Язык программирования СИ

Си (англ. C) — стандартизированный процедурный язык программирования, разработанный в начале 1970-х годов сотрудниками Bell Labs Кеном Томпсоном и Денисом Ритчи как развитие языка Би. Си был создан для использования в операционной системе UNIX. С тех пор он был портирован на многие другие операционные системы и стал одним из самых используемых языков программирования. Си ценят за его эффективность. Он является самым популярным языком для создания системного программного обеспечения. Его также часто используют для создания прикладных программ. Несмотря на то, что Си не разрабатывался для новичков, он активно используется для обучения программированию. В дальнейшем синтаксис языка Си стал основой для многих других языков.

Для языка Си характерны лаконичность, стандартный набор конструкций управления потоком выполнения, структур данных и обширный набор операций.

Обзор

Язык программирования Си отличается минимализмом. Авторы языка хотели, чтобы программы на нём легко компилировались с помощью однопроходного компилятора, чтобы каждой элементарной составляющей программы после компиляции соответствовало весьма небольшое число машинных команд, а использование базовых элементов языка не задействовало библиотеку времени выполнения. Однопроходный компилятор компилирует программу, не возвращаясь назад, к уже обработанному тексту. Поэтому использованию функции и переменных должно предшествовать их объявление. Код на Си можно легко писать на низком уровне абстракции, почти как на ассемблере. Иногда Си называют «универсальным ассемблером» или «ассемблером высокого уровня», что отражает различие языков ассемблера для разных платформ и единство стандарта Си, код которого может быть скомпилирован без изменений практически на любой модели компьютера. Си часто называют языком среднего уровня или даже низкого уровня, учитывая то, как близко он работает к реальным устройствам. Однако, в строгой классификации, он является языком высокого уровня.

Ранние разработки

- Язык программирования Си был разработан в лабораториях Bell Labs в период с 1969 по 1973 годы. Согласно Ритчи, самый активный период творчества пришёлся на 1972 год. Язык назвали «Си» (C — третья буква латинского алфавита), потому что многие его особенности берут начало от старого языка «Би» (B — вторая буква латинского алфавита).
- Самый первый компьютер, для которого была первоначально написана UNIX, предназначался для создания системы автоматического заполнения документов. Первая версия UNIX была написана на ассемблере. Позднее для того, чтобы переписать эту операционную систему, был разработан язык Си.

- К 1973 году язык Си стал достаточно силён, и большая часть ядра UNIX, первоначально написанная на ассемблере PDP-11/20, была переписана на Си. Это было одно из самых первых ядер операционных систем, написанное на языке, отличном от ассемблера

K&R C

- В 1978 году Ритчи и Керниган опубликовали первую редакцию книги «Язык программирования Си». Эта книга, известная среди программистов как «K&R», служила многие годы неформальной спецификацией языка. Версию языка Си, описанную в ней, часто называют «K&R C». (Вторая редакция этой книги посвящена более позднему стандарту ANSI C, описанному ниже.)

- K&R C часто считают самой главной частью языка, которую должен поддерживать компилятор Си. Многие годы даже после выхода ANSI C, он считался минимальным уровнем, которого следовало придерживаться программистам, желающим добиться от своих программ максимальной портативности, потому что не все компиляторы тогда поддерживали ANSI C, а хороший код на K&R C был верен и для ANSI C.

ISO C ANSI C

- В конце 1970-х годов Си начал вытеснять Бейсик с позиции ведущего языка для программирования микрокомпьютеров. В 1980-х годах он был адаптирован для использования в IBM PC, что привело к резкому росту его популярности. В то же время Бьярне Строуструп и другие в лабораториях Bell Labs начали работу по добавлению в Си возможностей объектно-ориентированного программирования. Язык, который они в итоге сделали, C++, в настоящее время является самым распространённым языком программирования. Си остаётся более популярным в UNIX-подобных системах.

- В 1983 году Американский Национальный Институт Стандартизации (ANSI) сформировал комитет для разработки стандартной спецификации Си. По окончании этого долгого и сложного процесса в 1989 году он был наконец утверждён как «Язык программирования Си» ANSI X3.159-1989. Эту версию языка принято называть ANSI C или C89. В 1990 году стандарт ANSI C был принят с небольшими изменениями Международной Организацией по Стандартизации (ISO) как ISO/IEC 9899:1990.

- Одной из целей этого стандарта была разработка надмножества K&R C, включающего многие особенности языка, созданные позднее. Однако комитет по стандартизации также включил в него и несколько новых возможностей, таких как прототипы функций (заимствованные из C++) и более сложный препроцессор.

- ANSI C сейчас поддерживают почти все существующие компиляторы. Почти весь код Си, написанный в последнее время, соответствует ANSI C. Любая программа, написанная только на стандартном Си, гарантированно будет правильно выполняться на любой платформе, имеющей соответствующую реализацию Си. Однако большинство программ написаны так, что они будут компилироваться и исполняться только на определённой платформе, потому, что:
- они используют нестандартные библиотеки, например, для графических дисплеев;
- они используют специфические платформо-зависимые средства;
- они рассчитаны на определённое значение размера некоторых типов данных или на определённый способ хранения этих данных в памяти для конкретной платформы.

C99

- После стандартизации в ANSI спецификация языка Си оставалась относительно неизменной в течение долгого времени, в то время как Си++ продолжал развиваться (в 1995 году в стандарт Си была внесена Первая нормативная поправка, но её почти никто не признавал). Однако в конце 1990-х годов стандарт подвергся пересмотру, что привело к публикации ISO 9899:1999 в 1999 году. Этот стандарт обычно называют «C99». В марте 2000 года он был принят и адаптирован ANSI.

- Вот некоторые новые особенности C99:
- подставляемые функции (inline);
- отсутствие ограничений на место объявления локальных переменных (как и в C++);
- новые типы данных, такие как long long int (для облегчения перехода от 32- к 64-битным числам), явный булевый тип данных _Bool и тип complex для представления комплексных чисел;
- массивы переменной длины;
- поддержка ограниченных указателей (restrict);
- именованная инициализация структур: struct { int x, y, z; } point = { .y=10, .z=20, .x=30 };
- поддержка однострочных комментариев, начинающихся на //, заимствованных из C++ (многие компиляторы Си поддерживали их и ранее в качестве дополнения);
- несколько новых библиотечных функций, таких как snprintf;
- несколько новых заголовочных файлов, таких как stdint.h.

- Интерес к поддержке новых особенностей C99 в настоящее время смешан. В то время как GCC[2], компилятор Си от Sun Microsystems и некоторые другие компиляторы в настоящее время поддерживают большую часть новых особенностей C99, компиляторы компаний Borland и Microsoft не делают этого, причём похоже, что две эти компании и не думают их добавлять.

Связь с C++

- Язык программирования C++ произошёл от Си. Однако в дальнейшем Си и C++ развивались независимо, что привело к росту несовместимостей между ними. Последняя редакция Си, C99, добавила в язык несколько конфликтующих с C++ особенностей. Эти различия затрудняют написание программ и библиотек, которые могли бы нормально компилироваться и работать одинаково в компиляторах Си и C++, что, конечно, запутывает тех, кто программирует на обоих языках.

- Бьёрн Страуструп, придумавший C++, неоднократно выступал за максимальное сокращение различий между Си и C++ для создания максимальной совместимости между этими языками. Противники же такой точки зрения считают, что так как Си и C++ являются двумя различными языками, то и совместимость между ними не так важна, хоть и полезна. Согласно этому лагерю, усилия по уменьшению несовместимости между ними не должны препятствовать попыткам улучшения каждого языка в отдельности.

C++

- C++ (произносится «си плас плас», допустимо также русскоязычное произношение «си плюс плюс») — компилируемый статически типизированный язык программирования общего назначения. Поддерживая разные парадигмы программирования, сочетает свойства как высокоуровневых, так и низкоуровневых языков. В сравнении с его предшественником — языком C, — наибольшее внимание уделено поддержке объектно-ориентированного и обобщённого программирования. Название «C++» происходит от названия языка C, в котором унарный оператор ++ обозначает инкремент переменной.

- Являясь одним из самых популярных языков программирования, C++ широко используется для разработки программного обеспечения. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также развлекательных приложений (например, видеоигры). Существует несколько реализаций языка C++ — как бесплатных, так и коммерческих. Их производят Проект GNU, Microsoft, Intel и Embarcadero (Borland). C++ оказал огромное влияние на другие языки программирования, в первую очередь на Java и C#.

- При создании C++ Бьёрн Страуструп стремился сохранить совместимость с языком C. Множество программ, которые могут одинаково успешно транслироваться как компиляторами C, так и компиляторами C++, довольно велико — отчасти благодаря тому, что синтаксис C++ был основан на синтаксисе C.

Ключевые слова
В C89 есть 32 ключевых слова:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Приоритет операций

Лексемы	Операция	Класс	Приоритет	Ассоциативность
имена, литералы	простые лексемы	первичный	16	нет
a[k]	индексы	постфиксный	16	слева направо
f(…)	вызов функции	постфиксный	16	слева направо
.	прямой выбор	постфиксный	16	слева направо
->	опосредованный выбор	постфиксный	16	слева направо
++ --	положительное и отрицательное приращение	постфиксный	16	слева направо
(имя типа) {init}	составной литерал (C99)	постфиксный	16	слева направо
++ --	положительное и отрицательное приращение	префиксный	15	справа налево
sizeof	размер	унарный	15	справа налево
~	побитовое НЕ	унарный	15	справа налево
!	логическое НЕ	унарный	15	справа налево
- +	изменение знака, плюс	унарный	15	справа налево
&	адрес	унарный	15	справа налево
*	опосредование (разыменование)	унарный	15	справа налево
(имя типа)	приведение типа	унарный	14	справа налево

Приоритет операций

* / %	мультипликативные операции	бинарный	13	слева направо
+ -	аддитивные операции	бинарный	12	слева направо
<< >>	сдвиг влево и вправо	бинарный	11	слева направо
< > <= >=	отношения	бинарный	10	слева направо
== !=	равенство/неравенство	бинарный	9	слева направо
&	побитовое И	бинарный	8	слева направо
^	побитовое исключающее ИЛИ	бинарный	7	слева направо
|	побитовое ИЛИ	бинарный	6	слева направо
&&	логическое И	бинарный	5	слева направо
||	логическое ИЛИ	бинарный	4	слева направо
? :	условие	тернарный	3	справа налево
= += -= *= /= %= <<= >>= &= ^= |=	присваивание	бинарный	2	справа налево
,	последовательная оценка	бинарный	1	слева направо

Базовые типы данных языка С

Название типа	Пояснения	Диапазон значений
`short`	Краткое целое число	-128 . . . 127
`unsigned short`	Краткое целое число без знака	0 . . . 255
`int`	Целое число	-32768 . . . 32767
`unsigned int`	Целое число	0 . . . 65535
`long`	Длинное целое число	-2^{30} . . . $2^{30}-1$
`unsigned long`	Длинное целое число без знака	0 . . . $2^{31}-1$
`char`	Один символ	символы кода ASCII
`char[ ]`	Строка	
`float`	Число с плавающей точкой	$3.4\ 10^{-38}$ . . . $3.4\ 10^{+38}$
`double`	Число с плавающей точкой двойной точности	$1.7\ 10^{-308}$ . . . $1.7\ 10^{+308}$

Hello в стиле СИ

- //*********prog1.cpp*********
- #include<stdio.h>
- void main(void)
- {
- printf("Hello\n");
- }

Hello в стиле C++

```
//*********prog2.cpp*********
#include<iostream.h>
void main(void)
{
cout<<"Hello"<<endl;
}
```

Hello в стиле C++ на современных компиляторах

```
//*********prog2.cpp*********
#include<iostream>
using namespace std;
int main(void)
{
cout<<"Hello"<<endl;
return 0;
}
```

Использование переменных

Любая переменная, используемая в программе, должна быть описана перед первым её использованием. Описать переменную значит указать её имя и тип.

- //*********prog3.cpp*********
- #include<stdio.h>
- void main(void)
- {
- float a,b,c; //Описаны 3 вещественных переменных
- a=10; b=5;
- c=a/b;
- printf("c=%f\n",c);
- }

Некоторые функции стандартного ввода-вывода

Функции стандартного ввода - вывода описаны в файле ***stdio.h***.

- ***printf()*** *- форматный вывод на экран:*
- *int printf(char *format, <список вывода>);*
- *Первый параметр является символьной строкой, которая выводится в поток вывода (экран). В ней могут встречаться спецификаторы формата. Остальные параметры - перечисление переменных и выражений, значения которых выводятся. Каждая спецификация формата имеет вид (параметры в квадратных скобках необязательны):*
- *%[flags][width][.prec]type*
- *Как только в строке встречается спецификатор формата, он замещается значением очередной переменной из списка.*

%[flags][width][.prec]type

где	type -	тип спецификации
	d или i	целое десятичное число со знаком
	u	десятичное число без знака
	x	целое 16-ричное число без знака
	f	число с плавающей точкой
	e	число в E-форме
	g	число с плавающей точкой или в E-форме
	c	один символ
	s	строка
	%	символ %
	flags -	признак выравнивания:
	+ или пусто	выравнивание по правому краю
	-	выравнивание по левому краю
	width -	целое число - общая ширина поля. Если это число начинается с цифры 0, вывод дополняется слева нулями до заданной ширины. В заданную ширину входят все символы вывода, включая знак, дробную часть и т.п.
	prec -	целое число, количество знаков после точки при выводе чисел с плавающей точкой

- ***scanf()*** *- форматный ввод с клавиатуры:*
- *int scanf(char *format, <список ввода>); Первый параметр является символьной строкой, которая задает спецификации формата (см. функцию* ***printf()****). Остальные параметры - перечисление адресов переменных, в которые вводятся данные. В этом списке перед именами всех переменных, кроме тех, которые вводятся по спецификации типа* ***%s****, должен стоять символ* ***&.***

```
//*********prog4.cpp*********
#include<stdio.h>
void main(void)
{
float a,b,c;
printf(“input a:”);
scanf(“%f”,&a);
printf(“input b:”);
scanf(“%f”,&b);
c=a/b;
printf("c=%f\n",c);
}
```

Вывод значений нескольких переменных

- //*********prog4.cpp*********
- #include<stdio.h>
- void main(void)
- {
- float a=1.5;
- int b=7;
- char c=‘A’;
- char str[]=“Stroka”;
- printf(“a=%f b=%d c=%c str=%s\n",a,b,c,str);
- }
- На экране увидим
- a=1.5 b=7 c=A str=Stroka

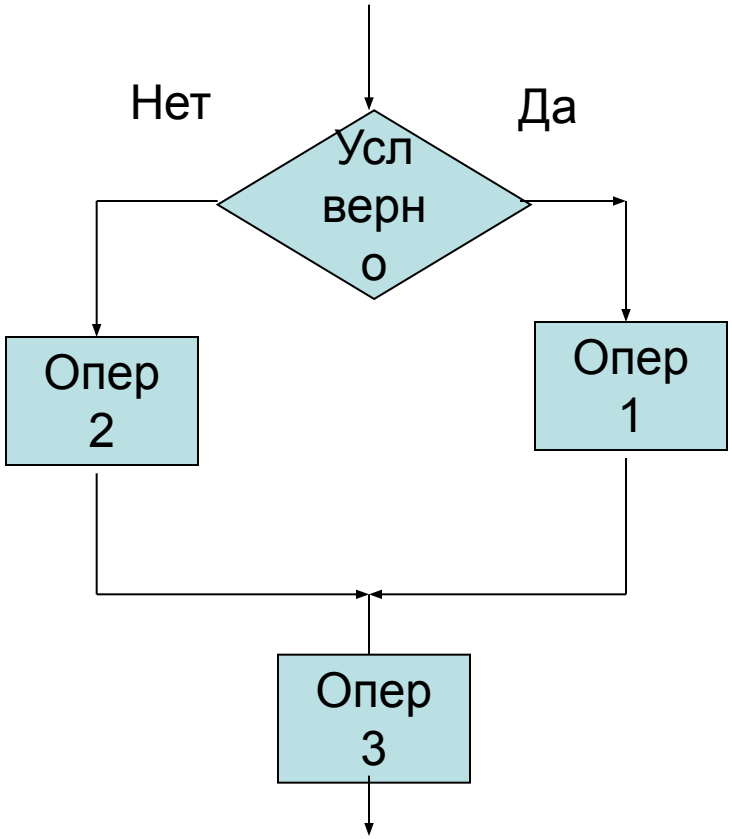
Ввод вывод в C++

```
//*********prog5.cpp*********
#include<iostream.h>
void main(void)
{
float a,b,c;
cout<<“input a”;
cin>>a;
cout<<“input b”;
cin>>b;
c=a/b;
cout<<”c=”<<c<<endl;
}
```

Условный оператор if

Полная форма

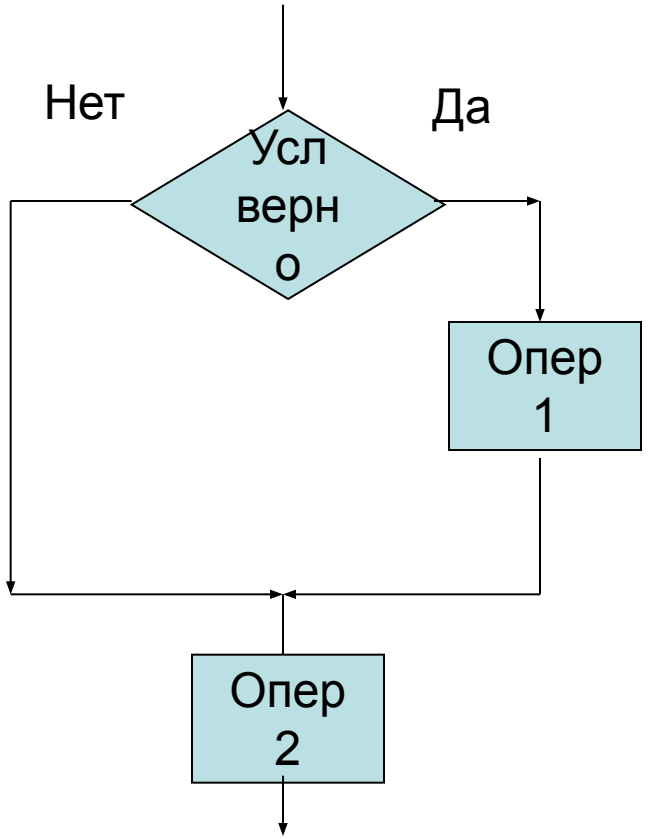
```
if(условие)
Опер1;
else Опер 2;
Опер 3;
```



Условный оператор if

Краткая форма

if(условие)Опер1;
Опер 2;



Логические операции

- Язык С имеет ровно три логические операции: это
- && или (AND);
- || или (OR);
- ! или (NOT).
- Как принято еще называть логические операции?

- Операция "&&" или операция "AND" называется еще операцией "и" или логическим умножением.
- Операция "||" или операция "OR" называется еще операцией "или" или логическим сложением.
- Операция "! или операция "NOT" называется еще операцией "не" или логическим отрицанием.
-

Таблицы истинности логических операций

- **Операция "&&"** называется логическим умножением потому, что выполняется таблица истинности этой операции, очень напоминающая таблицу обыкновенного умножения из арифметики.
- Логическое умножение это такая операция, которая истинна тогда и только тогда, когда истинны оба входящих в нее высказывания.

- 1 && 1 = 1
- 0 && 1 = 0
- 1 && 0 = 0
- 0 && 0 = 0

- **Операция "||"** (ИЛИ) называется логическим сложением потому, что выполняется таблица истинности этой операции, очень напоминающая таблицу обыкновенного сложения из арифметики.
- Логическое сложение это такая операция, которая истинна тогда и только тогда, когда истинно хотя бы одно из входящих в нее высказываний.

- 1 || 1 = 1
- 0 || 1 = 1
- 1 || 0 = 1
- 0 || 0 = 0

- **Операция "!"** (НЕ) называется логическим отрицанием потому, что выполняется следующая таблица истинности.
- Логическое отрицание это такая операция, которая истинна тогда и только тогда, когда ложно входящее в нее высказывание и наоборот.

- !1 = 0
- !0 = 1

Пример с полной формой if

30	$\begin{cases} 27+(x-3)^3 & \text{при } x>3, \\ x^3 & \text{при } 3>=x>1, \\ x & \text{при } 1>=x>0, \\ \frac{\sin^2 x}{2} & \text{при } x<=0 \end{cases}$	13

- /* Объявления переменных *x* и *y* и ввод исходных данных */
- **if(** x > 3)
- y = 27 + pow(x -3, 3);
- **else if**(x > 1)
- y = pow(x, 3);
- **else if(** x > 0)
- y = x;
- **else**
- y = pow(sin(x), 2) / 2;
- /* Вывод значения переменной “y” */

Пример с краткой формой if

30	$\begin{cases} 27+(x-3)^3 & \text{при } x>3, \\ x^3 & \text{при } 3>=x>1, \\ x & \text{при } 1>=x>0, \\ \frac{\sin^2 x}{2} & \text{при } x<=0 \end{cases}$	13

- /* Объявления переменных “x” и “y” и ввод исходных данных */
- **if**(x > 3) y = 27 + pow(x –3, 3);
- **if**(x <= 3 && x > 1) y = pow(x, 3);
- **if**(x <= 1 && x > 0) y = x;
- **if(**x >= 0) y = pow(sin(x), 2) / 2 ;
- /* Вывод значения переменной “y” */

Операции инкрементации и декрементации

- Операции инкрементации и декрементации являются унарными операциями, то есть операциями, имеющими один операнд.
- операнд++ //Постфиксная
- ++операнд //Префиксная
- Операция инкрементации ++ добавляет к операнду единицу.
- операнд-- //Постфиксная
- --операнд //Префиксная
- Операция декрементации -- вычитает из операнда единицу.

- Операндом может быть именующее выражение, например, имя переменной.
- Следующие три строки увеличивают переменную x на 1:
- x = x + 1;
- ++x;
- x++;

Префиксная (++x, --x) и постфиксная (x++ , x--) форма

- Операции инкрементации и декрементации имеют
- префиксную (++x, --x) и
- постфиксную (x++ , x--)
- форму записи.
- **При использовании префиксной формы записи операнд увеличивается или уменьшается сразу же.**
- Пример 1
- x = 3;
- y = ++x;
- Переменная x сразу же увеличивается до 4 и это значение присваивается переменной y.
- **При использовании постфиксной формы записи операнд увеличивается или уменьшается после того, как он используется.**
- Пример 2
- x = 3;
- y = x++;
- Переменной y присваивается значение 3, а затем переменная x увеличивается до 4.

Сложное присваивание

Сложное присваивание	Аналог
y+=5;	y=y+5;
y-=5;	y=y-5;
y*=5;	y=y*5;
y/=5;	y=y/5;

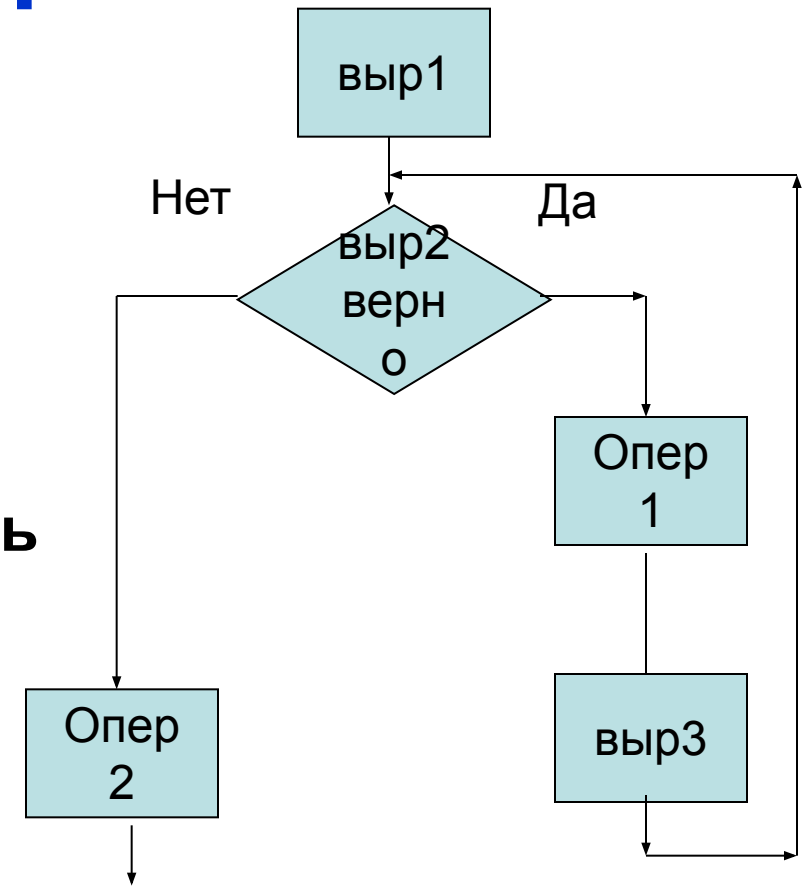
Операторы циклов

- for
- while
- do …..while

Оператор for

for(выр1; выр2;выр3)
Опер1;
Опер 2;

выр1-инициализационная часть
выр2-проверочная
выр3-послецикловая



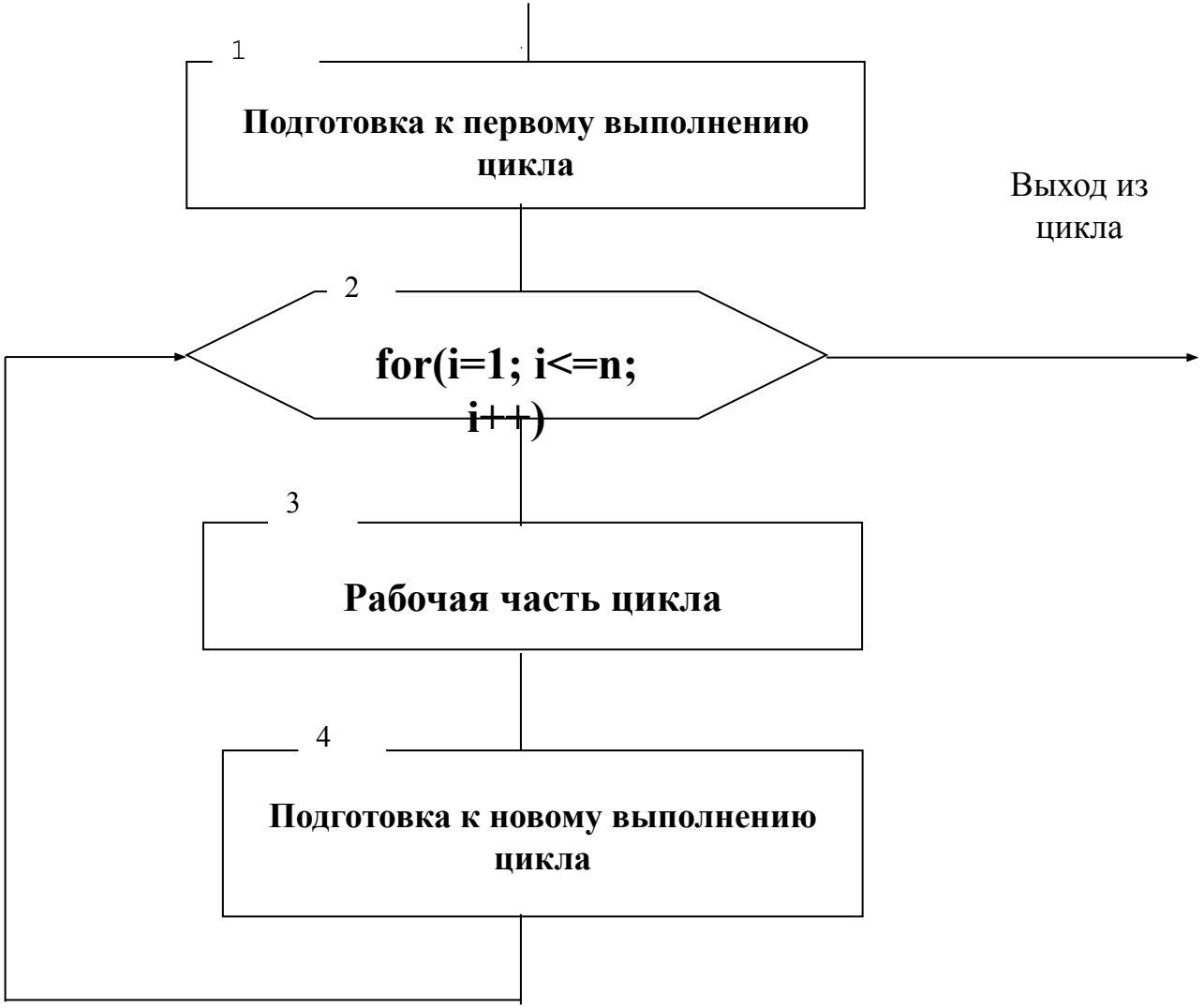
Пример

```
int i;
for( i=1;i<=5; i++)
   cout<<i;
```

На экране увидим: 12345

Переменную i обычно называют счетчиком цикла;

cout<<i; в данном случае является телом цикла

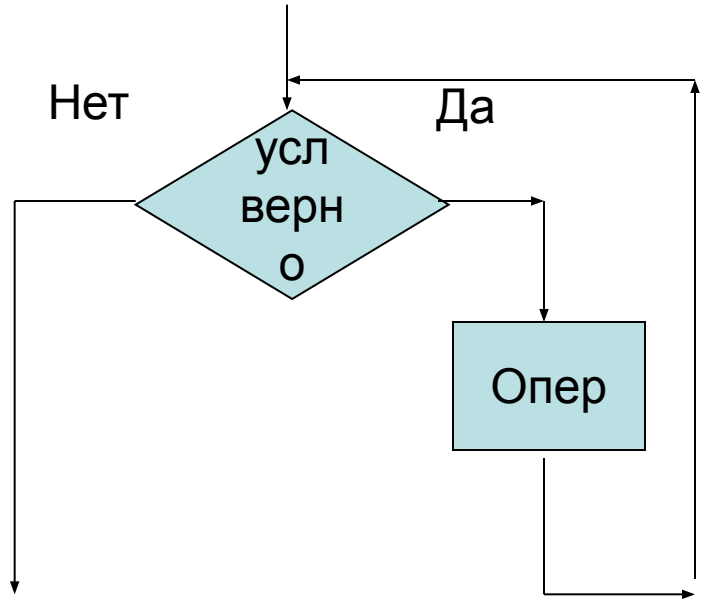


Обобщенная схема алгоритма

Оператор while

while(условие)
Опер;

Цикл с предусловием



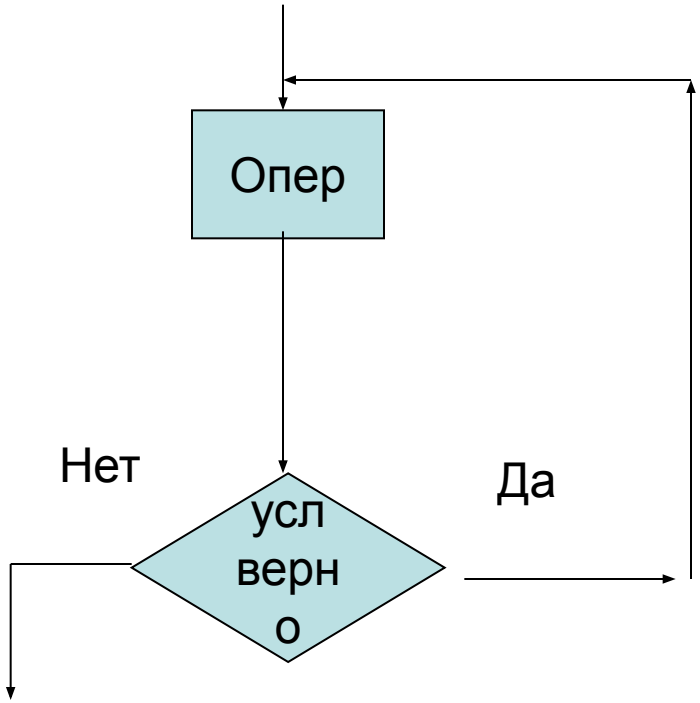
Пример

```
int i;
i=1;
while( i<=5)
   {
    cout<<i;
    i++;
   }
```

На экране увидим: 12345

Оператор do while

```
do
{
    Опер;
}
while(условие);
```



Цикл с постусловием
Тело цикла обязательно выполнится хоть 1 раз

Пример

```
int i;
i=1;
   do
   {
    cout<<i;
    i++;
   } while( i<=5);
```

На экране увидим: 12345

Сравнение операторов циклов

```
int i;
for( i=1;i<=5; i++)
   cout<<i;
```

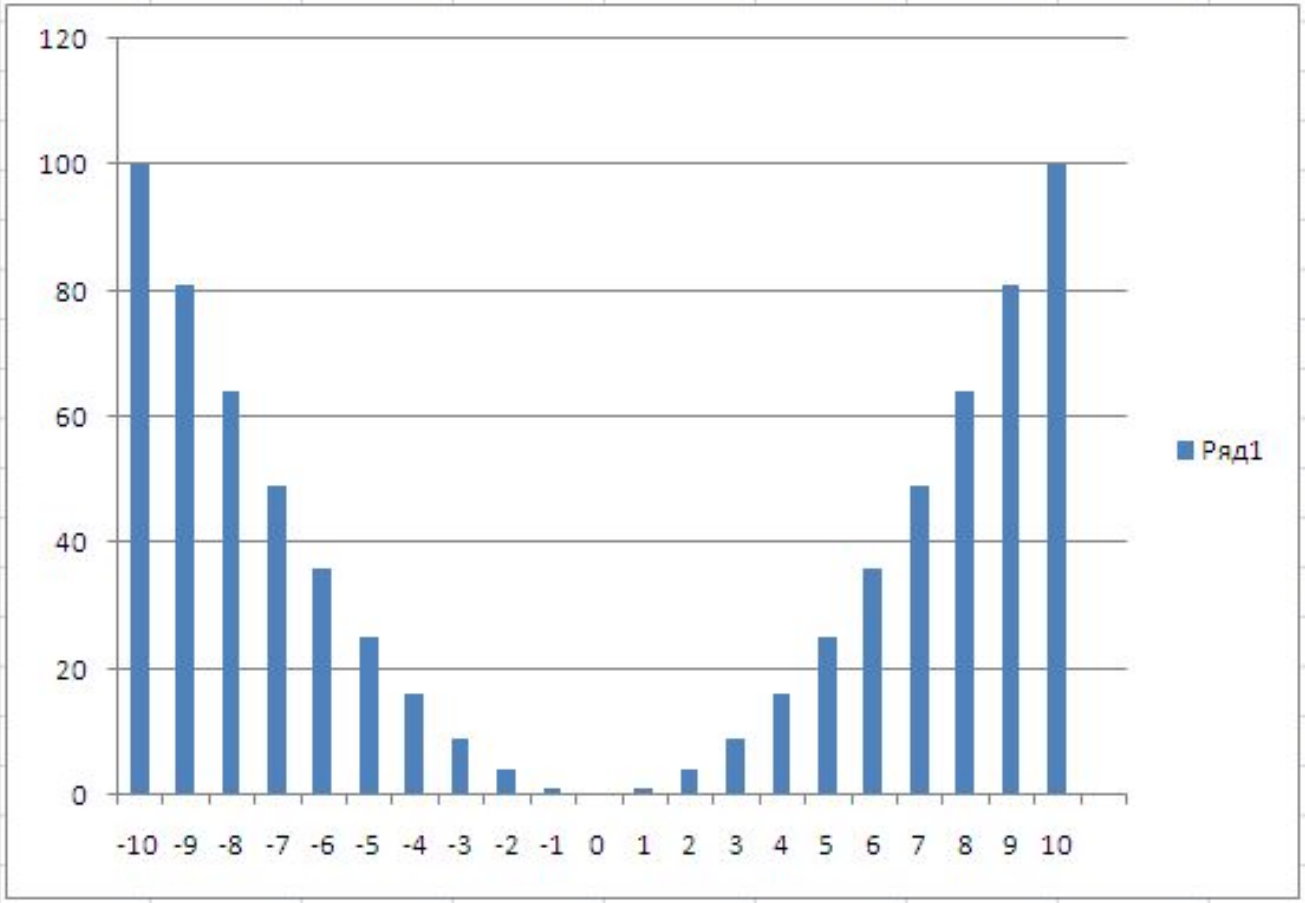
```
int i;
i=1;
while( i<=5)
{
     cout<<i;
     i++;
}
```

```
int i;
i=1;
do
 {
     cout<<i;
     i++;
} while( i<=5);
```

Задача табулирования

x	y
-10	100
-9	81
-8	64
-7	49
-6	36
-5	25
-4	16
-3	9
-2	4
-1	1
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

xn=-10 xk=10 N=11 y=x*x



Задача табулирования

```
//********* for ( format output) ****************
#include<iostream.h>
#include<iomanip.h>
void main(viod)
{
 float x,y,xn,xk,dx;  int n;
 cout<<"xn= "; cin>>xn;
 cout<<"xk= "; cin>>xk;
 cout<<"n=  "; cin>>n;
 dx=(xk-xn)/(n-1);   x=xn;
 cout<<setw(5)<<"i"<<setw(10)<<setprecision(3)
 <<"x"<<setw(10)<<setprecision(3)<<"y"<<endl;
 for( int i=0;i<n;i++,x+=dx)
 { y=x*x;
   cout<<setw(5)<<i<<setw(10)<<setprecision(3)
   <<x<<setw(10)<<setprecision(3)<<y<<endl;
 }
}
```

Операторы

- break
- continue

Операторы break и continue

Часто при возникновении некоторого события удобно иметь возможность досрочно завершить цикл.
Используемый для этой цели оператор break (разрыв) вызывает немедленный выход из циклов, организуемых с помощью операторов for, while, do-while, а также прекращение оператора switch.

```
#include <stdio.h>
int main(void)
 {
 int i;
 for(i=1;i<10;i++)
{
      if(i==5)
          break;
           printf(“%d” ,i);
}
return 0;
}
```

На экране увидим 1234

Оператор continue

Оператор continue тоже предназначен для прерывания циклического процесса, организуемого операторами for, while, do-while. Но в отличии от оператора break, он не прекращает дальнейшее выполнение цикла, а только немедленно переходит к следующей интерации того цикла, в теле которого он оказался. Он как бы имитирует безусловный переход на конечный оператор цикла, но не за ее пределы самого цикла.

```
#include <stdio.h>
int main(void)
 {
 int i;
 for(i=1;i<10;i++)
{
      if(i==5)
          continue;
           printf(“%d” ,i);
}
return 0;
}
```

На экране увидим 12346789

Переключатель switch

Оператор switch (переключатель) предназначен для принятия одного из многих решений. Он выглядит следующим образом:

- switch(целое выражение)
- {
- case константа1: оператор1;
- case константа2: оператор2;
- ...
- ...
- ...
- case константаn: операторn;
- default : оператор;
- }

При выполнении этого оператора вычисляется выражение, стоящее в скобках после ключевого слова switch, которое должно быть целым. Оно, в частности, может быть и символьным значением (в языке Си символьные значения автоматически расширяются до целых значений). Эта целая величина используется в качестве критерия для выбора одного из возможных вариантов. Ее значение сравнивается с константой операторов case. Вместо целой или литерной константы в операторе case может стоять некоторое константное выражение. Значения таких констант (выражений) должны быть различными в разных операторах case. При несовпадении выполняется переход к следующему case и сравнивается его константа. В случае совпадения "константы_i" выполняется "оператор_i", а также все последующие операторы case и default. Если не было ни одного совпадения и имеется оператор default, то выполняется стоящий за ним оператор. Если же оператора default не было, выполнение программы продолжится с оператора, следующего за структурой switch. Таким образом, при каждом выполнении оператора просматриваются все метки case.

Пример - калькулятор

```
 #include <stdio.h>
 main()
 {
 int a,b,c;
 char op;
printf( “ Input a op b”):
scanf(“%d “&a); scanf(“%c “&op);
scanf(“%d “&b);
switch(op)
 {
   case ‘+’:c=a+b;
   case ‘-’: c=a-b;
   case ‘*’: c=a*b;
   case ‘/’: c=a/b;
   default: printf(“ERROR!!!\n");
   }
 printf(“%d “,c);
 }
```

Данный пример работать не будет и мы всегда будем видеть ERROR!!! даже при вводе правильного выражения. Происходит это потому, что выполнится не только нужный нам оператор, а также и все последующие операторы case, а также вариант default. Чтобы обеспечить выбор одного из многих вариантов (что нам и требуется), используют обычно оператор break, который вызывает немедленный выход из оператора switch

Калькулятор (правильный)

- **Пример - калькулятор**
- **#include <stdio.h>**
- **main()**
- **{**
- **int a,b,c; char op;**
- **printf(“ Input a op b”):**
- **scanf(“%d “&a); scanf(“%c “&op);**
- **scanf(“%d “&b);**
- **switch(op)**
- **{**
- **case ‘+’:c=a+b; break;**
- **case ‘-’: c=a-b; break;**
- **case ‘*’: c=a*b; break;**
- **case ‘/’: c=a/b; break;**
- **default: printf(“ERROR!!!\n");**
- **}**
- **printf(“%d “,c);**
- **}**

Массивы

Массив - это упорядоченная совокупность данных одного типа. Можно говорить о массивах целых чисел, массивов символов и. т.д. Мы можем даже определить массив, элементы которого - массивы(массив массивов), определяя, таким образом, многомерные массивы. Любой массив в программе должен быть описан: после имени массива добавляют квадратные скобки [], внутри которых обычно стоит число, показывающее количество элементов массива. Например, запись int x[10]; определяет x как массив из 10 целых чисел.

В случае многомерных массивов показывают столько пар скобок , какова размерность массива, а число внутри скобок показывает размер массива по данному измерению. Например, описание двумерного массива выглядит так: int a[2][5];. Такое описание можно трактовать как матрицу из 2 строк и 5 столбцов. Для обращения к некоторому элементу массива указывают его имя и индекс, заключенный в квадратные скобки(для многомерного массива - несколько индексов , заключенные в отдельные квадратные скобки): a[1][3], x[i] a[0][k+2]. Индексы массива в Си всегда начинаются с 0, а не с 1, т.е. описание int x[5]; порождает элементы x[0], x[1], x[2], x[3], x[4]. Индекс может быть не только целой константой или целой переменной, но и любым выражением целого типа. Переменная с индексами в программе используется наравне с простой переменной (например, в операторе присваивания, в функциях ввода- вывода)..

Элементам массива могут быть присвоены начальные значения:

```
int a[6]={5,0,4,-17,49,1};
```

приведенная запись обеспечивает присвоения a[0]=5; a[1]=0; a[2]=4 ... a[5]=1. Для начального присвоения значений некоторому массиву надо в описании поместить справа от знака = список инициирующих значений, заключенные в фигурные скобки и разделенные запятыми

```
//********* mass1_sum.cpp ***
#include <iostream.h>
#define N 10
void main(void)
{
 int i;
 double sum;
//Определение массива
double arr[10];
//Ввод массива
 for(i=0;i<10;i++)
 {
 cout<<" arr["<<i<<"]=";
 cin>>arr[i];
 }
```

```
//Обработка массива
 sum=0;
 for(i=0;i<N;i++)
 {
  sum+=arr[i];
 }

//Вывод массива
 for(i=0;i<N;i++)
 {
 cout<<" arr["<<i<<"]="<<arr[i]<<endl;
 }
 cout<<" Sum="<<sum<<endl;
}
```

Расположение массивов в памяти

```
double arr[]={0.1,1.1,2.1,3.1,4.1,5.1,6.1,7.1,8.1,9.1};
 for(i=0;i<10;i++)
 {
 cout<<" arr["<<i<<"]="<<arr[i]<<"    addr="<<&arr[i]<<endl;
 }
```

- arr[0]=0.1 addr=0x1ebd0fa8
- arr[1]=1.1 addr=0x1ebd0fb0
- arr[2]=2.1 addr=0x1ebd0fb8
- arr[3]=3.1 addr=0x1ebd0fc0
- arr[4]=4.1 addr=0x1ebd0fc8
- arr[5]=5.1 addr=0x1ebd0fd0
- arr[6]=6.1 addr=0x1ebd0fd8
- arr[7]=7.1 addr=0x1ebd0fe0
- arr[8]=8.1 addr=0x1ebd0fe8
- arr[9]=9.1 addr=0x1ebd0ff0

```
  0x1ebd0fa8
+            8
= 0x1ebd0fb0
```

Размер переменной типа double 8 байт

Многомерные массивы

- Многомерные массивы - это массивы с более чем одним индексом.

- Чаще всего используются двумерные массивы.

- При описании многомерного массива
- необходимо указать С++,
- что массив имеет более чем одно измерение.
- int t[3][4];
- Описывается двумерный массив, из 3 строк
- и 4 столбцов.
- Элементы массива:
- t[0][0] t[0][1] t[0][2] t[0][3]
- t[1][0] t[1][1] t[1][2] t[1][3]
- t[2][0] t[2][1] t[2][2] t[2][3]

- При выполнении этой команды под массив резервируется место.
- Элементы массива располагаются в памяти один за другим.

- В памяти многомерные массивы представляются как одномерный массив, каждый из элементов которого, в свою очередь, представляет собой массив.

- Рассмотрим на примере двумерного массива.
- int a[3][2]={4, l, 5,7,2, 9};

- Представляется в памяти:

- a[0][0] заносится значение 4
- a[0][1] заносится значение 1
- a[1][0] заносится значение 5
- a[1][1] заносится значение 7
- a[2][0] заносится значение 2
- a[2][1] заносится значение 9

- Второй способ инициализации при описании массива
- int a[3][2]={ {4,1}, {5, 7}, {2, 9} };

- Обращение к элементу массива производится через индексы.
- cout<< a[0][0];

- **Программа инициализирует массив и выводит его элементы на экран.**
- **#include <iostream.h>**
- **int main ()**
- **{**
- **int a[3] [2]={ {1,2}, {3,4}, {5,6} };**
- **int i,j;**
- **for (i=0; i<3; i++)**
- **for(j=0;j<2;j++)**
- **cout <<"\n a["<< i <<"," << j <<"] ="<< a[i][j];**
- **return 0;**
- **}**

```
//Ввод массива
int a[3] [2];
int i,j;
for (i=0; i<3; i++)
    for(j=0;j<2;j++)
    {
      cout <<"a["<< i <<"][" << j <<"] =";
      cin>> a[i][j];
    }
```

- //обработка массива (сумма элем.)

- int s=0;
- for (i=0; i<3; i++)
  for(j=0;j<2;j++)
- s+=a[i][j];

```
//вывод на экран


for (i=0; i<3; i++)
{
    for(j=0;j<2;j++)
        cout <<setw(5)<<a[i][j];
    cout<<endl;
 }
```

Указатели

- Указатели — это переменные, которые хранят адрес объекта (переменной) в памяти.
- Для объявления указателя нужно добавить звездочку перед именем переменной. Так, например, следующий код создает два указателя, которые указывают на целое число.
- **int *pNumberOne;**
- **int *pNumberTwo;**
- Обратили внимание на префикс "p" в обоих именах переменных? Это принятый способ обозначить, что переменная является указателем. Так называемая венгерская нотация.

- Теперь сделаем так, чтобы указатели на что-нибудь указывали:

- **int some_number=5, some_other_number=10;**
- **pNumberOne = &some_number;**
- **pNumberTwo = &some_other_number;**

- Знак & (амперсанд) следует читать как "адрес переменной ..." и означает адрес переменной в памяти, который будет возвращен вместо значения самой переменной. Итак, в этом примере pNumberOne установлен и содержит адрес переменной some_number (указывает на some_number).

- Если мы хотим получить адрес переменной some_number, мы можем использовать pNumberOne. Если мы хотим получить значение переменной some_number через pNumberOne, нужно добавить звездочку (*) перед pNumberOne (*pNumberOne). Звездочка (*) разыменовывает (превращает в саму переменную) указатель.
- **cout<< pNumberOne; //Увидим адрес**
- **cout<< *pNumberOne; //Увидим значение**

```
#include <stdio.h>
void main()
{
// объявляем переменные:
int nNumber;
int *pPointer;
// инициализируем объявленные переменные:
nNumber = 15;
pPointer = &nNumber;
// выводим значение переменной nNumber:
printf("nNumber is equal to : %d\n", nNumber);
// теперь изменяем nNumber через pPointer:
*pPointer = 25;
// убедимся что nNumber изменил свое значение
// в результате предыдущего действия,
// выведя значение переменной ещё раз
printf("nNumber is equal to : %d\n", nNumber);
}
```

Динамическая память

- Динамическая память позволяет выделять/освобождать память во время работы программы. Код ниже демонстрирует, как выделить память для целого числа:
- **int *pNumber;**
- **pNumber = new int;**
- Первая строчка объявляет указатель pNumber. Вторая строчка выделяет память для целого числа (int) и указывает pNumber на эту область памяти. Вот ещё один пример, на этот раз с числом с двойной точностью (double).
- **double *pDouble;**
- **pDouble = new double;**

Освобождение памяти

С памятью всегда существуют сложности и в данном случае довольно серьезные, но эти сложности можно с легкостью обойти. Проблема заключается в том что не смотря на то, что память которая была динамически выделена остается нетронутой она никогда не освобождается автоматически. Память будет оставаться выделенной до тех пор, пока вы не скажете компьютеру что вам она больше не нужна. Проблема в том, что если вы не скажете системе, что память вам больше не нужна, она будет занимать место, которое возможно необходимо другим приложениям, либо частям вашего приложения. В частности это может привести к сбою системы по причине использования всей доступной памяти, поэтому это очень важно. Освобождение памяти когда она вам больше не нужна делается очень просто:

- **delete pPointer;**

Операции с указателями

Унарные операции: инкремент и декремент. При выполнении операций ++ и -- значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель.

Пример:

```
int *ptr, a[10];
ptr=&a[5];
ptr++; /* равно адресу элемента a[6] */
ptr--; /* равно адресу элемента a[5] */
```

Операции с указателями

В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа int. При этом результатом операции будет указатель на исходный тип, а его значение будет на указанное число элементов больше или меньше исходного.

Пример:

```
int *ptr1, *ptr2, a[10];
int i=2;
ptr1=a+(i+4); /* равно адресу элемента a[6] */
ptr2=ptr1-i; /* равно адресу элемента a[4] */
```

Операции с указателями

В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип int и равен числу элементов исходного типа между уменьшаемым и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение.

Пример:

```
int *ptr1, *ptr2, a[10];
int i;
ptr1=a+4;
ptr2=a+9;
i=ptr1-ptr2; /* равно 5 */
i=ptr2-ptr1; /* равно -5 */
```

Операции с указателями

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=",">, >= при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина).

Пример:

```
int *ptr1, *ptr2, a[10];
ptr1=a+5;
ptr2=a+7;
if (prt1>ptr2) a[3]=4;
```

В данном примере значение ptr1 меньше значения ptr2 и поэтому оператор a[3]=4 не будет выполнен.

Методы доступа к элементам массивов

Для доступа к элементам массива существует два различных способа. Первый способ связан с использованием обычных индексных выражений в квадратных скобках, например, array[16]=3 или array[i+2]=7. При таком способе доступа записываются два выражения, причем второе выражение заключается в квадратные скобки. Одно из этих выражений должно быть указателем, а второе - выражением целого типа.

Методы доступа к элементам массивов

Второй способ доступа к элементам массива связан с использованием адресных выражений и операции разадресации в форме *(array+16)=3 или *(array+i+2)=7.

При реализации на компьютере первый способ приводится ко второму, т.е. индексное выражение преобразуется к адресному. Для приведенного примера array[16] преобразуются в *(array+16).

Функции

Мощность языка программирования С во многом определяется легкостью и гибкостью в определении и использовании функций в программах на языке программирования С. В отличие от других языков программирования высокого уровня в языке программирования С нет деления на процедуры, подпрограммы и функции, здесь вся программа строится только из функций.

Функция - это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на С должна быть функция с именем main (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

функции

При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры), используемые во время выполнения функции. Функция может возвращать некоторое (одно !) значение. Это возвращаемое значение и есть результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов ни встретился. Допускается также использовать функции не имеющие аргументов и функции не возвращающие никаких значений. Действие таких функций может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т.п..

Функции

С использованием функций в языке СИ связаны три понятия:

- **определение функции** (описание действий, выполняемых функцией)
- **объявление функции** (задание формы обращения к функции)
- **вызов функции**.

Определение функции задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления переменных и операторы, называемые телом функции, и определяющие действие функции. Пример:

```
int max ( int a, int b)

{  if (a>b)
      return a;
   else
      return b;
}
```

В данном примере определена функция с именем max, имеющая 2 параметра. Функция возвращает целое значение (максимальное из a и b.

- В языке СИ нет требования, чтобы определение функции обязательно предшествовало ее вызову. Определения используемых функций могут следовать за определением функции main, перед ним, или находится в другом файле.

- Чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров до вызова функции нужно поместить **объявление (прототип) функции**.

- **Объявление функции имеет такой же вид, что и определение функции**, с той лишь разницей, что тело функции отсутствует, и имена формальных параметров тоже могут быть опущены. Для функции, определенной в последнем примере, прототип может иметь вид

- int max (int a, int b);

Функции

В программах на языке СИ широко используются, так называемые, библиотечные функции, т.е. функции предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

- В соответствии с синтаксисом языка СИ определение функции имеет следующую форму:

- [спецификатор-класса-памяти] [спецификатор-типа] имя-функции

- ([список-формальных-параметров])

- { тело-функции }

- Необязательный спецификатор-класса-памяти задает класс памяти функции, который может быть static или extern.

Функции (возвращаемое значение)

Функция возвращает значение если ее выполнение заканчивается оператором return, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата. Если оператор return не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора return), то возвращаемое значение не определено. Для функций, не использующих возвращаемое значение, должен быть использован тип void, указывающий на отсутствие возвращаемого значения. Если функция определена как функция, возвращающая некоторое значение, а в операторе return при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым.

Список-формальных-параметров

- Список-формальных-параметров - это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры - это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических параметров. Список-формальных-параметров может заканчиваться запятой (,) или запятой с многоточием (,...), это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но над дополнительными аргументами не проводится контроль типов.

- Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров рекомендуется указать слово void.

Формальные параметры

Порядок и типы формальных параметров должны быть одинаковыми в определении функции и ее объявлении. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров. Тип формального параметра может быть любым основным типом, структурой, объединением, перечислением, указателем или массивом.

Передача параметров по значению

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются. Поскольку передача параметров происходит по значению, в теле функции нельзя изменить значения переменных в вызывающей функции, являющихся фактическими параметрами.

- Пример:
- /* Неправильное использование параметров */
- void change (int x, int y)
- { int k=x;
- x=y;
- y=k;
- }

- В данной функции значения переменных x и y, являющихся формальными параметрами, меняются местами, но поскольку эти переменные существуют только внутри функции change, значения фактических параметров, используемых при вызове функции, останутся неизменными.

Передача параметров по указателю

Однако, если в качестве параметра передать указатель на некоторую переменную, то используя операцию разадресации можно изменить значение этой переменной.

```
/*     Правильное использование параметров   */
       void change (int *x, int *y)
       {  int k=*x;
            *x=*y;
            *y=k;
       }
```

- При вызове такой функции в качестве фактических параметров должны быть использованы не значения переменных, а их адреса

```
change (&a,&b);
```

Передача параметров по ссылке

```
/*     Правильное использование параметров   */

       void change (int &x, int &y)

       {  int k=x;

            x=y;

            y=k;

       }
```

- Вызов такой функции:
- change (a,b);
  - Фактически передаются адреса!

Ввод массива

```
#include<stdio.h>
void vvod(float mas[],int n)
{
    int i;
    for(i=0; i<n; i++)
    {
        printf("mas[%d]=", i);
        scanf("%f", &mas[i]);
    }
}
```

Вывод массива

```
void vivod(float mas[], int n)
{
    int i;
    for(i=0; i<n; i++)
        printf("mas[%d]=%7.3f\n",i,mas[i]);
}
```

Обработка массива

(функция возвращает сумму отрицательных элементов)

```
float otr(float mas[],int n)
{
      int i;
     float s=0;
      for(i=0; i<n; i++)
         if(mas[i]<0)
           s+=mas[i];    //s=s+mas[i];
       return s;
}
```

Вызов функций

```
int main()
{
    float s;
    int n;
    char c;
    float a[10];

    printf("vvesti razmer\n");
    scanf("%d",&n);
    vvod(a,n);
    vivod(a,n);
    printf("sumotr=%7.3f\n",otr(a,n));

    scanf("%c\n",&c);
    return 0;
}
```

Лаб. Раб. 7 вар 9

Даны три числовые последовательности *a*, *b* и *c*. Сформировать две новые последовательности *x* и y. Формирование выполняется в два этапа. На первом этапе осуществляется нормировка исходных последовательностей *a*, *b* и *c*. В результате нормировки получаются последовательности a', b' и c'. Затем формируются последовательности *x* и y.

$$a'_i = \frac{a_i}{\sum_{i=1}^{n} a_i}, \quad b'_i = \frac{b_i}{\sum_{i=1}^{n} b_i}, \quad c'_i = \frac{c_i}{\sum_{i=1}^{n} c_i},$$

$$x_i = a'_i + b'_i.$$

$$y_i = b'_i + c'_i,$$

i = 0, 1, …, n-1

Функция main

```
int main()
{
    int n;    char c;
    float a[100], b[100], c[100], as[100], bs[100], cs[100], x[100], y[100];
    printf("vvesti razmer\n");
    scanf("%d",&n);
    vvod(a,n); vvod(b,n);vvod(c,n);
    strih(a,as,n); strih(b,bs,n); strih(c,cs,n);
    calc(as,bs,x,n); calc(bs,cs,y,n);
    vivod(x,n); vivod(y,n);
    scanf("%c\n",&c);
    return 0;
}
```

Функция, возвращающая сумму элементов массива

```
float sum(float mas[],int n)
{
    int i;
   float s=0;
    for(i=0; i<n; i++)
        s+=mas[i];
     return s;
}
```

Функция strih

```
void strih(float m[],float ms[],int n)
{
    int i;
   float s; s=sum(m,n);
    for(i=0; i<n; i++)
      ms[i]=m[i]/s;
}
```

Функция calc

- void calc(float m1[],float m2[],float mrez[],int n)
- {
- int i;
- for(i=0; i<n; i++)
  mrez[i]=m1[i]+m2[i];
- }

Прототипы функций

- **void vvod(float mas[],int n);**
- **void vivod(float mas[], int n);**
- **float sum(float mas[],int n);**
- **void strih(float m[],float ms[],int n);**
- **void calc(float m1[],float m2[],float mrez[],int n);**

Прототипы функций указываются в том случае, если функция не определена до первого её вызова!

```
14  void inputarr( float mas[][10],int n,int m)
15  {
16      int i,j;
17      printf("input array\n");
18      for(i=0; i<n; i++)
19       for(j=0;j<m;j++)
20       {
21              printf("mas[%d][%d]=", i,j);
22              scanf("%f", &mas[i][j]);
23       }
24  }
```

```
27
28    void outarr( float mas[][10],int n,int m)
29    {
30         int i,j;
31         printf("input array\n");
32        for(i=0; i<n; i++)
33        {
34         for(j=0;j<m;j++)
35                   printf("%5.3f    ", mas[i][j]);
36         printf("\n");
37         }
38    }
```

```
40  void calcarr( float mas[][10],int n,int m)
41  {
42      int i,j;
43     for(i=0; i<n; i++)
44      {
45              for(j=0;j<m;j++)
46              {
47
48              mas[i][j]*=2;
49              }
50      }
51  }
```

```
53  int main(int argc, char** argv)
54  {
55      float a[10][10];
56      float s[10];
57      int n,m;
58      printf("input size of array n:\n");
59      scanf("%d",&n);
60      printf("input size of array m:\n");
61      scanf("%d",&m);
62      inputarr( a,n,m);
63      calcarr( a,n,m);
64      printf("Rezultat:\n");
65      outarr(a,n,m);
66      scanf("%d",&n);
67      return 0;
68  }
```

Область действия (видимость) переменных

```
#include<iostream.h>
void main(void)
{   int a=10;

    {
    int a=5;
    cout<<a<<endl;
    }
    cout<<a<<endl;
}
```

Переменная видна в том блоке программы, в котором она определена, и во вложенных блоках. Локальное имя преобладает над глобальным.

Автоматические и статические преременные

```
#include<iostream.h>
int calc()
{
 int a=0;
 a++;
 return(a);
}
void main(void)
{

      int x;
       x=calc();
       cout<<"x="<<x<<endl;
       x=calc();
       cout<<"x="<<x<<endl;
       cin>>x;
}
```

Автоматическая переменная создается каждый раз при вызове функции, а статическая один раз.
На экране увидим
X=1
X=1

```
#include<iostream.h>
int calc()
{
 static int a=0;
 a++;
 return(a);
}
void main(void)
{
	int x;
	x=calc();
	cout<<"x="<<x<<endl;
	x=calc();
	cout<<"x="<<x<<endl;
	cin>>x;
}
```

Автоматическая переменная создается каждый раз при вызове функции, а статическая один раз.
На экране увидим
X=1
X=2

Динамические массивы

```
#include <iostream.h>

void inputarr(int *inarr, int n, char arrname[])
{
   int i;

   cout << "Input the " << n << " digits for array " <<
   arrname << ":\n";
   for (i=0; i<n; i++) cin >> *(inarr+i);
}
```

```
void outputarr(int *outarr, int n, char
   arrname[])
{
   int i;

   for (i=0; i<n; i++)
     cout << arrname << "[" << i << "]=" <<
   *(outarr+i) <<  "\n";
}
```

```
void createoutarr(int arr1[], int arr2[], int
   outarr[], int n)
{
   int i;

   for (i=0; i<n; i++) outarr[i] = arr1[i] - arr2[i];

}
```

```
void main()
{
    int *x,*y,*z,*xy,*xz,*yz;
    int Size;
     cout<<"Enter size of array ";
     cin>>Size;
     x =new int[Size];
     y =new int[Size];
     z =new int[Size];
     xy=new int[Size];
     xz=new int[Size];
     yz=new int[Size];
```

```
inputarr(x, Size, "x");
inputarr(y, Size, "y");
inputarr(z, Size, "z");

createoutarr(x, y, xy, Size);
createoutarr(x, z, xz, Size);
createoutarr(y, z, yz, Size);

outputarr(xy, Size, "xy");
outputarr(xz, Size, "xz");
outputarr(yz, Size, "yz");
```

Освобождение динамической памяти

```
    delete  [] x;
    delete [] y;
    delete [] z;
    delete [] xy;
    delete [] xz;
    delete [] yz;
}
```

Передача имен функций в качестве параметров

Функцию можно вызвать через указатель на нее. Для этого объявляется указатель соответствующего типа и ему с помощью операции взятия адреса присваивается адрес функции:

```
void f(int a ){ /* ... */ }//определение функции
void (*pf)(int);//указатель на функцию
...
pf = &f;/* указателю присваивается адрес функции
(можно написать pf = f;) */
pf(10);/* функция f вызывается через указатель pf
(можно написать (*pf)(10) ) */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int f(int a ){ return a; }
int (*pf)(int);
int main(void)
{
 pf = &f;
 printf("%d\n",pf(10));
 pf=f;
 printf("%d\n",pf(10));
   return 0;
}
```

На экране
10
10

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

int f(int a ){ return a; }
int (*pf)(int);
void fun(int (*pf)(int) , int x)
{
  printf("%d\n",pf(x));

}
int main(void)
{
  pf = &f;
  printf("%d\n",pf(10));
  pf=f;
  printf("%d\n",pf(10));
  fun(f,20);

   return 0;
}
```

На экране
10
10
20

Параметры со значениями по умолчанию

- Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и могут опускаться при вызове функции :
- int f(int a, int b = 0);
- void f1(int, int = 100, char* = 0);
- void err(int errValue = errno);//errno — глобальная переменная
- f(100); f(a, 1);// варианты вызова функции f
- f1(a); f1(a, 10); f1(a, 10, "Vasia");// варианты вызова функции f1
- f1(a,,"Vasia") // неверно!

Функция Си/ C++ - qsort

- //void qsort(void *base, size_t nelem,
- //size_t width, int (*fcmp)(const void *, const void *));

- Описание.
- Функция qsort выполняет алгоритм быстрой сортировки, чтобы
- отсортировать массив из nelem элементов, каждый элемент размером
- width байт. Аргумент base является указателем на базу массива,
- который нужно отсортировать. Функция qsort перезаписывает этот
- массив с отсортированными элементами.
- Аргумент fcmp является указателем на функцию, поставляемую пользователем, которая сравнивает два элемента массива и
- возвращает значение, определяющее их отношение.
- Функция qsort может вызывать процедуру fcmp один или
- несколько раз в процессе сортировки, передавая при каждом вызове
- указатели на два элемента массива. Процедура должна сравнивать
- элементы, а затем возвращать одно из следующих значений:
- Значение Его смысл
- меньше 0 element 1 меньше element 2
  0 element 1 равен element 2
- больше 0 element 1 больше element 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int sort_function( const void *a, const void *b);

char list[5][4] = { "cat", "car", "cab", "cap", "can" };

int main(void)
{
   int  x;

   qsort((void *)list, 5, sizeof(list[0]), sort_function);
   for (x = 0; x < 5; x++)
      printf("%s\n", list[x]);
   return 0;
}

int sort_function( const void *a, const void *b)
{
   return( strcmp((char *)a,(char *)b) );
}
```

Массив указателей на функции

Массив указателей на функции определяется точно также, как и обычный массив – с помощью квадратных скобок после имени:

```
float (*menu[4])(float, float);
```

```
6  #define ERROR_DIV_BY_ZERO -2
7  #define EPSILON 0.000001f
8
9  float doSum(float a, float b) {
10     return a + b;
11 }
12
13 float doSub(float a, float b) {
14     return a - b;
15 }
16
17 float doMul(float a, float b) {
18     return a * b;
19 }
20
21 float doDiv(float a, float b) {
22     if (fabs(b) <= EPSILON) {
23         exit(ERROR_DIV_BY_ZERO);
24     }
25     return a / b;
26 }
27
```

```
28  void main() {
29      float (*menu[4])(float, float);
30      int op;
31      float a, b;
32      menu[0] = doSum;
33      menu[1] = doSub;
34      menu[2] = doMul;
35      menu[3] = doDiv;
36      printf("enter a: ");
37      scanf("%f", &a);
38      printf("enter b: ");
39      scanf("%f", &b);
40      printf("enter operation [0 - add, 1 - sub, 2 - mul, 3 - div]");
41      scanf("%d", &op);
42      if (op >= 0 && op < 4) {
43          printf("%.6f", menu[op](a, b));
44      }
45      getch();
46  }
```

Точно также можно было создать массив динамически

```
void main() {
    float (**menu)(float, float) = NULL;
    int op;
    float a, b;
    menu = (float(**)(float, float)) malloc(4*sizeof(float(*)(float, float)));
    menu[0] = doSum;
    menu[1] = doSub;
    menu[2] = doMul;
    menu[3] = doDiv;
    printf("enter a: ");
    scanf("%f", &a);
    printf("enter b: ");
    scanf("%f", &b);
    printf("enter operation [0 - add, 1 - sub, 2 - mul, 3 - div]");
    scanf("%d", &op);
    if (op >= 0 && op < 4) {
        printf("%.6f", menu[op](a, b));
    }
    free(menu);
    getch();
}
```

Часто указатели на функцию становятся громоздкими. Работу с ними можно упростить, если ввести новый тип. Предыдущий пример можно переписать так

```
8
9   typedef float (*operation)(float, float);

27  void main() {
28      operation *menu = NULL;
29      int op;
30      float a, b;
31      menu = (operation*) malloc(4*sizeof(operation));
32      menu[0] = doSum;
33      menu[1] = doSub;
34      menu[2] = doMul;
35      menu[3] = doDiv;
```

Ещё один пример: функция any возвращает 1, если в переданном массиве содержится хотя бы один элемент, удовлетворяющий условию pred и 0 в противном случае.

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

typedef int (*Predicat)(void*);

int isBetweenInt(void* a) {
    return *((int*) a) > 10 && *((int*) a) < 12;
}

int isBetweenDouble(void* a) {
    return *((double*) a) > 10.0 && *((double*) a) < 12.0;
}

int any(void* arr, unsigned num, size_t size, Predicat pred) {
    unsigned i;
    char* ptr = (char*) arr;
    for (i = 0; i < num; i++) {
        if (pred(ptr + i*size)) {
            return 1;
        }
    }
    return 0;
}

void main() {
    int a[10] = {1, 1, 2, 2, 3, 0, 1, 2, 1, 3};
    double b[10] = {1, 2, 11, 2, 3, 4, 5, 6, 7, 10};

    printf("has 'a' any value > 10 and < 12? %d\n", any(a, 10, sizeof(int), isBetweenInt));
    printf("has 'b' any value > 10 and < 12? %d", any(b, 10, sizeof(double), isBetweenDouble));

    getch();
}
```