисплей и т.п.), с которым программист работает с помощью указателя на объект типа FILE. Так, функция, работающая с потоком данных, возвращает в качестве значения, значение указателя на объект типа FILE.

Открытие файла**: fopen_s()**
Фрагмент программы:

```
FILE *in;
fopen_s(&in,“test.dat”,”r”);
```

Функцией fopen() управляют три основных параметра:

1. Имя файла, который следует открыть (test.dat).
2. Описание того, как должен использоваться файл
   - “r” – чтение существующего файла;
   - “w’ – запись;
   - “a” – добавление в конец файла (если файла нет, он создается);
   - “r+” – чтение и запись в файл (файл должен существовать);
   - “w+” – чтение и запись в файл (если файл уже существует, то его содержимое уничтожается);
   - “a+” – чтение и добавление в файл.

   Кроме этого различают еще текстовый (t) и двоичный (b) потоки. Если при открытии файла не указан тип потока, то по умолчанию файл обрабатывается системой как текстовый файл.
3. Указатель на файл (in). Его значение возвращается функцией и с этого момента программа ссылается на файл при помощи указателя, а не по имени файла.

Закрытие файла: **fclose()**
Пример:

```
fclose(in);
```

Аргументом функции является указатель на файл, а не имя файла. Функция возвращает 0, если файл закрыт успешно, и -1 в противном случае.

Ввод и вывод символа из файла: **getc(), putc()**
Пример:

```
ch=getc(in);        /* ввод символа */
putc(ch,out);       /* вывод символа */
```

Пример:/* Иллюстрация работы функций fopen(), fclose() */

/* Чтение из файла. Имя файла записывается в тело программы */

```
#include <stdio.h>
#include<windows.h>
#include <stdlib.h>
#include <string.h>


int main()
{
	char cStr1[] = "Необходимо имя файла \n";
	char cStr2[] = "Невозможно открыть файл";
	FILE *in, *out;
	int ch;
	char name[20];
	int count = 0;
```

```
	CharToOem(cStr1, cStr1);
	CharToOem(cStr2, cStr2);


	if (fopen_s(&in, "c:\\Test.txt", "r") == 0)
	{
		strcpy_s(name, 20, "c:\\Programm\\Out");
		printf("%s\n", name);
		strcat_s(name, 20, ".txt");
		printf("%s\n", name);
		fopen_s(&out, name, "w+");

		while ((ch = fgetc(in)) != EOF)
			if (count++ % 3 == 0)
				putc(ch, out);

		fclose(in);
		fclose(out);
	}
	else
		printf("Error!!!\n");


	system("pause");
	return 0;
}
```

/* Чтение из файла. Имя файла записывается в командной строке */

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
	char cStr1[] = "Необходимо имя файла \n";
	char cStr2[] = "Невозможно открыть файл";
	FILE *in, *out;
	int ch;
	char name[20];
	int count = 0;
	CharToOem(cStr1, cStr1);
	CharToOem(cStr2, cStr2);
	if (argc < 2)
		printf("%sn", cStr1);
	else
	{
		if ((fopen_s(&in, argv[1], "r")) == 0)
		{
			strcpy_s(name, 20, "c:\\Programm\\res.txt");
			fopen_s(&out, name, "w+");
			while ((ch = getc(in)) != EOF)
				if (count++ % 3 == 0)
				{
					putc(ch, out);
				}
		fclose(in);
		fclose(out);
		}
		else
			printf("%s %s\n", cStr2, argv[1]);
	}
	system("pause");
	return 0;
```

}

Замечание: Необходимо иметь текстовый файл, имя которого задается при запуске программы на выполнение. Содержимое файла в примере: “Даже Эдди нас опередил с детским хором”. В результате выполнения программы появляется новый файл с текстом …..

Функции ввода и вывода данных из файла: **fscanf_s(), fprintf().** Эти две функции работают также как и функции printf(), scanf(), но не со стандартными устройствами ввода и вывода, а с файлами. При работе с этими функциями необходим еще один дополнительный аргумент – указатель на файл.

```
FILE *in, *out;

if ((fopen_s(&in,"c:\\Programm\\dat.txt", "r")) == 0)
{
        fscanf_s(in, "%d", &n);
        …

        fprintf(out, "%6.2f ", x[i]);
```

Работа с динамическими массивами

В Си реализовано статитическое выделение памяти: объем необходимой памяти известен на этапе компиляции.
Можно организовать динамическое выделение памяти: выделение памяти во время выполнения программы.
Прототипы функций, управляющих выделением памяти, находятся в файле ***stdlib.h***

1. Функции выделения памяти.
2. Функции изменения объема выджеляемой памяти.
3. Функции освобождения памяти.

Функции выделения памяти:

1. calloc() – динамически выделяет блок памяти и инициализирует его нулями.
2. malloc() - динамически выделяет блок памяти без инициализации его содержимого.

Формат функций:

1. calloc(size_t n, size_t s), где n – количество элементов, под хранениекоторых выделяется память; s - размер пямяти в байтах для хранения одного элемента.
2. malloc(size_t s), где s – размер выделяемой памяти в байтах.

Пример:

```
int *p;
p = calloc(10, sizeof(int));

int *p;
p = malloc(10* sizeof(int));
```

Выделение памяти с инициализацией :

```
int * p=(int*) calloc(10, sizeof());
```

Функция перераспределения памяти:

realloc(void *p, size_t s) , где p – указатель на блоквыделенной памяти, s – новый размер блока выделяемой памяти.

Функция освобождения памяти:

```
free(void * P);
```

Выделение памяти по массив
Пример: /* Выделение памяти и ввод данных в двумерный массив x(m, n) * /

```
int ** x;
```

```
x= (int **) calloc(n, sizeof(int))
{
        for(i=0; i<n; i++)
        x[i]=(int *) calloc(m, sizeof(int));
        for(j=0; j<m; j++)
        {
          scanf_s("%d",&x[i][j]);
        }
}
```

Задание: Написать программу:

/* Нахождение выборочного мат. ожидания, стандартного отклонения, коэффициента корреляции, значений однофакторной линейной регрессии. Данные хранятся в файле dat.txt, результаты выполнения программы записываются в файл dat1.txt */

$$M(x)=\frac{1}{n}\sum_{i=1}^{n}x_i,\quad M(y)=\frac{1}{n}\sum_{i=1}^{n}y_i.$$

$$S(x)=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-M(x))^2},\quad S(y)=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-M(y))^2}.$$

$$R(x,y)=\frac{\sum_{i=1}^{n}(x_i-M(x))(y_i-M(y))}{n\cdot S(x)\cdot S(y)}.$$

$$yr=M(y)+\frac{\mathrm{S(y)}}{\mathrm{S(x)}}\cdot R\cdot(x-M(x)).$$

<u>Интерполирование по методу наименьших квадратов</u>

Пусть мы ищем приближающий полином первой степени: $y=a_0+a_1x.$

Рассмотрим функционал, который надо минимизировать: $I=\sum_{i=1}^{n}\left(y_i-f(x_i)\right)^2\to\min$, где

$f(x)=a_0+a_1x$.

Таким образом, надо найти два коэффициента a_0,a_1 так, чтобы минимизировать функционал I.

Для этого найдем частные производные по параметрам и приравняем их к нулю:

$$\begin{cases}\frac{\partial I}{\partial a_0}=0\\ \frac{\partial I}{\partial a_1}=0\end{cases}\Rightarrow\begin{cases}\frac{\partial I}{\partial a_0}=2\cdot\sum_{i=1}^{n}\left(y_i-a_0-a_1x_i\right)\cdot(-1)=0\\ \frac{\partial I}{\partial a_0}=2\cdot\sum_{i=1}^{n}\left(y_i-a_0-a_1x_i\right)\cdot(-x_i)=0\end{cases}\Rightarrow$$

$$\begin{cases}na_0+\sum_{i=1}^{n}a_1x_i=\sum_{i=1}^{n}y_i\\ \sum_{i=1}^{n}a_0x_i+\sum_{i=1}^{n}a_1x_i^2=\sum_{i=1}^{n}x_iy_i\end{cases}\Rightarrow\begin{cases}a_0+a_1\overline{x}=\overline{y}\\ a_0\overline{x}+a_1(\overline{x^2})=\overline{xy}\end{cases},$$

где $\overline{x}=\frac{1}{n}\sum_{i=1}^{n}x_i,\quad\overline{y}=\frac{1}{n}\sum_{i=1}^{n}y_i,\quad\overline{x^2}=\frac{1}{n}\sum_{i=1}^{n}x_i^2,\quad\overline{xy}=\frac{1}{n}\sum_{i=1}^{n}x_iy_i$.

Таким образом, имеем два уравнения и два неизвестных, и неизвестные входят в уравнения линейно. То есть, имеем СЛАУ (систему линейных алгебраических уравнений) 2-го порядка.
Запишем в матричном виде:

$$A \cdot a = b, \text{ где } A = \begin{pmatrix} 1 & \overline{x} \\ \overline{x} & \overline{x^2} \end{pmatrix}, \quad a = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, \quad b = \begin{pmatrix} \overline{y} \\ \overline{xy} \end{pmatrix}.$$

Домножая слева левую и правую части на A^{-1}, получим решение $a = A^{-1}b$, то есть вектор коэффициентов, минимизирующий рассматриваемый функционал в смысле метода наименьших квадратов:
$a = A^{-1}b$

Пусть x – рост, y – вес, представленные выборочными значениями. Пусть в файле dat.txt хранятся выборки объемом n=10 для роста и веса соответственно:

155 170 175 182 177 183 169 180 177 168
59 74 80 87 83 90 75 94 85 77 .

Тогда реультаты выполнения программы (файл dat1.txt):

M1=173.60 M2= 80.40 Sr1= 7.98 Sr2= 9.43 R= 0.96

x:155.00 170.00 175.00 182.00 177.00 183.00 169.00 180.00 177.00 168.00
y: 59.00 74.00 80.00 87.00 83.00 90.00 75.00 94.00 85.00 77.00
yr: 59.31 76.32 81.99 89.92 84.26 91.06 75.18 87.66 84.26 74.05

Урок от 26.11.18

Стандартные функции

Функции обработки строк в Си

Для обработки строк в Си определено много библиотечных функций. Строковые функции объявлены в заголовочном файле <string.h>
Наиболее часто используются следующие функции:

Имя функции	Выполняемое действие
strcpy(s1,s2)	Копирование s2 в s1
strcat(s1,s2)	Конкатенация (объединение) 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, положительное значение, 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

Замечание: Сравнение в функции strcmp() происходит в лексикографическом порядке.
Пример:

```
#include <stdio.h>
#include<string.h>
int main()
{
char s1[]="Password";
char s2[]="pass";
char s3[]="pasport";
char s4[]="Password";
printf("%d\t%d\t%d\n", strcmp(s1,s2),strcmp(s2,s3),strcmp(s1,s4));
return 0;
}
//Результат выполнения программы: -1		1		0
```

Основные функции ввода – вывода

Функция	Выполняемое действие
getchar()	Читает символ с клавиатуры. Ожидает возврат каретки.
getche()	Читает символ, при этом он отображается на экране. Не ожидает возврата каретки. В стандарте Си не определена.
getch()	Читает символ, но не отображает его на экране. Не ожидает возврата каретки. В стандарте Си не определена.
putchar()	Отображает символ на экране.
gets()	Читает строку символов, введенную с клавиатуры, и записывает по адресу, на который указывает аргумент. Ввод осуществляется, пока не введен <enter>.
puts()	Отображает строку символов на экране.

Основные функции файловой системы Си

Имя функции	Назначение
fopen()	Открывает файл
fclose()	Закрывает файл
putc()	Записывает символ в файл
fputc()	Записывает символ в файл (для совместимости)
getc()	Читает символ из файла
fgetc()	Читает символ из файла (для совместимости)
fgets()	Читает строку из файла
fputs()	Записывает строку в файл
fseek()	Устанавливает файловый указатель на определенный байт файла
ftell()	Возвращает текущее значение указателя
fprintf()	Вывод в файл

fscanf()	Ввод из файла
feof()	Возвращает значение истина, если достигнут конец файла
ferror()	Возвращает значение истина, если произошла ошибка
rewind()	Устанавливает указатель текущей позиции в начало файла
remove()	Стирает файл
fflush()	Дозапись потока в файл

Изменение программы: // стр. 49

```
#include<stdio.h>
int (*fun)(int,int), sum(int,int), razn(int,int);
void main()
{
        int a=10, b=20, c;
        c=sum(a,b);
        printf("c=%d\tsum=%d\trazn=%d\n", c, sum(a,b), razn(a,b));
        fun=sum;
        printf("*fun=%d\n",(*fun)(a,b));
        fun=razn;
        printf("*fun=%d\n",(*fun)(a,b));
 }
int sum(int x,int y)
{
return (x+y);
}
int razn(int x,int y)
{
return (y-x);
}
```

Приложение: /* *Стандартная библиотека Си* */

Содержимое и форма стандартной библиотеки Си задается стандартом ANSI/ISO.

Заголовочные файлы

Список заголовочных файлов, определенных в стандарте C89

Заголовок	Назначение
<assert.h>	Определяет макрос assert()
<ctype.h>	Обработка символов
<errno.h>	Выдача сообщений об ошибках
<float.h>	Задает пределы значений с плавающей точкой
<limits.h>	Задает различные ограничения
<locate.h>	Поддерживает локализацию
<math.h>	Определения, используемые математической библиотекой
<setjmp.h>	Поддерживает нелокальные переходы
<signal.h>	Поддерживает обработку сигналов
<stdarg.h>	Списки входных параметров функции с переменным числом аргументов
<stddef.h>	Определяет наиболее часто используемые константы
<stdio.h>	Поддерживает систему ввода/вывода
<stdlib.h>	Смешанные объявления
<string.h>	Функции обработки строк
<time.h>	Функции, обращающиеся к системному времени

Список заголовочных файлов, добавленных в C99

Заголовок	Назначение
<comlex.h>	Арифметические операции с комплексными числами
<fenv.h>	Доступ к флажкам состояния вычислителя
<inttypes.h>	Стандартный, переносимый набор имен целочисленных типов
<iso646.h>	Макросы, соответствующие различным операторам, например && и ^
<stdbool.h>	Логические типы данных, определяет макрос bool
<tgmath.h>	Определяет макросы для родового (абстрактного) типа чисел с плавающей точкой
<wchar.h>	Поддерживает функции обработки ногобайтовых слов и двухбайтовых символов
<wctype.h>	Поддерживает функции классификации многобайтовых слов и двухбайтовых символов

Математические функции

Стандарт С89 определяет 22 математические функции. Для использования математических функций необходимо включить заголовок <math.h>.
В версии С89 аргументы математических функций и значения, возвращаемые функциями, должны быть значения типа double.

Математические функции стандарта С89

Функция	Назначение
acos	Возвращает значения арккосинуса
asin	Возвращает значения арксинус
atan	Возвращает значение арктангеса
atan2	Возвращает значение арктангеса отношения a/b
ceil	Возвращает наименьшее целое, которое больше значения аргумента или равно ему
cos	Возвращает значение косинуса (значение аргумента должно быть в радианах)
cosh	Возвращает значение гиперболического косинуса
exp	Возвращает значение экспоненты от аргумента
fabs	Возвращает абсолютное значение аргумента
floor	Возвращает наибольшее целое, которое меньше значения аргумента или равно ему
fmod	Возвращает остаток от деления аргументов a/b
frexp	Разбивает число на мантиссу и целый показатель степени числа 2
ldexp	Возвращает значение выражения $num * 2^{exp}$
log	Возвращает значение натурального логарифма
log10	Возвращает значение логарифма по основанию 10
modf	Разбивает аргумент на целую и дробную части
pow	Возвращает значение аргумента возведенного в степень
sin	Возвращает значение синуса
sinh	Возвращает значение гиперболического синуса
sqrt	Возвращает значение квадратного корня
tan	Возвращает значение тангеса
tanh	Возвращает значение гиперболического тангеса

Служебные функции

В библиотеке стандартных функций определен ряд служебных функций. Они описаны в заголовочном файле <stdlib.h>. Рассмотри некоторые из них:

Функция	Назначение
abort	Вызывает немедленное аварийное завершение программы
	Возвращает абсолютное значение целочисленного аргумента
atof	Преобразует символьную строку в значение типа double
atoi	Преобразует символьную строку в значение типа int
atol	Преобразует символьную строку в значение типа long int
bsearch	Выполняет двоичный поиск в отсортированном массиве
div	Возвращает частное и остаток, полученные в результате деления
exit	Вызывает немедленное нормальное завершение программы
qsort	Сортирует массив (алгоритм быстрой сортировки Хоара)
rand	Генерирует последовательность псевдослучайных чисел
crand	Устанавливает исходное число для последовательности, генерируемой функцией rand()
strtod	Преобразует строковое представление числа, которое содержится в строке, в значение типа double

Функции динамического распределения памяти

Прототипы функций динамического распределения памяти находятся в <stdlib.h>. Область свободной памяти, в которой распределяется память, называется динамически распределяемой областью памяти или *кучей* (*heap*).

Функция	Назначение
calloc	Выделяет память, достаточную для размещения массива. Все биты инициализируются нулями
free	Возвращает обратно динамически выделенную память
malloc	Возвращает указатель на первый байт выделяемой памяти
realloc	Изменяет размер блока ранее выделенной памяти

Функции времени, даты и локализации

Для использования функций времени и даты необходим заголовочный файл <time.h>. Этот файл определяет три типа данных, связанных с исчислением времени: clock_t, time_t и tm. Типы данных clock_t, time_t предназначены для представления системного времени и даты в виде некоторого целого значения. Структурный тип tm содержит дату и время, разбитые на составляющие компоненты. В этом заголовочном файле также определен макрос CLOCKS_PER_SEC, который содержит число тактов системных часов в секунду.

Рассмотрим некоторые из функций времени и даты:

Функция	**Назначение**
asctime	Возвращает указатель на строку, которая имеет форму: день_недели месяц число часы:минуты:секунды год\n\0
clock	возвращает значение, которое возвращает время работы вызывающей пограммы
ctime	возвращает указатель на строку, имющую вид: день месяц год часы:минуты:секунды
difftime	возвращает разность в секундах между значениями параметров time1 и time2 (time2-time1)
strtime	помещает информацию о времени и дате в строку, в соответствии с командами форматирования
time	возвращает текущее клендарное время суток

Задание

// Программа "***Шифр Цезаря***"

Открытый текст – сообщение, которое подлежит шифрованию.

Результат применения к открытому тексту алгоритма шифрования называют шифротекстом.

Шифр Цезаря – каждая буква открытого текста заменяется третьей после нее буквой в алфавите, который считается записанным по кругу.

Пусть x – номер буквы открытого текста в алфавите, c – элемент шифротекста.

Шифрование – x ->c=x+3 (mod 67)

Дешифрование производится обратным сдвигом: x=c-3 (mod M), что эквивалентно x=c+(M-K) (mod M) M=67. K=3.

Урок от 2.12.18

Функции ввода / вывода (дополнение)

С функциями ввода/вывода ассоциирован заголовок `<stdio.h>`. Этот заголовок определяет некоторые макросы и типы, которые используются файловой системой. Наиболее важным из них является тип `FILE`, который используется для объявления указателя на файл. Два других часто используемых типа — `size_t` и `fpos_t`. Тип `size_t`, представляющий собой некоторую разновидность целых без знака, — это тип результата, возвращаемого функцией `sizeof`. Тип `fpos_t` определяет объект, который однозначно задает каждую позицию в файле. Самым популярным макросом, определенным в этом заголовке, является макрос `EOF`, значение которого указывает на конец файла. Другие типы данных и макросы, определенные в заголовке `<stdio.h>`, описаны вместе с функциями, с которыми они связаны.

Многие функции ввода/вывода при возникновении ошибки присваивают встроенной глобальной переменной целого типа `errno` определенное значение. Анализ этой переменной поможет программе получить более подробную информацию о возникшей ошибке. Значения, которые может принимать переменная `errno`, зависят от конкретной реализации компилятора.

fseek– установка позиции в потоке данных.

Синтаксис:

```
#include < stdio.h >
int fseek(FILE *stream, long int offset, int whence);
```

Аргументы:

stream – указатель на управляющую таблицу потока данных.
offset – смещение позиции.
whence – точка отсчета смещения.

Возвращаемое значение:

0 – при успешной установки позиции.

Отличное от нуля значение, если при работе функции произошли ошибки. При этом переменной ernno будет присвоен код ошибки:

[EINVAL] – неверное значение аргумента whence
[ESPIPE] – недопустимое значение аргумента offset

Описание:

Функция fseek() устанавливает позицию в потоке данных, заданным аргументом stream. Относительно установленной позиции будет осуществляться чтение и запись данных.

Точка отсчета устанавливаемой позиции определяется аргументом whence, который может принимать значения:

SEEK_SET – смещение отсчитывается от начала файла
SEEK_CUR – смещение отсчитывается от текущей позиции
SEEK_END – смещение отсчитывается от конца файла

Код	Положение в файле
0	начало файла
1	текущая позиция
2	конец файла

Смещение задается аргументом offset, причем положительное значение аргумента означает смещения вправо от указанной аргументом whence позиции, отрицательное – смещение влево.

Для двоичных потоков данных, смещение (offset) – это количество байт.

Для текстовых потоков данных смещение (offset) должно быть равно 0, либо получено с помощью функции ftell(), при этом точка отсчета (whence) должна иметь значение SEEK_SET.

```
/* Чтение и запись в файл с функцией  feof()*/

#include <stdio.h >
#include <windows.h>
#include <stdlib.h>
#include<locale.h>

int main()
{
	FILE *in, *out;
	long num = 10L;
	/* #include<locale.h> */
	setlocale(LC_ALL, "rus");
//	setlocale(0, "");
	/*#include <windows.h>*/
//	SetConsoleOutputCP(1251);

	printf("Читаем на русском языке!\n");

		if (fopen_s(&in,"c:\\Programm\\test.txt", "r") != 0)
		printf("Не открыть файл \n");

		else
		{
			fopen_s(&out,"c:\\Programm\\data1.txt", "w");
			fseek(in, num, 0);
			while (!feof(in))
			{
				fprintf(out, "%c", getc(in));
			}

			fclose(in);
			fclose(out);
			}
		system("pause");
```

```
        return 0;
}
```

Метод Монте-Карло

/* Приближенное вычисление числа пи */

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<locale.h>
#define P 3.14159265

int main()
{
	struct coord {
		double x;
		double y;
	} z;
	long i, j, n, n1 = 1e+8;
	double S, Sr = 0;
	srand((unsigned)time(NULL));
	setlocale(LC_ALL, "rus");
	printf("Введите число циклов моделирования : \n");
	scanf_s("%d", &n);
	time_t beg_t, end_t;
	beg_t = time(NULL);
	for (j = 0; j < n; j++) {
		S = 0;
		for (i = 0; i < n1; i++)
		{
			z.x = (double)rand() / (double)RAND_MAX;
			z.y = (double)rand() / (double)RAND_MAX;
			z.x -= 0.5;
			z.y -= 0.5;
			if ((z.x*z.x + z.y*z.y) <= 0.25)
				S++;
		}
		S /= (double)n1;
		printf("S=%10.8f\n", S);
		Sr += S;
	}
	end_t = time(NULL);
	printf("T=%f sec\n", difftime(end_t, beg_t));
	printf("Int=%10.8f\tPi=%10.8f\n", Sr / (double)n, 4 * Sr / (double)n);

	system("pause");
	return 0;
}
```

Раздел II. Технология программирования (C++)

В этом разделе рассматриваются элементы технологии объектно-ориентированного программирования. В качестве базового языка используется язык C++, который рассматривается как некоторая надстройка над языком Си. То есть, язык C++ интерпретируется как расширение языка Си с поддержкой ООП (Object Oriented Programming, OOP).

Основные понятия объектно-ориентированного программирования

ООП – это новая концепция в технологии программирования по созданию сложных программных комплексов.
ООП основано на трех основополагающих концепциях:
инкапсуляция;
полиморфизм;
наследование.
Замечание: Иногда сюда добавляют еще одну концепцию – концепцию абстракции типов.

Инкапсуляция (encapsulation) – это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то и другое от внешнего вмешательства или неправильного использования.

Когда данные и коды объединены, то говорят, что создается объект, то есть иными словами, объект – это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть закрытыми (private), открытыми (public) или защищенными (protected). Закрытые коды или данные доступные только для других частей этого объекта. Если коды и данные являются открытыми, то они доступны и для других частей программы. Использование защищенных данных и кодов занимает промежуточный вариант между использованием открытых и закрытых (см. далее).

Полиморфизм (polymorphism) (от греческого слова polymorphos) – это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.
Выполнение действий в каждом конкретном случае будет определяться типом данных. Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. То есть, в C++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функции (function overloading).

В более общем смысле концепцией полиморфизма является идея "один интерфейс, множество методов". То есть полиморфизм позволяет манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.
Элементы полиморфизма присутствовали в других языках и раньше (знак "+").

Наследование (inheritance) – это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним свойства, характерные только для него.

Наследование позволяет поддерживать концепцию иерархии классов.

Пример; // полиморфизм
В языке Си нахождение абсолютной величины числа требует трех различных функций
abs() – для целых чисел;
labs() – для длинных чисел;
fabs() – для чисел с плавающей точкой.
В C++ используется одна функция abs().
Полиморфизм в жизни – колесо (колесо, рулевой механизм).

Пример; // наследование (иерархия классов)
комната -> дом-> строение -> конструкция.

Структура программы C++

В настоящее время глобально существует две версии C++:
Традиционная (с начала 90-х годов двадцатого века);
Стандартная (современная,усовершенствованная).

Примеры отличий:
Изменился стиль оформления заголовков (headers).
Появилась инструкция namespace.

```
// Программа на C++ в традиционном стиле:
#include <iostream.h>
int main()
{

}

// Программа на C++ в современном стиле:
#include <iostream>
using namespace std;
int main()
{

}
```

iostream.h в C++ имеет то же самое назначение, что stdio.h в Си.
Заголовки уже не обязательно являются именами файлов, поэтому не надо указывать расширение .h, а только имя заголовка в угловых скобках.

В стандартном C++ к стандартным заголовкам Си просто добавляется префикс с и удаляется расширение .h

Пример:

в Си	в C++
math.h	cmath
string.h	cstring

Пространство имен

Эта некая объявляемая область, необходимая для того, чтобы избежать конфликтов имен идентификаторов. Традиционно имена библиотечных функций располагались в глобальном пространстве имен (язык Си). Однако содержание заголовков нового стиля помещается в пространстве имен std.
Для того чтобы пространство имен была видимым, используют инструкцию:
using namespace std;
Это инструкция помещает std в глобальное пространство имен.

Комментарии в C++

/* */ многострочный комментарий;
// – однострочный комментарий.

Консольный ввод и вывод в C++

Хотя функции printf() и scanf() доступны в C++, для ввода и вывода используют операторы, а не функции ввода-вывода.
Оператор ввода: cin>>
Оператор вывода: cout<<
Конец строки: endl
В Си это операторы левого и правого сдвига. В C++ они сохраняют свое первоначальное значение, выполняя при этом еще ввод и вывод.

Пример:

```
cout<<"Это строка вывода на экран.\n";
cout<<10.99;
```

т.е., в общем виде

```
cout<<выражение;
```

```
int num;
cin>>num; // & ставить не надо.
```

Операторы << >> – являются примером перегрузки операторов.

```
//Первая программа на C++

#include<iostream>
#include<cmath>
using namespace std;

int main()
{
	printf("Hello C++\n");
	int a, b = 10;
	double c;
	scanf_s("%lf", &c);
	c = sqrt(c);
	cout << "a=";
	cin >> a;
	cout << "b=" << b << " a=" << a <<" c=" << c << endl;
	system("pause");
	return 0;
}
```

Замечание:

1. При стандартных настройках вариант <iostream.h> не работает.
2. Возможно использование функций языка Си(printf(), scanf()).

Два двоеточия называются оператором расширения области видимости.

```
//Первая программа на C++ (без использования using namespace std;)


#include<iostream>
#include<cmath>

int main()
{
	printf("Hello C++\n");
	int a, b = 10;
	double c;
	scanf_s("%lf", &c);
	c = sqrt(c);
	std::cout << "a=";
	std::cin >> a;
	std::cout << "b=" << b << " a=" << a <<" c=" << c << std::endl;
	system("pause");
	return 0;
}
```

Классы

Одним из наиболее важных понятий C++ является понятии класса.
Класс – это механизм для создания объектов.
Класс объявляется с помощью ключевого слова class
Синтаксис объявления класса похож на синтаксис объявления структуры.

```
class имя_класса {
                закрытые функции и переменные класса
                public:
                открытые функции и переменные класса
} список_объектов;
```

В объявлении класса список_объектов не обязателен, можно объявлять объекты класса позже, по мере необходимости.
Функции и переменные, объявленные внутри объявления класса, становятся членами (members) этого класса.

По умолчанию все переменные и функции, объявленные в классе, становятся закрытыми. То есть, они доступны только для других членов того же класса.

Для объявления открытых членов класса используется ключевое слово public, за которым следует двоеточие. Все переменные и функции, объявленные после public, доступны как для других членов класса, так и для данных любой другой части программы, в которой находится этот класс.
Отличием между структурой и классом в C++ является то, что члены класса по умолчанию, являются закрытыми, а члены структуры – открытыми.

Пример:

```
class myclass  {
             // закрытые члены класса
                 double a;
               public:
                 void set_a(double num);
                 double get_a();
           };
```

В этом примере переменная a – закрытая переменная; set_a() и get_a() – две открытые функции. Прототипы функций объявляются внутри класса (функции – члены класса).
После объявления set_a() и get_a() в myclass, они еще должны быть определены. Для определения функции-члена класса надо связать имя класс, частью которого является функция-член, с именем функции.

Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Два двоеточия называются оператором расширения области видимости.

```
void myclass::set_a(double num)
      {
        a=num;
}
double myclass::get_a()
      {
        return a;
       }
```

Основная форма определения функции-члена:

```
тип_возвр_значения имя_класса::имя_функции(список параметров)
      {
            … // тело функции
      }
```

Создание объектов типа myclass:

```
myclass ob1; // объект типа myclass
```

Обращение к членам класса производится через оператор точка (.):

```
ob1.set_a(99.9);
```

Обращение к закрытой переменной a, только через открытую функцию.

```cpp
// Вторая программа на C++

#include<iostream>
using namespace std;

class myclass {
	// закрытые члены класса
	double a;
public:
	void set_a(double num);
	double get_a();
};

void myclass::set_a(double num)
{
	a = num;
}
double myclass::get_a()
{
	return a;
}


int main()
{
	double b;
	myclass ob1; // объект типа myclass

	// Обращение к членам класса производится через оператор точка(.) :

		ob1.set_a(99.9);
	// Ошибка ( a - закрытая переменная)
	//	cout << " a=" << ob1.a << endl;
		b = ob1.get_a();
		cout << "a=" << b << endl;
	system("pause");
	return 0;
}
```

Дополнение: краткая история и некоторые замечания по развитию языка C++

Язык C++ разработал сотрудник лабораторий Bell Бьерн Страуструп, который начал работать над ним в **1979** году. Первоначально язык назывался "Си с классами", так как в начале в Си были добавлены классы (с инкапсуляцией), производные классы, строгая проверка типов, inline-функции и аргументы по умолчанию.

В 1**983** году (Рик Маскитти) произошло переименование языка из "Си с классами" в C++. Кроме того, в него были добавлены новые возможности, такие как виртуальные функции, перегрузка функций и операторов, ссылки, константы, пользовательский контроль над управлением свободной памятью, улучшенная проверка типов и новый стиль комментариев (//).

В **1985** году вышло первое издание Бьерна Страуструпа "Язык программирования C++".

В **1989** году вышла версии 2.0. языка C++ . Его новые возможности включали множественное наследование, абстрактные классы, статические функции-члены, функции-константы и защищённые члены.

В **1998** году был ратифицирован международный стандарт языка C++: ISO/IEC 14882 "Standard for the C++ Programming Language".

Стандарт C++ 1998 года состоит из двух основных частей: ядра языка и стандартной библиотеки, которая включает STL (Стандартную Библиотеку Шаблонов) и версию стандартной библиотеки Си.

В **2003** году была выпущена новая версия стандарта языка C++ — ISO/IEC 14882:2003.

В настоящее время нет единого языка программирования C++, это название относится к семейству языков. Никто не обладает правами на язык C++, он является свободным.

Замечание: При написании программ надо учитывать, на каком компиляторе будет прогоняться программа. Большая часть кода Си будет справедлива и для C++, но, все-таки, C++ не является надмножеством Си и не включает его в себя. То есть, код, справедливый для обоих Си и C++, может давать разные результаты в зависимости от того, где он скомпилирован.

Пример: // отличия Си от C++

```
#include <stdio.h>
int main()
{
   printf("%s\n", (sizeof('a') == sizeof(char)) ? "C++" : "C");
   return 0;
```

}

Комментарий: программа печатает "C", если компилируется компилятором Си, и "C++" — если компилятором C++. Так происходит из-за того, что символьные константы в Си (в примере 'a') представлены целым типом int, а в C++ — типом char.
Далее будут рассмотрены и некоторые другие отличия языка Си от языка C++.

Visual C++ позволяет разрабатывать приложения в терминах традиционного процедурного, объектно-ориентированного, консольного и визуального программирования.
В базовом курсе будет использовать консольное приложение, то есть приложение, которое не использует Windows-окна для обработки сообщений от пользователя (в дополнение могут быть даны элементы визуального программирования). Точкой входа в консольное приложение в языке C++ является метод main().
Каждая программа должна иметь точку входа и может содержать описание одного или нескольких классов.

Недостатки ООП

1. Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Для абстрактных методов, которые пусты, в документации должно говориться о том, для каких целей предполагается использовать переопределяемый метод.
2. Реализация выполнения программы рассредоточивается по нескольким классам, и чтобы понять, как она работает, необходимо просматривать весь код.
3. Неоднозначность в использовании инкапсуляции данных.
4. Излишняя универсальность. В объектно-ориентированной программе обычно реализованы избыточные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо, но они не могут быть удалены.
5. Рост объёма памяти для выполнения программы и уменьшение скорости выполнения программы с использованием ООП.

Замечание: Надежность объекто-ориентированного программного обеспечения и быстрота его написания дают определяющие преимущества перед использованием процедурного программного обеспечения.

Классы (продолжение)

Перегрузка функций

В C++ две или более функции могут иметь одно и то же имя, отличаясь либо типом, либо числом своих аргументов, либо и тем и другим.

Если две или более функции имеют одинаковое имя, то говорят, что они перегружены.

Для перегрузки функции надо просто объявить и определить все требуемые варианты.

Компилятор автоматически выберет правильный вариант вызова на основании числа или типа используемых в функции аргументов.

Одно из основных применений перегрузки функций – это достижение полиморфизма (один интерфейс – множество методов).

Перегрузка функции. Определение адреса перегруженной функции

Также как и в Си можно присвоить адрес функции указателю и получить доступ к функции через указатель.

Адрес функции можно найти, если поместить имя функции в правой части инструкции присваивания без всяких скобок или аргументов.

Пример:// *Перегрузка функции. Определение адреса перегруженной функции*

```
#include <iostream>
using namespace std;

//Функция вычисления квадрата
void sq(float length)
{
	cout << "Square=" << length * length << endl;
}

//Функция вычисления прямоугольника
void sq(float length, float width)
{
	cout << "Rectangle=" << width * length << endl;
}

int main()
{
	sq(10);
	sq(3, 7);

	// Вызов функции через указатели

	void(*fp1)(float);
	void(*fp2)(float, float);

	fp1 = sq;	// получение адреса функции вычисления площади квадрата
	fp2 = sq;	// получение адреса функции вычисления площади
прямоугольника
	fp1(5);		// вызов функции вычисления площади квадрата
	fp2(4, 5);	// вызов функции вычисления площади прямоугольника
```

```
	system("pause");
	return(0);
}
```

Встраиваемые функции (inline)

В C++ существуют функции, которые не вызываются, а "встраиваются" в программу в место ее вызова (тело функции подставляется в каждую точку вызова).

Механизм "встраивания" функций подобен макросам языка Си, определенных с использованием директивы препроцессора #define. Однако между ними есть различия. Вызовы "встроенных" функций обрабатываются компилятором, а макросы обрабатываются препроцессором, который делает просто текстовую подстановку. При вызове "встраиваемых" функций компилятор проверяет типы аргументов.

Преимущество использования "встроенных" функций: нет вызова, поэтому выполняется быстрее.

Недостаток: если они большие, то занимают большой объем памяти.

Объявление "встраиваемых" функций: пишется спецификатор *inline* перед определением функции.

Пример: // *Использование встраиваемых функций*

```
#include<iostream>
using namespace std;

inline int choose(int x)
{
	return (x % 2);
}

int main()
{
	int addition = 0, multiplication = 1;
	for (int i = 0; i <= 5; i++)
		if (choose(i))
			multiplication *= i;
		else
			addition += i;
	cout << "addition=" << addition << endl;
	cout << "multiplication=" << multiplication << endl;
	system("pause");
	return 0;
}
```

// Встраиваемые функции в объявлении класса

```
#include<iostream>
using namespace std;
```

```
class sample_inline {
public:
	int choose(int);
};

inline int sample_inline::choose(int x)
{
	return (x % 2);
}
int main()
{
	int addition = 0, multiplication = 1;
	sample_inline ob;

	for (int i = 0; i <= 5; i++)
		if (ob.choose(i))
			multiplication *= i;
		else
			addition += i;
	cout << "addition=" << addition << endl;
	cout << "multiplication=" << multiplication << endl;

	system("pause");
	return 0;
}
```

// Компактный стиль встраиваемых функций в объявлении класса

```
#include<iostream>
using namespace std;

class sample_inline {
public:
	int choose(int x) { return (x % 2); }
};
int main()
{
	int addition = 0, multiplication = 1;
	sample_inline ob;

	for (int i = 0; i <= 5; i++)
		if (ob.choose(i))
			multiplication *= i;
		else
			addition += i;
	cout << "addition=" << addition << endl;
	cout << "multiplication=" << multiplication << endl;

	system("pause");
	return 0;
}
```

То есть, если встраиваемая функция задана внутри класса, то ключевое слово inline не требуется.

Замечание: спецификатор inline для компилятора является запросом, а не командой. Если компилятор не в состоянии выполнить запрос, функция будет компилироваться как обычная функция, а запрос inline будет проигнорирован. (В некоторых компиляторах не воспринимает циклы, goto, switch …).

Конструкторы и деструкторы

*Рассмотрим **вопросы инициализации** при работе с классами в C++.*
Для инициализации при работе с объектами в C++ имеется функция-конструктор (constructor function), включаемая в описание класса.

В классе может быть несколько конструкторов и только одна функция, обратная конструктору, называемая деструктором (destructor). Количество конструкторов определяется количеством имеющихся способов создания объекта. Конструктор выполняет инициализацию переменных-членов класса и резервирует динамическую память для объекта.

Конструктор класса вызывается всякий раз при создании объекта этого класса. Таким образом, необходимая объекту инициализация, при наличии конструктора, выполняется автоматически.
Конструктор имеет то же имя, что и класс, частью которого он является, и не имеет возвращаемого значения.
Функция – деструктор (деструктор), вызывается при удалении объекта и служит для освобождения памяти. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале.

Конструктор без параметров
Конструктор без параметров инициализирует переменные-члены, присваивая им устанавливаемые по умолчанию значения. Если конструктор не определен, компилятор генерирует его по умолчанию (*таким образом, конструктор существует всегда*).
Замечание: Если аргументы конструктору не передаются, то в определении объекта не следует включать пустые скобки (иначе будет определена функция, возвращающая тип класса).
Пример: // *Работа конструктора без параметров*

```
#include<iostream>
```

```
using namespace std;

class myclass {
	int a;
public:
	myclass();  // конструктор
	void show();
};
myclass::myclass()
{
	cout << "Пример инициализации в конструкторе \n";
	a = 25;
}
void myclass::show()
{
	cout << a << endl;
}
int main()
{
	setlocale(LC_ALL, "Russian");
	myclass ob;
	ob.show();

	system("pause");
	return 0;
}
```

Для глобальных объектов конструктор объекта вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении инструкции объявления переменной.

Пример:// фрагмент программы работы деструктора

```
~myclass(); // объявление деструктора в классе

myclass::~myclass() // определение деструктора вне класса
{
… // тело функции
}
```

Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы. Адреса конструктора и деструктора получить невозможно.

Замечание: Создать экземпляр класса можно при условии, что конструктор является функцией-членом класса типа *public*. Если класс предназначен лишь для создания других классов, конструктор можно сделать защищенным членом, поместив его в раздел *protected*.

Конструктор с параметрами

Конструктору можно передавать аргументы.

Пример:// *Работа конструктора с параметром*

```
#include<iostream>
using namespace std;

class myclass {
	int a;
public:
	myclass(int x);   // конструктор
	void show();
};
myclass::myclass(int x)
{
	cout << "Инициализация в конструкторе \n";
	a = x;
}
void myclass::show()
{
	cout << a << endl;
}
int main()
{
	setlocale(LC_ALL, "Russian");
	myclass ob(25);
	ob.show();
	system("pause");

	return 0;
}
```

Значение, передаваемое в myclass(), используется для инициализации параметра a.

Замечание: Выражение myclass ob(25); эквивалентно выражению

myclass ob = myclass(25);

Существует еще одна форма инициализации конструктора – инициализация расположена между прототипом функции и телом функции. От прототипа функции список инициализации отделен двоеточием, а инициализирующие значения помещены в скобках, после соответствующих переменных класса, через запятую. Инициализация переменных класса происходит до начала исполнения тела конструктора. Такой подход позволяет задавать, например, начальные значения констант и ссылок.

Пример:

```
#include<iostream>
using namespace std;
```

```
class myclass {
	int a, b;
	float c;
	const int d;
public:
	myclass(int x, int y, float z, int w) : a(x), b(y), c(z), d(w) { cout
<< "Инициализация в конструкторе \n"; }
	void show();
};
void myclass::show()
{
	cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d << endl;
}
int main()
{
	setlocale(LC_ALL, "Russian");
	myclass ob(25, 20, 15.5, 55);
	ob.show();
	system("pause");
	return 0;
}
```

Задание: Проверить невозможность инициализации константы в конструкторе другим способом (без списка инициализации).