ЛИТЕРАТУРА

1. Демидович Е. Основы алгоритмизации и программирования. Язык Си: учебное пособие – СПб.: БХВ – Петербург, 2006. – 440с.
2. Жешке Р. Толковый словарь стандарта языка Си. – СПб.: Питер, 1994. – 221с.
3. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 1992. – 272с.
4. Кочан С. Программирование на языке С, 3-е издание: Пер. с англ. – М.: ООО "И. Д. Вильямс", 2007. – 496с.
5. Подбельский В., Фомин С. Программирование на языке Си: учебное пособие. 2-е доп. Изд. – М: Финансы и статистика, 2001. – 2001. – 600с.
6. Прата С. Язык программирования С. Лекции и упражнения, 5-е издание.: Пер. с англ. – М.: Издательский дом "Вильямс", 2006. – 960с.
7. Шилдт Г. Полный справочник по С. 4-е издание.: Пер. с англ. – М.: Издательский дом "Вильямс", 2002. – 704с.
8. Харбисон С., Стил Г. Язык программирования С.: Пер. с англ. – М: ООО Бином Пресс, 2004. – 528с.

Язык программирования СИ

Си (англ. *C*) — компилируемый статически типизированный язык программирования общего назначения, разработанный в 1969—1973 годах сотрудником Bell Labs Деннисом Ритчи как развитие языка Би. Первоначально был разработан для реализации операционной системы UNIX, но впоследствии былперенесён на множество других платформ. Согласно дизайну языка, его конструкции близко сопоставляются типичным машинным инструкциям, благодаря чему он нашёл применение в проектах, для которых был свойственен язык ассемблера, в том числе как в операционных системах, так и в различном прикладном программном обеспечении для множества устройств — от суперкомпьютеров до встраиваемых систем. Язык программирования Си оказал существенное влияние на развитие индустрии программного обеспечения, а его синтаксис стал основой для таких языков программирования, как C++, C#, Java и Objective-C.

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.

C11

- 8 декабря 2011 опубликован новый стандарт для языка Си (ISO/IEC 9899:2011). Основные изменения:
- поддержка многопоточности;
- улучшенная поддержка Юникода;
- обобщённые макросы (type-generic expressions, позволяют статичную перегрузку);
- анонимные структуры и объединения (упрощают обращение ко вложенным конструкциям);
- управление выравниванием объектов;
- статичные утверждения (static assertions);
- удаление опасной функции gets (в пользу безопасной gets_s);
- функция quick_exit;
- спецификатор функции _Noreturn;
- новый режим эксклюзивного открытия файла.

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	$1.7\ 10^{-308}$ . . . $1.7\ 10^{+308}$

Тип данных	Размер	Минимальный диапазон значений	Первое появление
`signed char`	минимум 8 бит	от −127[10] (= -(2^7−1)) до 127	K&R C
`unsigned char`	минимум 8 бит	от 0 до 255 (=2^8−1)	K&R C
`char`	минимум 8 бит	от −127 до 127 или от 0 до 255 в зависимости от компилятора	K&R C
`short int`	минимум 16 бит	от −32,767 (= -(2^15−1)) до 32,767	K&R C
`unsigned short int`	минимум 16 бит	от 0 до 65,535 (= 2^16−1)	K&R C
`int`	минимум 16 бит	от −32,767 до 32,767	K&R C
`unsigned int`	минимум 16 бит	от 0 до 65,535 (= 2^16−1)	K&R C
`long int`	минимум 32 бита	от −2,147,483,647 до 2,147,483,647	K&R C
`unsigned long int`	минимум 32 бита	от 0 до 4,294,967,295 (= 2^32−1)	K&R C
`long long int`	минимум 64 бита	от −9,223,372,036,854,775,807 до 9,223,372,036,854,775,807	C99
`unsigned long long int`	минимум 64 бита	от 0 до 18,446,744,073,709,551,615 (= 2^64−1)	C99
`int8_t`	8 бит	от −127 до 127	C99
`uint8_t`	8 бит	от 0 до 255 (=2^8−1)	C99
`int16_t`	16 бит	от −32,767 до 32,767	C99
`uint16_t`	16 бит	от 0 до 65,535 (= 2^16−1)	C99
`int32_t`	32 бита	от −2,147,483,647 до 2,147,483,647	C99
`uin32_t`	32 бита	от 0 до 4,294,967,295 (= 2^32−1)	C99
`int64_t`	64 бита	от −9,223,372,036,854,775,807 до 9,223,372,036,854,775,807	C99
`uint64_t`	64 бита	от 0 до 18,446,744,073,709,551,615 (= 2^64−1)	C99

Типы `int_leastN_t`, `uint_leastN_t`, `int_fastN_t` и `uint_fastN_t` (`N` = 8, 16, 32 или 64), введенные стандартом C99, размером и диапазоном совпадают с соответствующими типами `char`, `short`, `int` и `long`.

В таблице приведён минимальный диапазон значений согласно стандарту языка. Компиляторы языка Си могут расширять диапазон значений.

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(“a=%6.3f\n b=%f\n c=%f\n",a,b,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);
}
```

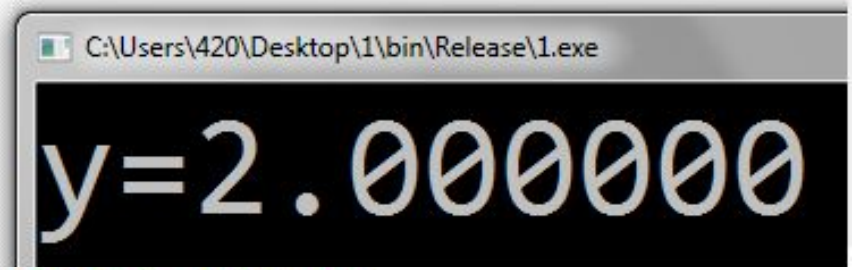
Вывод значений нескольких переменных

- //*********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;
}
```

```
1  #include <iostream>
2  #include <stdio.h>
3  int main()
4  {
5      float y;
6      y=5/2;
7      printf("y=%f\n",y);
8      return 0;
9  }
10
```



Целочисленное деление (оба операнда — целые числа)

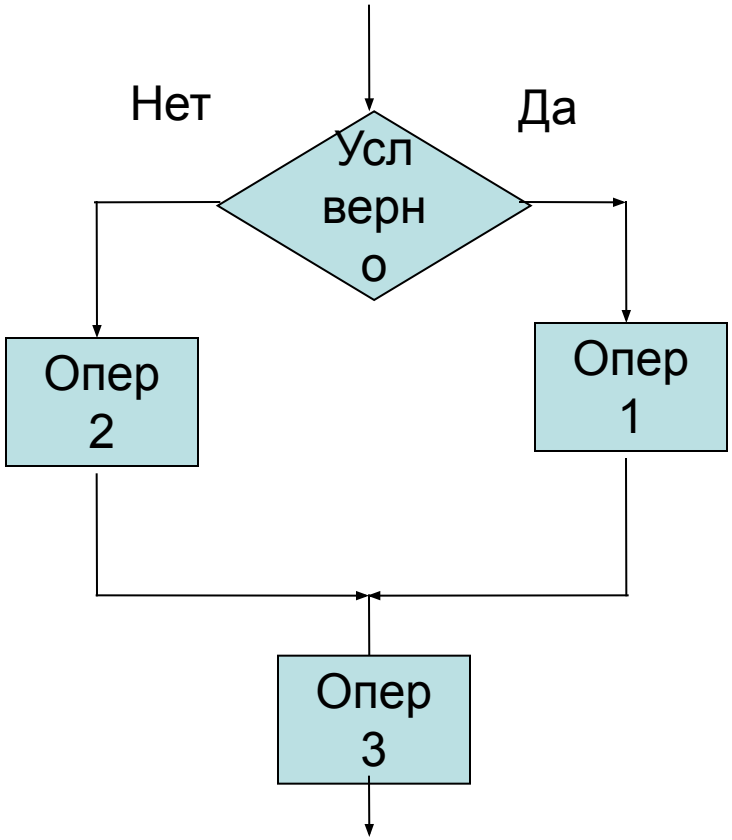
```
1  #include <iostream>
2  #include <stdio.h>
3  int main()
4  {
5      float y;
6      y=5./2;
7      printf("y=%f\n",y);
8      return 0;
9  }
10
```

Деление не целочисленное (операнд 5. - вещественное число)

Условный оператор if

Полная форма

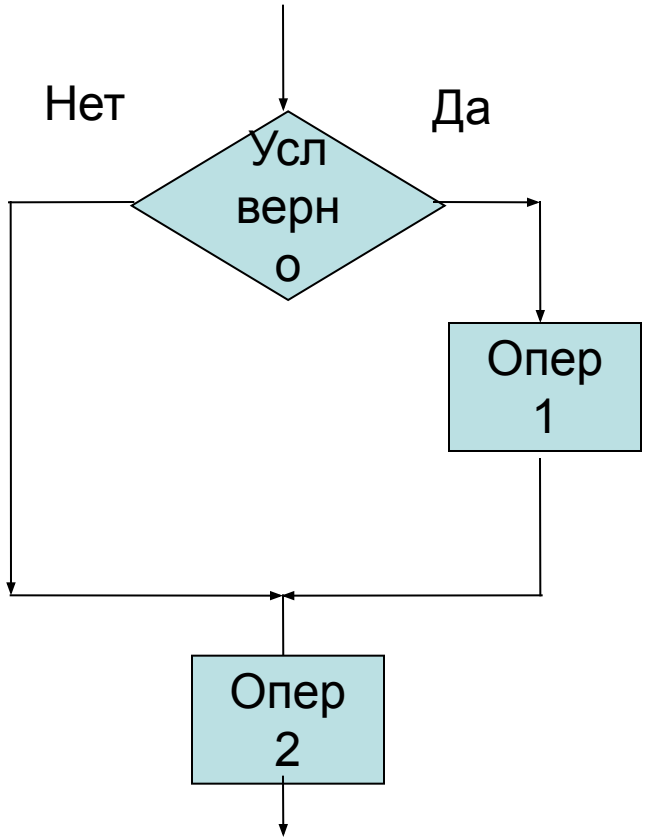
```
if(условие)
Опер1;
else Опер 2;
Опер 3;
```



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 инкремента и декремента

- 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.

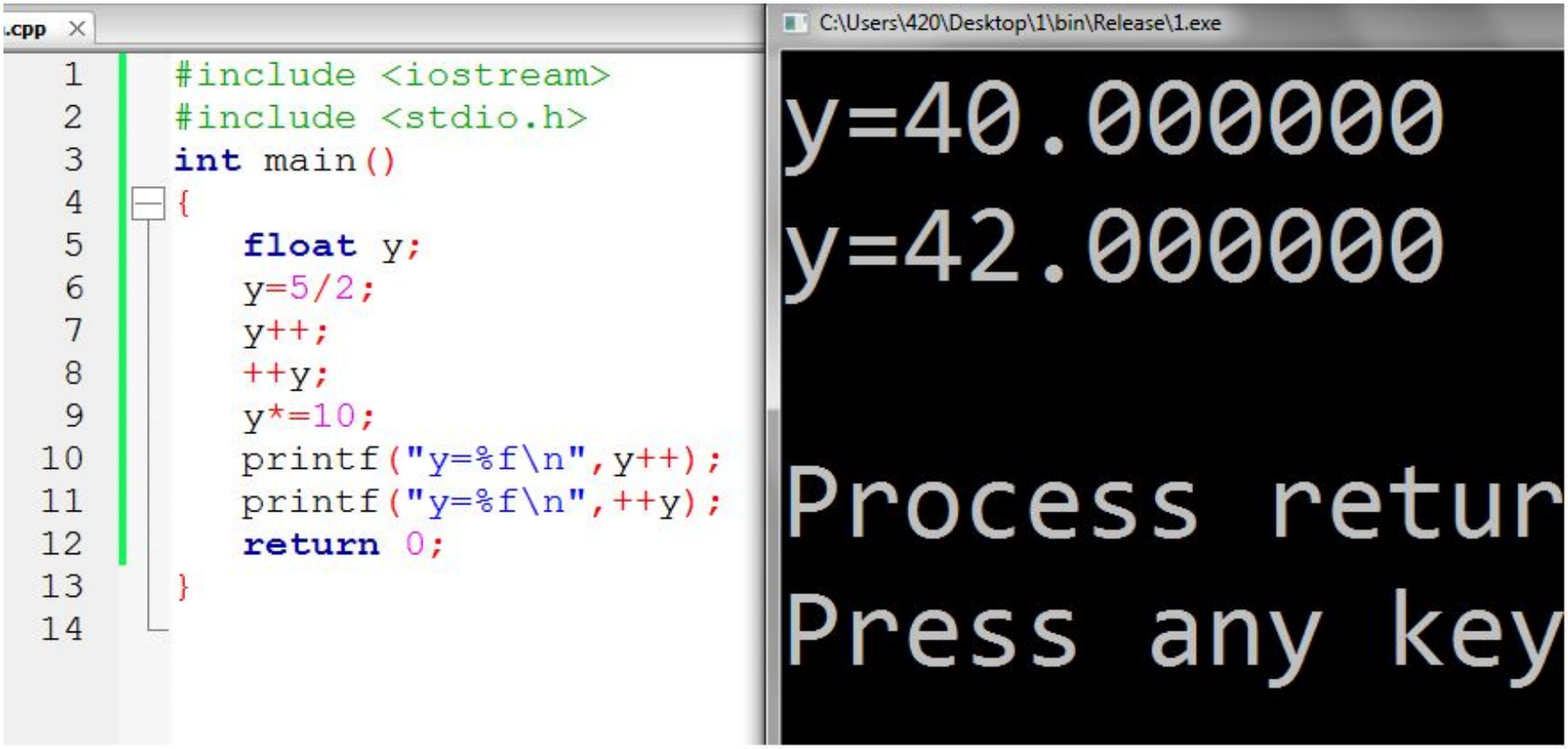
- Операндом может быть именующее выражение, например, имя переменной.
- Следующие три строки увеличивают переменную x на 1:
- x = x + 1;
- ++x;
- x++;

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;


.cpp
1 #include <iostream>
2 #include <stdio.h>
3 int main()
4 {
5 float y;
6 y=5/2;
7 y++;
8 ++y;
9 y*=10;
10 printf("y=%f\n",y++);
11 printf("y=%f\n",++y);
12 return 0;
13 }
14
C:\Users\420\Desktop\1\bin\Release\1.exe
y=40.000000
y=42.000000
Process retur
Press any key

```cpp
#include <iostream>
#include <stdio.h>
int main()
{
    float y;
    y=5/2;
    y++;
    ++y;
    y*=10;
    printf("1234567890\n");
    printf("%10.5f\n",++y);
    return 0;
}
```

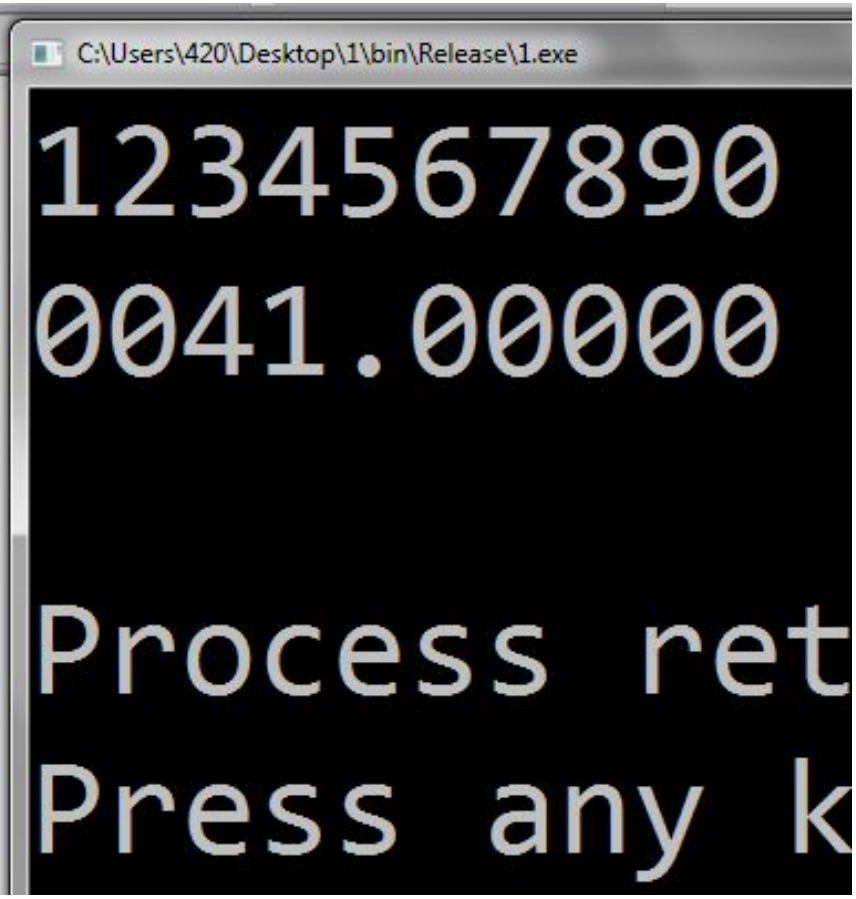
```
C:\Users\420\Desktop\1\bin\Release\1.exe
1234567890
  41.00000

Process retu
```

```cpp
#include <iostream>
#include <stdio.h>
int main()
{
    float y;
    y=5/2;
    y++;
    ++y;
    y*=10;
    printf("1234567890\n");
    printf("%-10.5f\n",++y);
    return 0;
}

```

```
C:\Users\420\Desktop\1\bin\Release\1.exe
1234567890
41.00000

Process ret
Press any k
```

Заполнение лидирующими нулями

```
1  #include <iostream>
2  #include <stdio.h>
3  int main()
4  {
5      float y;
6      y=5/2;
7      y++;
8      ++y;
9      y*=10;
10     printf("1234567890\n");
11     printf("%010.5f\n",++y);
12     return 0;
13 }
14
```



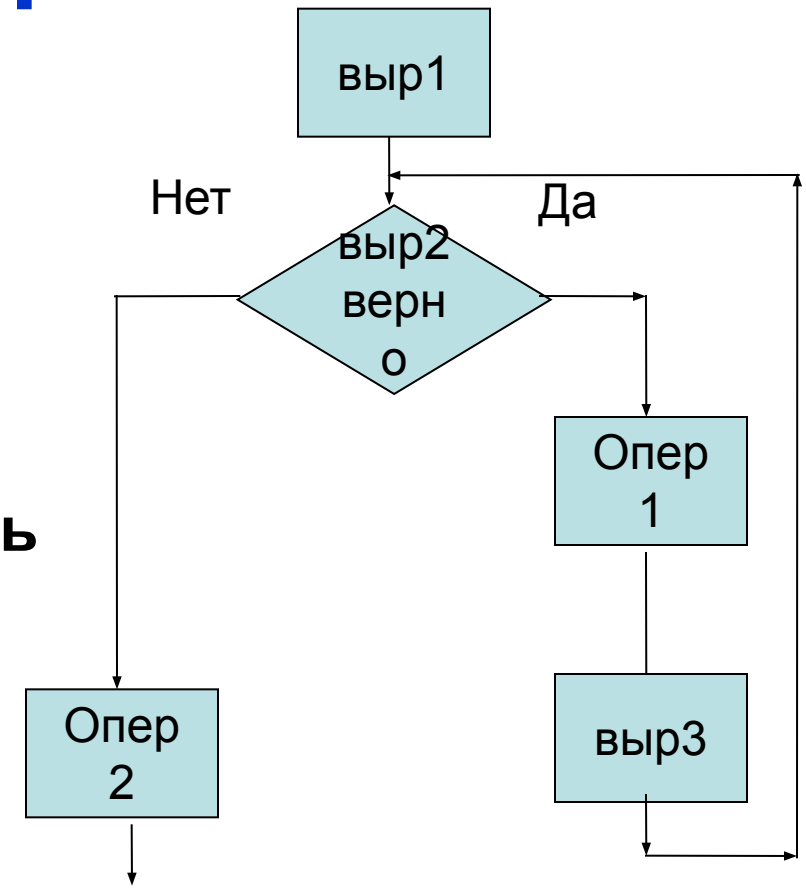
Операторы циклов

- for
- while
- do …..while

Оператор for

for(выр1; выр2;выр3)
Опер1;
Опер 2;

выр1-инициализационная часть
выр2-проверочная
выр3-послецикловая



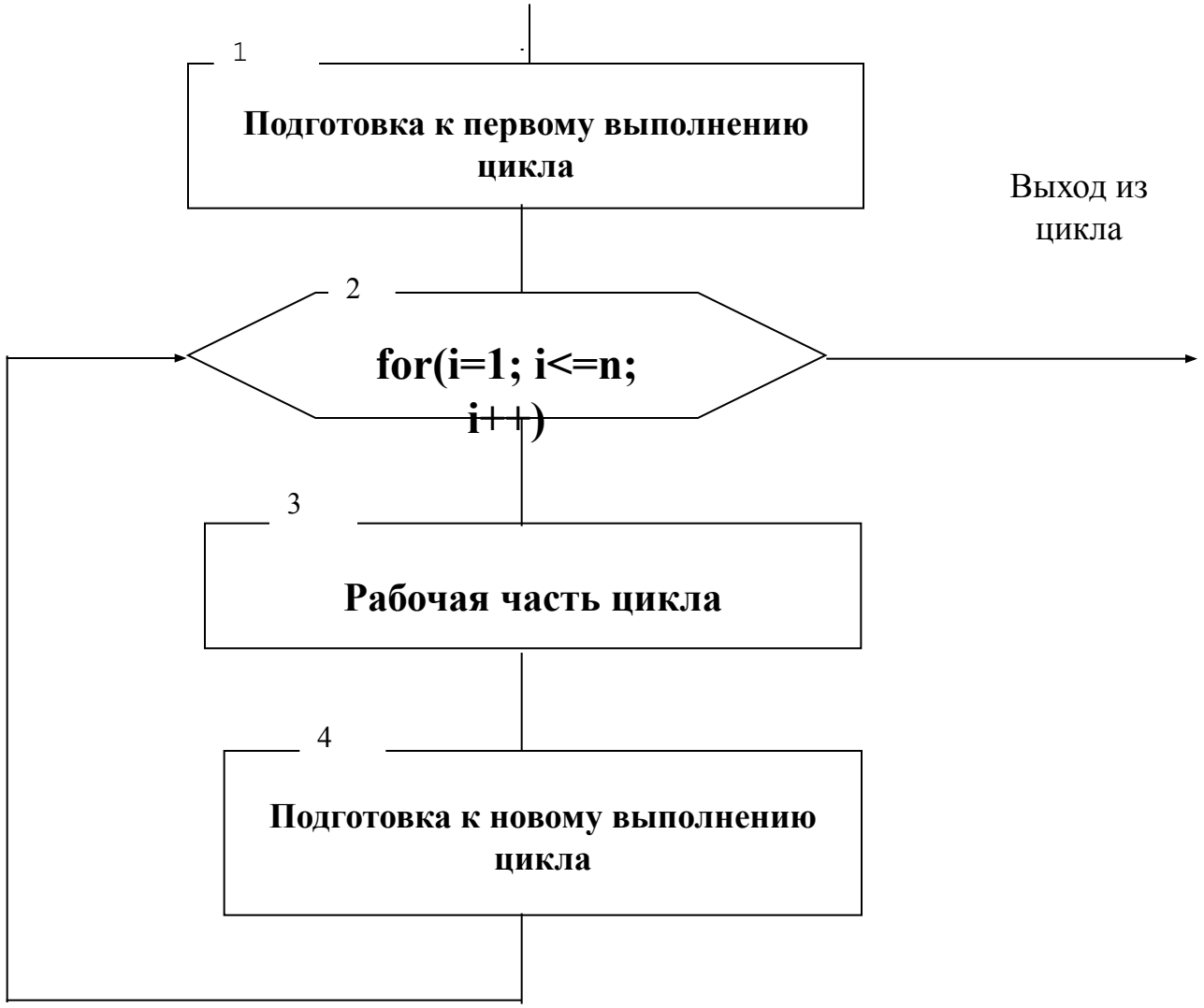
Пример

```
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

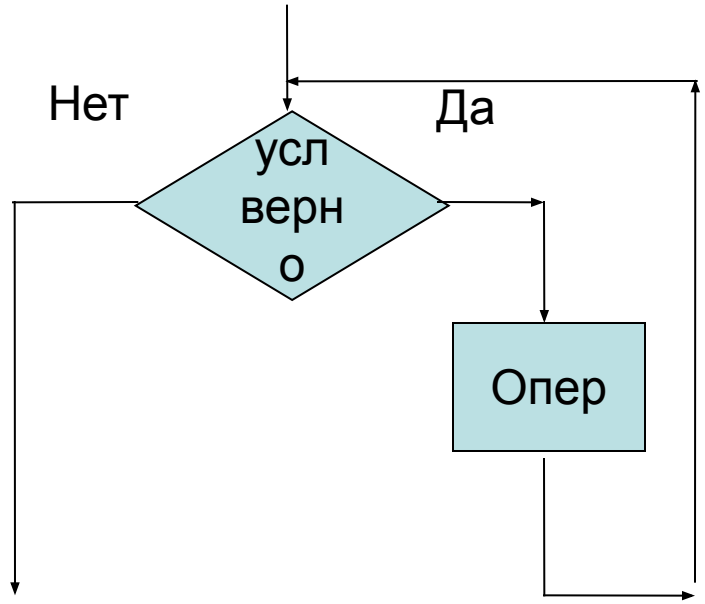


Обобщенная схема алгоритма

Оператор while

while(условие)
Опер;

Цикл с предусловием



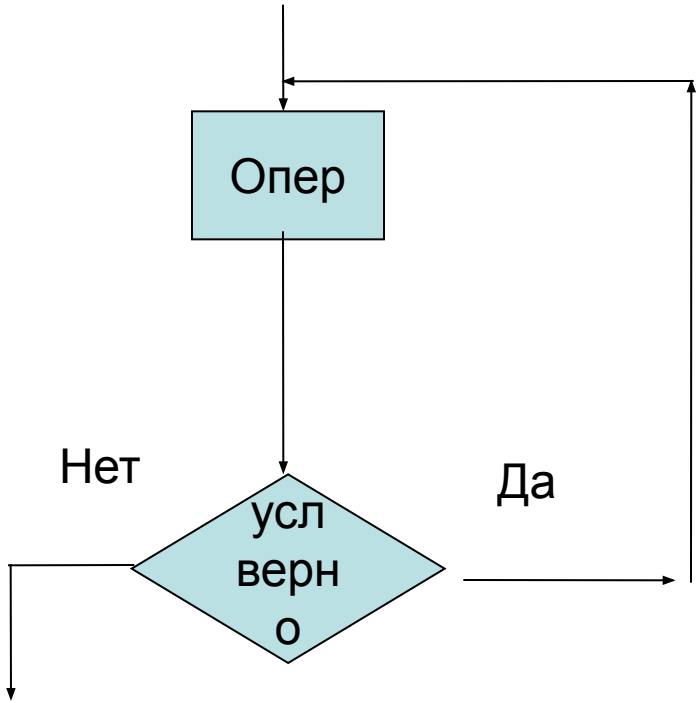
Пример

```
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;
```

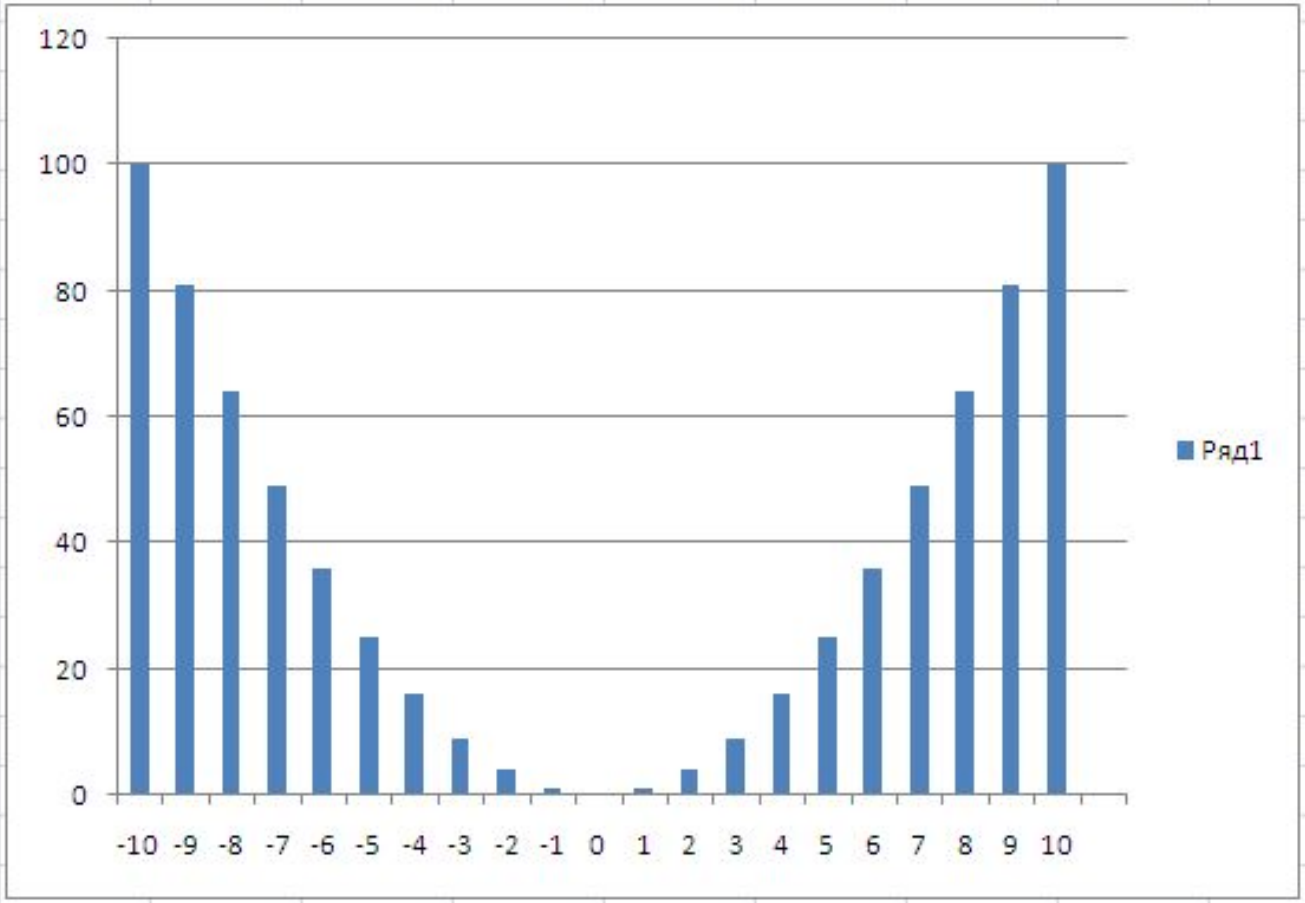
```
int i;
i=1;
while( i<=5)
{
     cout<<i;
     i++;
}
```

```
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;
 }
```

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

int main()
{
    float arr[20][20];
    int n,m,i,j;
    int k;
    printf("vvedite n,m");
    scanf("%d%d",&n,&m);
    //input array
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
           printf("arr[%d][%d]=",i,j);
           scanf("%f",&arr[i][j]);
        }
    }
    //calk
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
           arr[i][j]*=2;
        }-
    }
    //output arr
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
           printf("%5.2f   ",arr[i][j]);
        }
        printf("\n");
    }
return 0;
}
```

```
D:\CodeBloks\mas2d\bin\Debug\mas2d.ex
vvedite n,m3 4
arr[0][0]=1
arr[0][1]=1
arr[0][2]=1
arr[0][3]=1
arr[1][0]=2
arr[1][1]=2
arr[1][2]=2
arr[1][3]=2
arr[2][0]=3
arr[2][1]=3
arr[2][2]=3
arr[2][3]=3
 2.00    2.00    2.00    2.00
 4.00    4.00    4.00    4.00
 6.00    6.00    6.00    6.00
```

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.

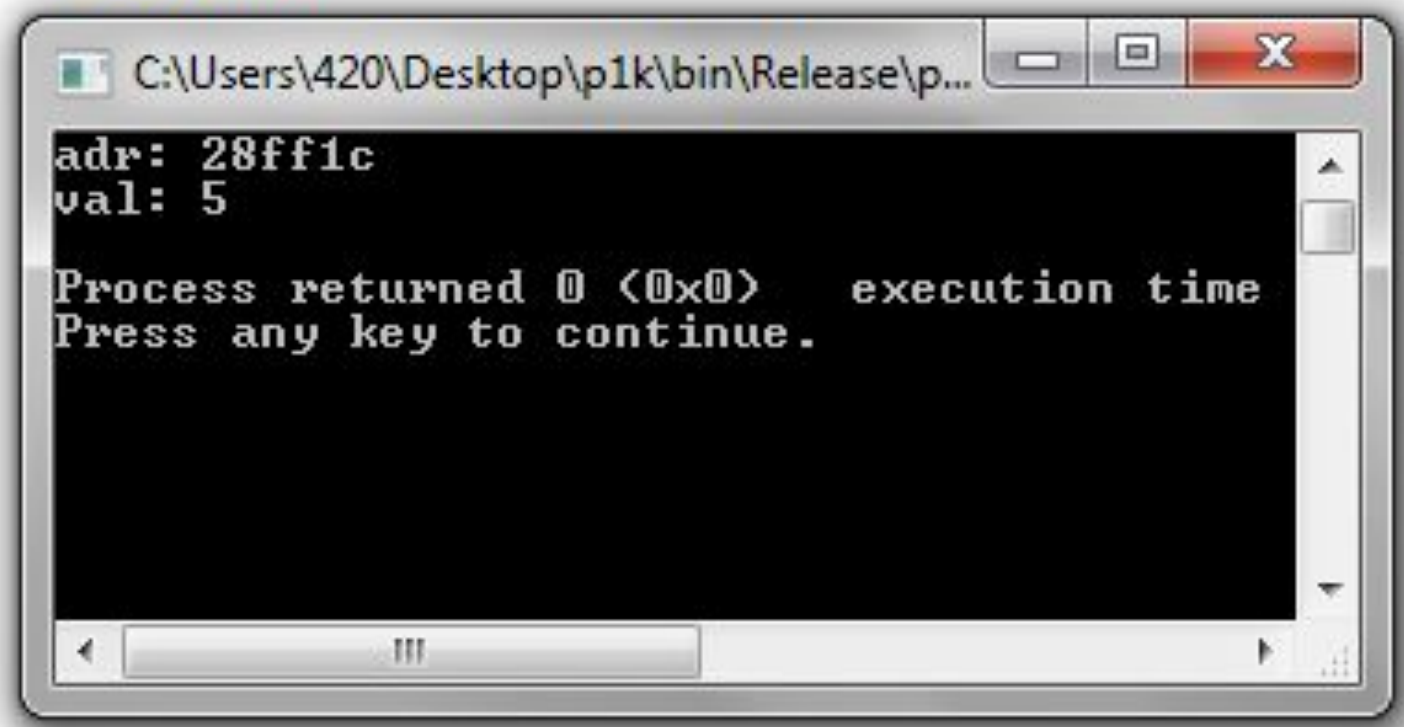
- Теперь сделаем так, чтобы указатели на что-нибудь указывали:

- **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**

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

int main()
{
    int some_number=5;
    int *pNumberOne = &some_number;
    printf("adr: %x\n",pNumberOne);
    printf("val: %d\n",*pNumberOne);
    return 0;
}
```



```
#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

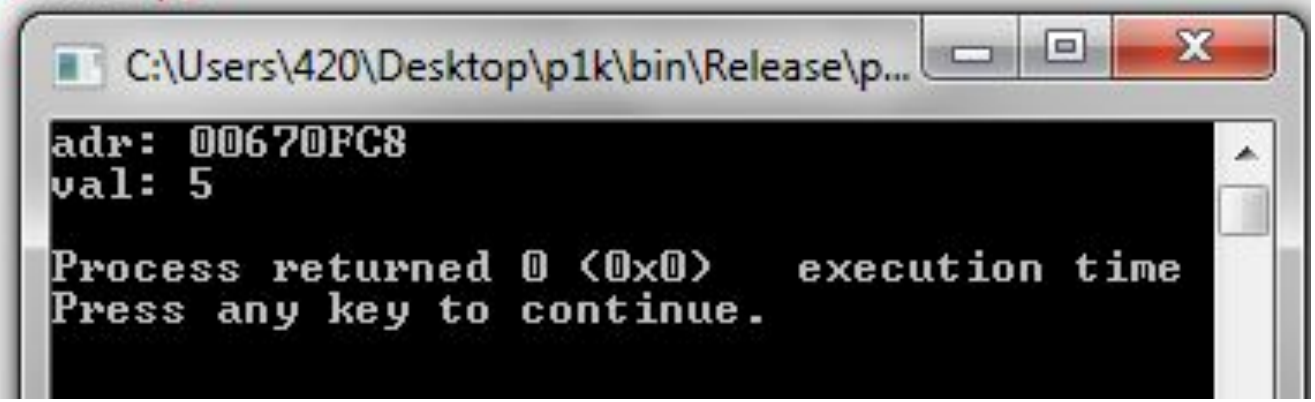
- Динамическая память позволяет выделять/освобождать память во время работы программы. Код ниже демонстрирует, как выделить память для целого числа:
- **int *pNumber;**
- **pNumber = new int;**
- Первая строчка объявляет указатель pNumber. Вторая строчка выделяет память для целого числа (int) и указывает 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;**

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int main()
{
    int *pNumberOne = malloc(sizeof(int));
    *pNumberOne=5;
    printf("adr: %p\n",pNumberOne);
    printf("val: %d\n",*pNumberOne);
    free(pNumberOne);
    return 0;
}
```



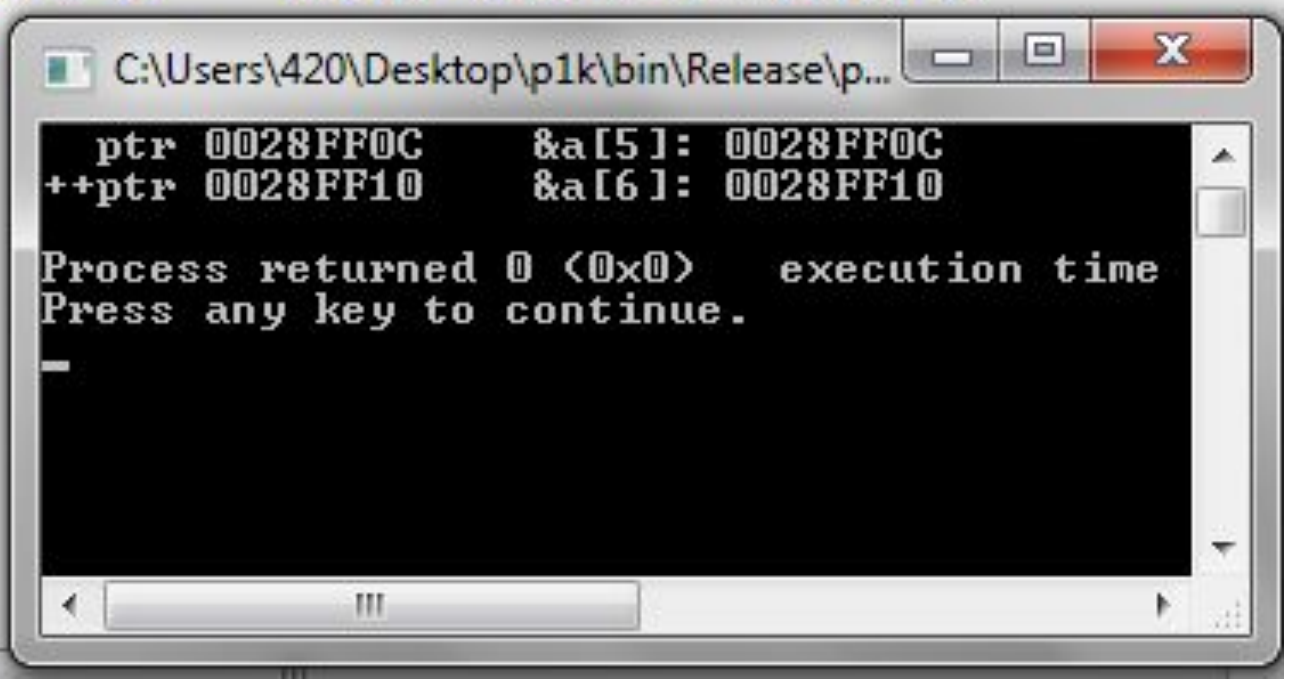
Операции с указателями

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] */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int main()
{
    int *ptr, a[10];
    ptr=&a[5];
    ++ptr; /* равно адресу элемента a[6] */
    --ptr; /* равно адресу элемента a[5] */
    printf("  ptr %p     &a[5]: %p\n",ptr,&a[5]);
    printf("++ptr %p     &a[6]: %p\n",++ptr,&a[6]);
    return 0;
}
```



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] */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int main()
{
    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] */
    printf("  ptr1 %p     &a[6]: %p\n",ptr1,&a[6]);
    printf("  ptr2 %p     &a[4]: %p\n",ptr2,&a[4]);
    return 0;
}
```

C:\Users\420\Desktop\p1k\bin\Release\p...

```
  ptr1 0028FF10     &a[6]: 0028FF10
  ptr2 0028FF08     &a[4]: 0028FF08

Process returned 0 (0x0)   execution t
Press any key to continue.
```

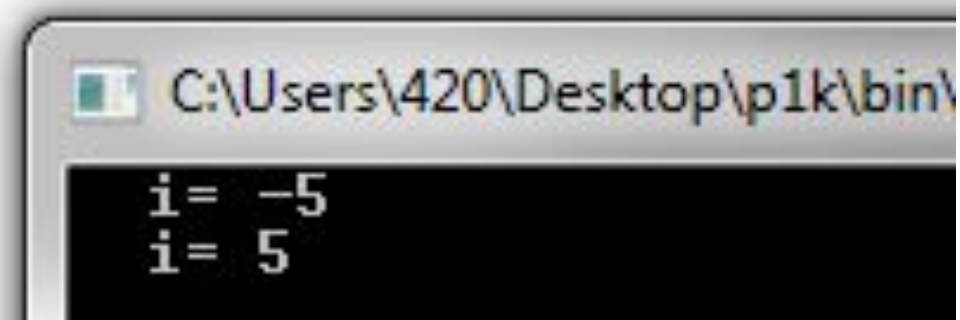
Операции с указателями

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 */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int main()
{
   int *ptr1, *ptr2, a[10];
   int i;
   ptr1=a+4;
   ptr2=a+9;
   i=ptr1-ptr2; /* равно -5 */
   printf("  i= %d\n",i);
   i=ptr2-ptr1; /* равно  5 */
   printf("  i= %d\n",i);
   return 0;
}
```



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!

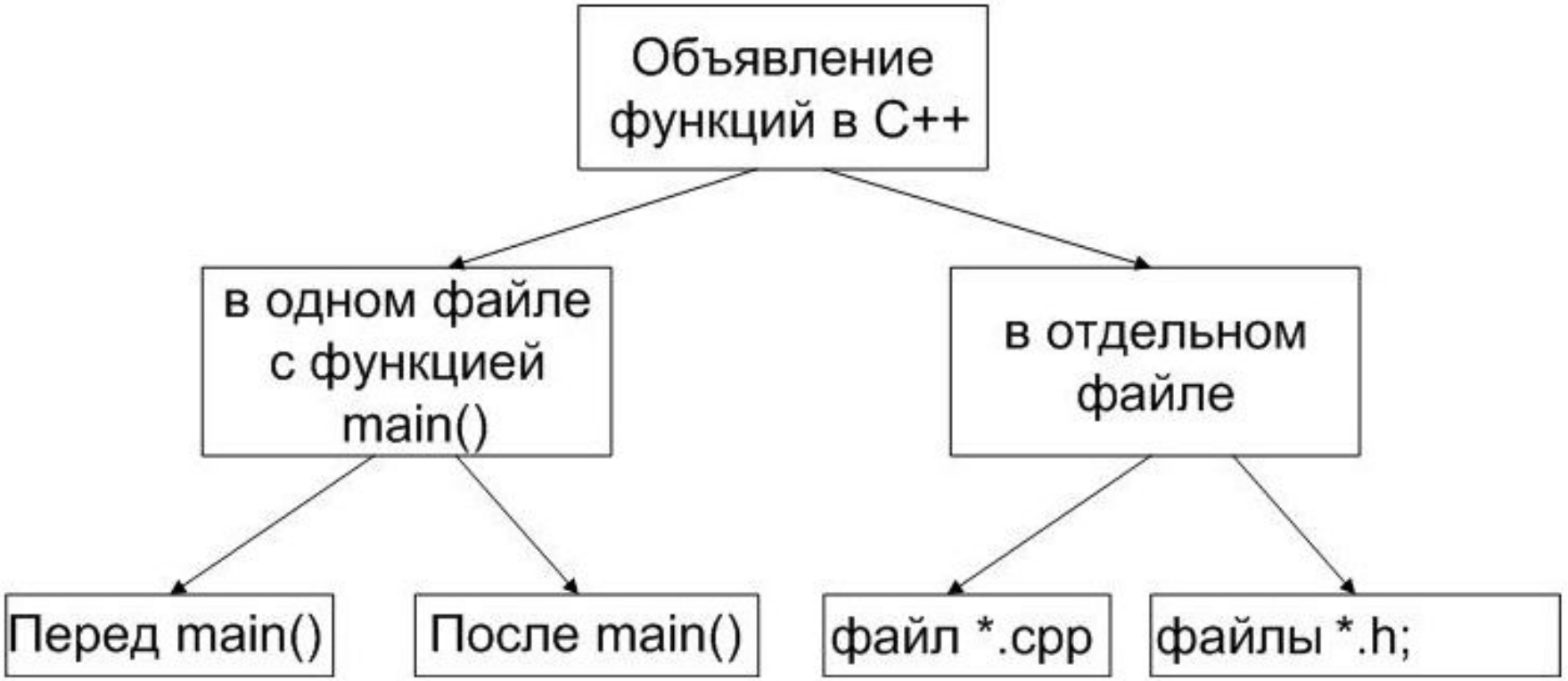
```
14  void inputarr( float mas[][10],int n,int m)
15  {
16      int i,j;
17      printf("input array\n");
18      for(i=0; i<n; i++)
19       for(j=0;j<m;j++)
20       {
21              printf("mas[%d][%d]=", i,j);
22              scanf("%f", &mas[i][j]);
23       }
24  }
```

```
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  }
```

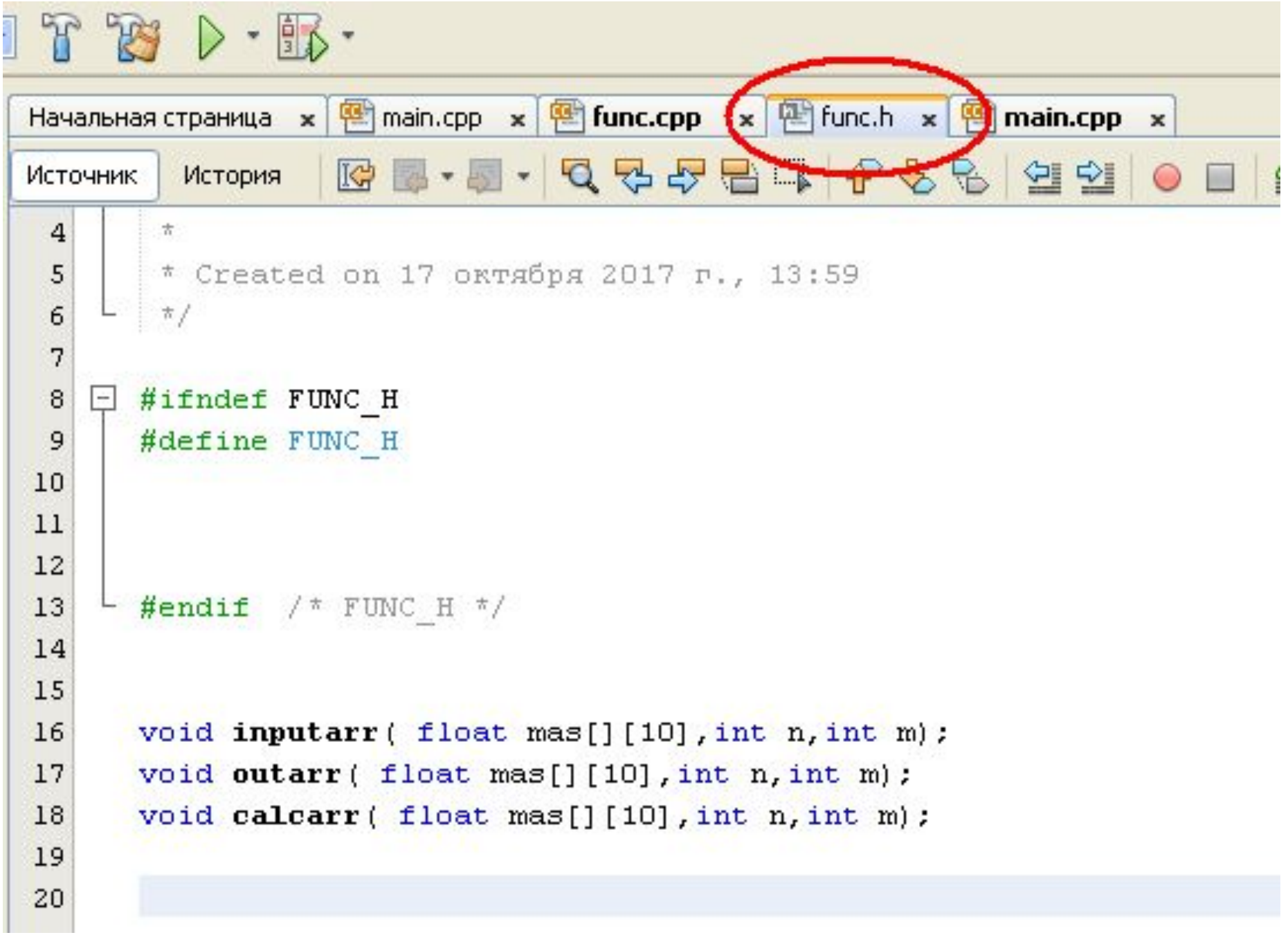
Проекты

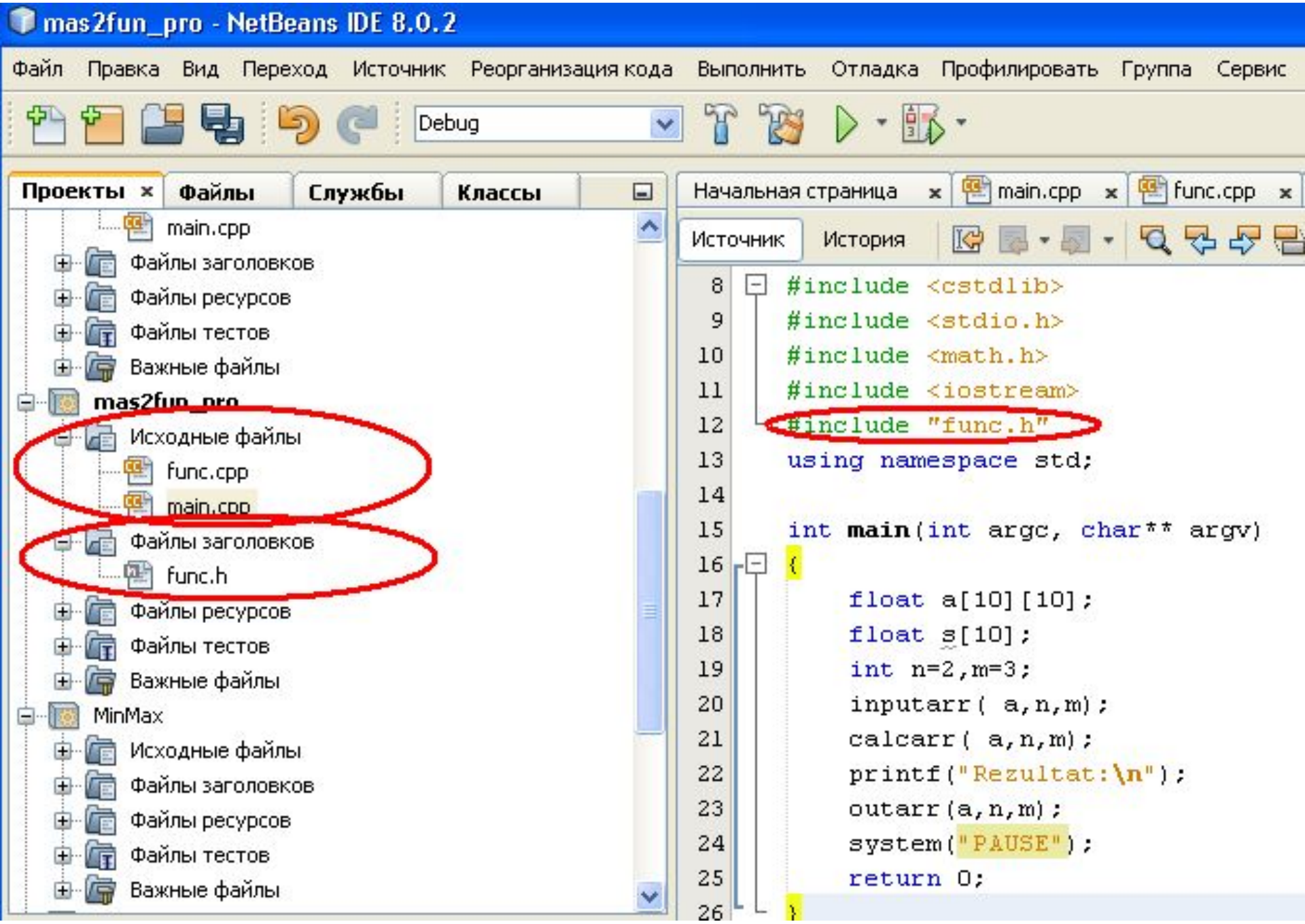

Объявление функций в C++
в одном файле с функцией main()
в отдельном файле
Перед main()
После main()
файл *.cpp
файлы *.h;

Начальная страница | main.cpp | func.cpp | func.h | main.cpp

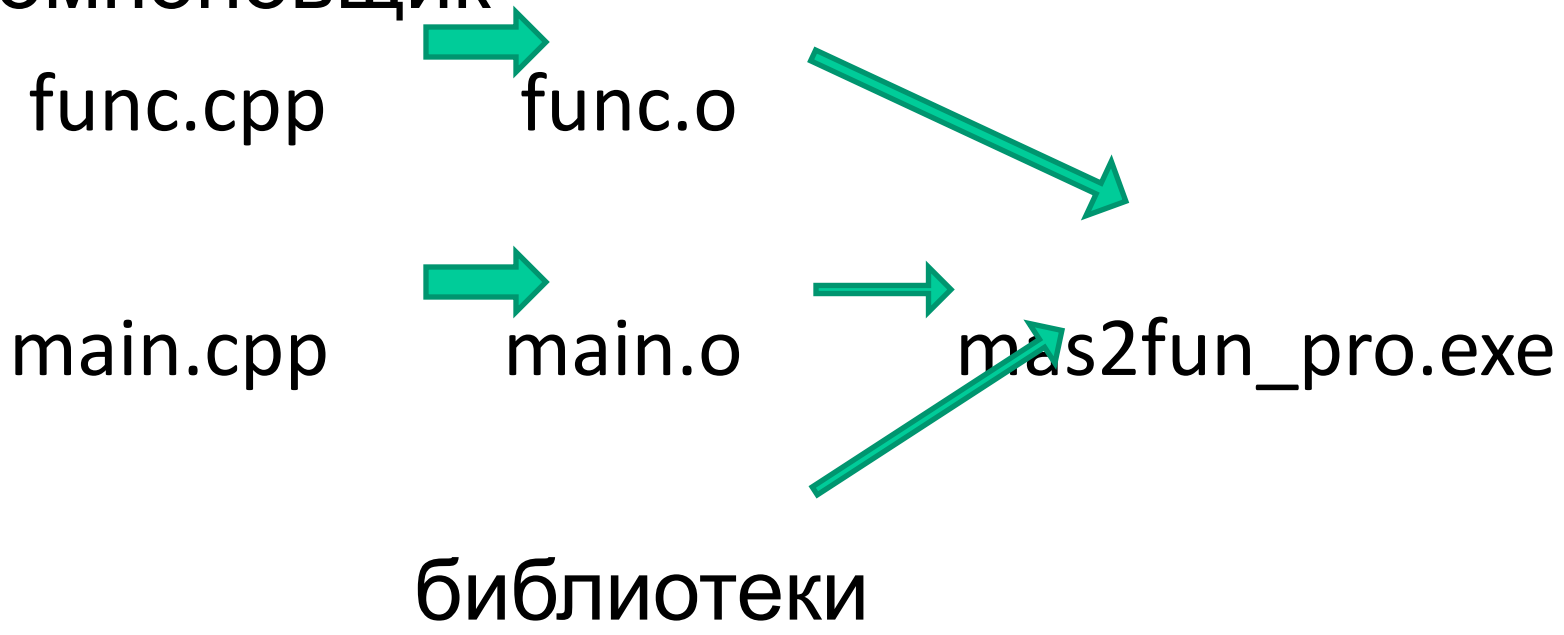
Источник | История

```
13
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  }
25
26  void outarr( float mas[][10],int n,int m)
27  {
28       int i,j;
29       printf("input array\n");
30      for(i=0; i<n; i++)
31      {
32       for(j=0;j<m;j++)
33              printf("%5.3f    ", mas[i][j]);
34       printf("\n");
35       }
36  }
```


Начальная страница
main.cpp
func.cpp
func.h
main.cpp
Источник
История
*
* Created on 17 октября 2017 г., 13:59
*/
#ifndef FUNC_H
#define FUNC_H
#endif /* FUNC_H */
void inputarr(float mas[][10],int n,int m);
void outarr(float mas[][10],int n,int m);
void calcarr(float mas[][10],int n,int m);


mas2fun_pro - NetBeans IDE 8.0.2
Файл Правка Вид Переход Источник Реорганизация кода Выполнить Отладка Профилировать Группа Сервис
Debug
Проекты
Файлы
Службы
Классы
main.cpp
Файлы заголовков
Файлы ресурсов
Файлы тестов
Важные файлы
mas2fun_pro
Исходные файлы
func.cpp
main.cpp
Файлы заголовков
func.h
Файлы ресурсов
Файлы тестов
Важные файлы
MinMax
Исходные файлы
Файлы заголовков
Файлы ресурсов
Файлы тестов
Важные файлы
Начальная страница
main.cpp
func.cpp
Источник
История
8 #include <cstdlib>
9 #include <stdio.h>
10 #include <math.h>
11 #include <iostream>
12 #include "func.h"
13 using namespace std;
14
15 int main(int argc, char** argv)
16 {
17 float a[10][10];
18 float s[10];
19 int n=2,m=3;
20 inputarr(a,n,m);
21 calcarr(a,n,m);
22 printf("Rezultat:\n");
23 outarr(a,n,m);
24 system("PAUSE");
25 return 0;
26 }

Препроцессор Компилятор
Компоновщик



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

Перегрузка функций

- **Перегрузка функций** — *это механизм, который позволяет двум родственным функциям иметь одинаковые имена.*
- В C++ несколько функций могут иметь одинаковые имена, но при условии, что их параметры будут различными. Такую ситуацию называют *перегрузкой функций* (function overloading), а функции, которые в ней задействованы, —*перегруженными* (overloaded). Перегрузка функций — один из способов реализации полиморфизма в C++.
- Рассмотрим простой пример перегрузки функций.

```cpp
#include <cstdlib>
#include <iostream>
using namespace std;
void f(int i); // один целочисленный параметр
void f(int i, int j); // два целочисленных параметра
void f(double k); // один параметр типа double
int main()
{
f (10); // вызов функции f(int)
f(10, 20); // вызов функции f (int, int)
f(12.23); // вызов функции f(double)
return 0;
}
void f(int i)
{
cout << "В функции f(int), i равно " << i << '\n';
}
void f(int i, int j)
{
cout << "В функции f(int, int), i равно " << i;
cout << ", j равно " << j << '\n';
}
void f(double k)
{
cout << "В функции f(double), k равно " << k << ' \n';
}
```

main

Терминал | Уведомления | Вывод ×

ыполнение) × | nezav_ssilka (Собрать, Выполнение) × | nezav_ssilka (Выполнение) × | peregru

```
В функции f(int), i равно 10
В функции f(int, int), i равно 10, j равно 20
В функции f(double), k равно 12.238202
ВЫПОЛНЕНИЕ SUCCESSFUL (общее время: 141ms)
```

Функция *f()* перегружается три раза. Первая версия принимает один целочисленный параметр, вторая — два целочисленных параметра, а третья — один double параметр. Поскольку списки параметров для всех трех версий различны, компилятор обладает достаточной информацией, чтобы вызвать правильную версию каждой функции. В общем случае для создания перегрузки некоторой функции достаточно объявить различные ее версии.

Для определения того, какую версию перегруженной функции вызвать, компилятор использует *тип* и/или *количество аргументов*. Таким образом, перегруженные функции должны отличаться *типами* и/или *числом параметров*. Несмотря на то что перегруженные методы могут отличаться и типами возвращаемых значений, этого вида информации недостаточно для C++, чтобы во всех случаях компилятор мог решить, какую именно функцию нужно вызвать.

Аргументы, передаваемые функции по умолчанию

- В C++ мы можем придать параметру некоторое значение, которое будет автоматически использовано, если при вызове функции не задается аргумент, соответствующий этому параметру. Аргументы, передаваемые функции по умолчанию, можно использовать, чтобы упростить обращение к сложным функциям, а также в качестве "сокращенной формы" перегрузки функций.
- Задание аргументов, передаваемых функции по умолчанию, синтаксически аналогично инициализации переменных. Рассмотрим следующий пример, в котором объявляется функция *myfunc()*, принимающая один аргумент типа *double* с действующим по умолчанию значением *0.0* и один символьный аргумент с действующим по умолчанию значением *'X'*.
- void myfunc(double num = 0.0, char ch = 'X')
- {
- .
- .
- }
- После такого объявления функцию *myfunc()* можно вызвать одним из трех следующих способов.
- myfunc(198.234, 'A'); // Передаем явно заданные значения.
- myfunc(10.1); // Передаем для параметра num значение 10.1, а для параметра ch позволяем применить аргумент, задаваемый по умолчанию ('X').
- myfunc(); // Для обоих параметров num и ch позволяем применить аргументы, задаваемые по умолчанию.
- При первом вызове параметру *num* передается значение *198.234*, а параметру *ch*— символ*'A'*. Во время второго вызова параметру *num* передается значение*10.1*, а параметр *ch* по умолчанию устанавливается равным символу*‘ X'*. Наконец, в результате третьего вызова как параметр *num*, так и параметр *ch* по умолчанию устанавливаются равными значениям, заданным в объявлении функции.

- Включение в C++ возможности передачи аргументов по умолчанию позволяет программистам упрощать код программ. Чтобы предусмотреть максимально возможное количество ситуаций и обеспечить их корректную обработку, функции часто объявляются с большим числом параметров, чем необходимо в наиболее распространенных случаях. Поэтому благодаря применению аргументов по умолчанию программисту нужно указывать не все аргументы (используемые в общем случае), а только те, которые имеют смысл для определенной ситуации.
- *Аргумент, передаваемый функции по умолчанию, представляет собой значение, которое будет автоматически передано параметру функции в случае, если аргумент, соответствующий этому параметру, явным образом не задан.*
- Насколько полезна возможность передачи аргументов по умолчанию, показано на примере функции *clrscr()*, представленной в следующей программе. Функция *clrscr()* очищает экран путем вывода последовательности символов новой строки (это не самый эффективный способ, но он очень подходит для данного примера). Поскольку в наиболее часто используемом режиме представления видеоизображений на экран дисплея выводится*25* строк текста, то в качестве аргумента по умолчанию используется значение *25*. Но так как в других видеорежимах на экране может отображаться больше или меньше *25* строк, аргумент, действующий по умолчанию, можно переопределить, явно указав нужное значение.

```
#include <cstdlib>
#include <iostream>
using namespace std;
void clrscr(int size=25);
int main()
{
int i;
for(i=0; i<30; i++ ) cout << i << '\n';
clrscr(); // Очищаем 25 строк.
for(i=0; i<30; i++ ) cout << i << '\n';
clrscr(10); // Очищаем 10 строк.
return 0;
}
void clrscr(int size)
{
for(; size; size--)cout << '\n';
}
```

Как видно из кода этой программы, если значение, действующее по умолчанию, соответствует ситуации, при вызове функции *clrscr()* аргумент указывать не нужно. Но в других случаях аргумент, действующий по умолчанию, можно переопределить и передать параметру *size* нужное значение.

При создании функций, имеющих значения аргументов, передаваемых по умолчанию, необходимо помнить две вещи. Эти значения по умолчанию должны быть заданы только однажды, причем при первом объявлении функции в файле. В предыдущем примере аргумент по умолчанию был задан в прототипе функции *clrscr()*. При попытке определить новые (или даже те же) передаваемые по умолчанию значения аргументов в определении функции *clrscr()* компилятор отобразит сообщение об ошибке и не скомпилирует вашу программу.

Несмотря на то что передаваемые по умолчанию аргументы должны быть определены только один раз, для каждой версии перегруженной функции для передачи по умолчанию можно задавать различные аргументы. Таким образом, разные версии перегруженной функции могут иметь различные значения аргументов, действующие по умолчанию.

- Важно понимать, что все параметры, которые принимают значения по умолчанию, должны быть расположены справа от остальных. Например, следующий прототип функции содержит ошибку.
- // Неверно!
- void f(int a = 1, int b);
- Если вы начали определять параметры, которые принимают значения по умолчанию, нельзя после них указывать параметры, задаваемые при вызове функции только явным образом. Поэтому следующее объявление также неверно и не будет скомпилировано.
- int myfunc(float f, char *str, int i=10, int j);
- Поскольку для параметра *i* определено значение по умолчанию, для параметра *j* также нужно задать значение по умолчанию.

Об использовании аргументов, передаваемых по умолчанию

Несмотря на то что аргументы, передаваемые функции по умолчанию, — очень мощное средство программирования (при их корректном использовании), с ними могут иногда возникать проблемы. Их назначение — позволить функции эффективно выполнять свою работу, обеспечивая при всей простоте этого механизма значительную гибкость. В этом смысле все передаваемые по умолчанию аргументы должны отражать способ наиболее общего использования функции или альтернативного ее применения. Если не существует некоторого единого значения, которое обычно присваивается тому или иному параметру, то и нет смысла объявлять соответствующий аргумент по умолчанию. На самом деле объявление аргументов, передаваемых функции по умолчанию, при недостаточном для этого основании деструктуризирует код, поскольку такие аргументы способны сбить с толку любого, кому придется разбираться в такой программе. Наконец, основным принципом использования аргументов по умолчанию должен быть, как у врачей, принцип *"не навредить"*. Другими словами, случайное использование аргумента по умолчанию не должно привести к необратимым отрицательным последствиям. Ведь такой аргумент можно просто забыть указать при вызове некоторой функции, и, если это случится, подобный промах не должен вызвать, например, потерю важных данных!

```cpp
#include <iostream>

using namespace std;
void param( int p1, int p2, int p3=3, int p4=4)
{
    cout<<"p1="<<p1<<"  p2="<<p2<<"  p3="<<p3<<"  p4="<<p4<<endl;
}
int main()
{
    param(11,12,13,14);
    param(11,12,13);
    param(11,12);
    return 0;
}
```

```
D:\CodeBloks\arg\bin\Debug\arg.exe
p1=11  p2=12  p3=13  p4=14
p1=11  p2=12  p3=13  p4=4
p1=11  p2=12  p3=3  p4=4

Process returned 0 (0x0)   execution time
Press any key to continue.
```

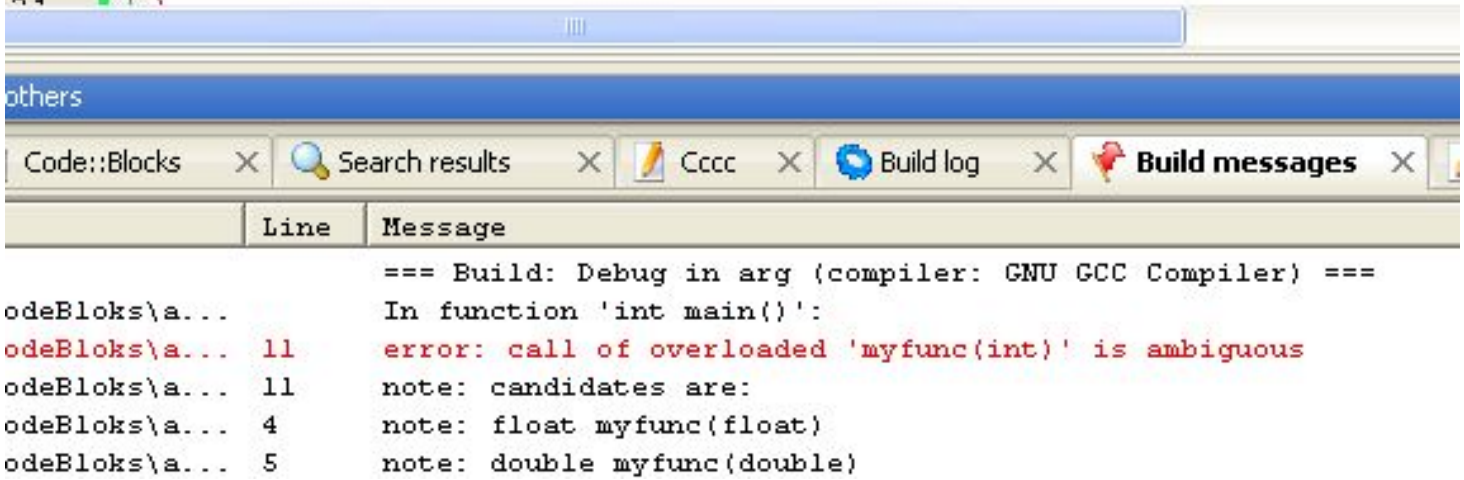
Перегрузка функций и неоднозначность

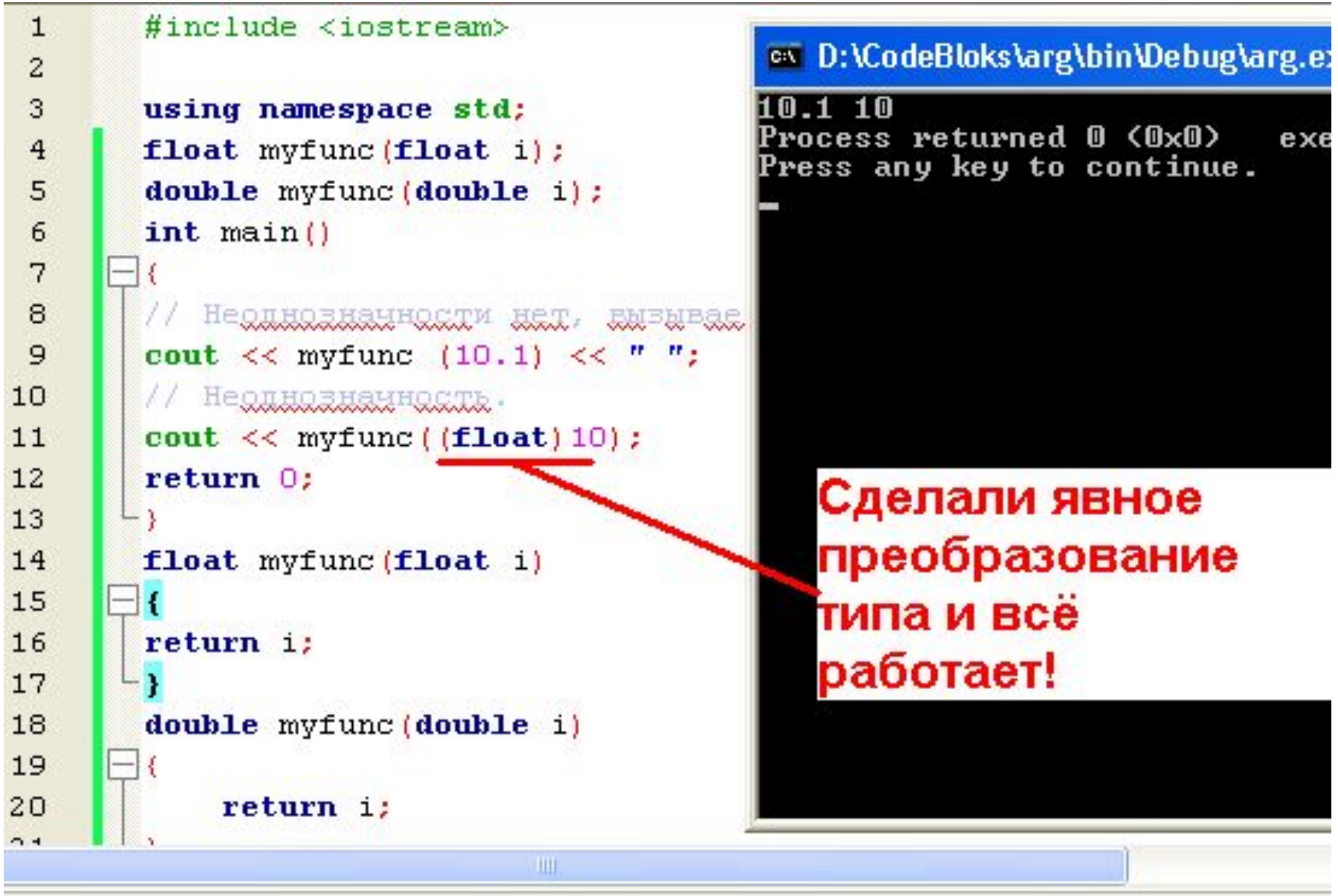
- *Неоднозначность возникает тогда, когда компилятор не может определить различие между двумя перегруженными функциями.*
- Возможны ситуации, в которых компилятор не способен сделать выбор между двумя (или более) корректно перегруженными функциями. Такие ситуации и называют *неоднозначными*. Инструкции, создающие неоднозначность, являются
- ошибочными, а программы, которые их содержат, скомпилированы не будут.
- Основной причиной неоднозначности в C++ является автоматическое преобразование типов. В C++ делается попытка автоматически преобразовать тип аргументов, используемых для вызова функции, в тип параметров, определенных функцией. Рассмотрим пример.
- int myfunc(double d);
- .
- .
- .
- cout << myfunc('c'); // Ошибки нет, выполняется преобразование типов.
- Как отмечено в комментарии, ошибки здесь нет, поскольку C++ автоматически преобразует символ *'c'* в его *double*-эквивалент. Вообще говоря, в C++ запрещено довольно мало видов преобразований типов. Несмотря на то что автоматическое преобразование типов — это очень удобно, оно, тем не менее, является главной причиной неоднозначности. Рассмотрим следующую программу.

```
// Неоднозначность вследствие перегрузки функций.
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);
int main()
{
// Неоднозначности нет, вызывается функция myfunc(double).
cout << myfunc (10.1) << " ";
// Неоднозначность.
cout << myfunc(10);
return 0;
}
float myfunc(float i)
{
return i;
}
double myfunc(double i)
{
return -i;
}
```

- Здесь благодаря перегрузке функция *myfunc()* может принимать аргументы либо типа *float*, либо типа *double*. При выполнении строки кода
- cout << myfunc (10.1) << " ";
- не возникает никакой неоднозначности: компилятор "уверенно" обеспечивает вызов функции *myfunc(double)*, поскольку, если не задано явным образом иное, все литералы с плавающей точкой в C++ автоматически получают тип *double*. Но при вызове функции *myfunc()* с аргументом, равным целому числу *10*, в программу вносится неоднозначность, поскольку компилятору неизвестно, в какой тип ему следует преобразовать этот аргумент: *float* или *double*. Оба преобразования допустимы. В такой неоднозначной ситуации будет выдано сообщение об ошибке, и программа не скомпилируется.
- На примере этой программы хотелось бы подчеркнуть, что неоднозначность в ней вызвана не перегрузкой функции *myfunc()*, объявленной дважды для приема *double*- и *float*-аргумента, а использованием при конкретном вызове функции *myfunc()* аргумента неопределенного для преобразования типа. Другими словами, ошибка состоит не в перегрузке функции *myfunc()*, а в конкретном ее вызове.

```
1   #include <iostream>
2
3   using namespace std;
4   float myfunc(float i);
5   double myfunc(double i);
6   int main()
7   {
8   // Неоднозначности нет, вызывается функция myfunc(double).
9   cout << myfunc (10.1) << " ";
10  // Неоднозначность.
11  cout << myfunc(10);
12  return 0;
13  }
14  float myfunc(float i)
15  {
16  return i;
17  }
18  double myfunc(double i)
19  {
20      return i;
```

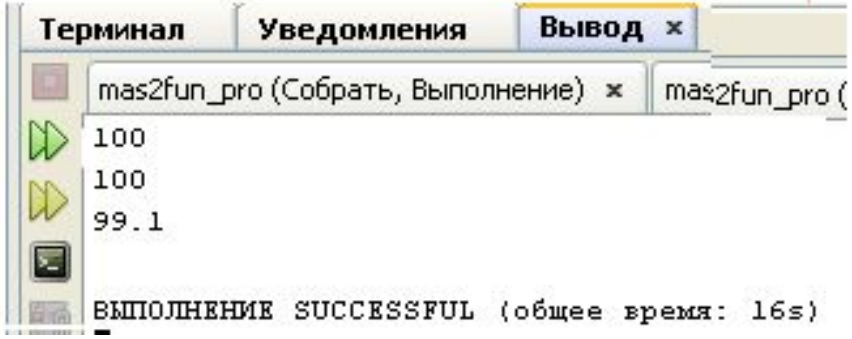



#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);
int main()
{
// Неоднозначности нет, вызывае
cout << myfunc (10.1) << " ";
// Неоднозначность.
cout << myfunc((float)10);
return 0;
}
float myfunc(float i)
{
return i;
}
double myfunc(double i)
{
return i;
D:\CodeBloks\arg\bin\Debug\arg.ex
10.1 10
Process returned 0 (0x0) exe
Press any key to continue.
Сделали явное преобразование типа и всё работает!

Возврат ссылок

Функция может возвращать ссылку. В программировании на C++ предусмотрено несколько применений для ссылочных значений, возвращаемых функциями. Если функция возвращает ссылку, это означает, что она возвращает неявный указатель на значение, передаваемое ею в инструкции *return*. Этот факт открывает поразительные возможности: функцию, оказывается, можно использовать в левой части инструкции присваивания!

```cpp
#include <cstdlib>
#include <iostream>
using namespace std;
double &f();
double val = 100.0;
int main()
{
double newval;
cout << f() << '\n'; // Отображаем значение val.
newval = f(); // Присваиваем значение val переменной newval.
cout << newval << '\n'; // Отображаем значение newval.
f() = 99.1; // Изменяем значение val.
cout << f() << '\n'; // Отображаем новое значение val.
system("PAUSE");
return 0;
}
double &f()
{
return val; // Возвращаем ссылку на val.
}
```



- **Рассмотрим эту программу подробнее. Судя по прототипу функции *f()*, она должна возвращать ссылку на *double*-значение. За объявлением функции *f()* следует объявление глобальной переменной *val*, которая инициализируется значением *100*. При выполнении следующей инструкции выводится исходное значение переменной *val*.**

- **cout << f() << '\n'; // Отображаем значение val.**

- **После вызова функция *f()* возвращает ссылку на переменную *val*. Поскольку функция *f()* объявлена с "обязательством" вернуть ссылку, при выполнении строки**
-
- **return val; // Возвращаем ссылку на val.**

- **автоматически возвращается ссылка на глобальную переменную *val*. Эта ссылка затем используется инструкцией *cout* для отображения значения *val*.**
- **При выполнении строки**

- **newval = f(); //Присваиваем значение val переменной newval.**

- **ссылка на переменную *val*, возвращенная функцией *f()*, используется для присвоения значения *val* переменной *newval*.**
- **А вот самая интересная строка в программе.**

- **f() = 99.1; // Изменяем значение val.**

- **При выполнении этой инструкции присваивания значение переменной *val* становится равным числу *99,1*. И вот почему: поскольку функция *f()* возвращает ссылку на переменную *val*, эта ссылка и является приемником инструкции присваивания. Таким образом, значение *99,1* присваивается переменной *val* косвенно, через ссылку на нее, которую возвращает функция *f()*.**
- **Наконец, при выполнении строки**

- **cout << f() << '\n'; // Отображаем новое значение val.**

- **отображается новое значение переменной *val* (после того, как ссылка на переменную *val* будет возвращена в результате вызова функции *f()* в инструкции *cout*).**

программа изменяет значения второго и четвертого элементов массива

```
8  #include <cstdlib>
9  #include <iostream>
10 using namespace std;
11 double &change_it(int i);
12 // Функция возвращает ссылку.
13 double vals[] = {1.1, 2.2, 3.3, 4.4, 5.5};
14 int main()
15 {
16 int i;
17 cout << "Вот исходные значения: ";
18 for(i=0; i<5; i++)
19 cout << vals[i] << ' ';
20 cout << '\n';
21 change_it(1) = 5298.23; // Изменяем 2-йэлемент.
22 change_it(3) = -98.8; //Изменяем4-йэлемент.
23 cout << "Вот измененные значения: ";
24 for(i=0; i<5; i++)
25 cout << vals[i] << ' ';
26 cout << '\n';
27 return 0;
28 }
29 double &change_it(int i)
30 {
31 return vals[i]; // Возвращаем ссылку на i-йэлемент.
32 }
```

change_it

Терминал | Уведомления | Вывод ×

change (Собрать, Выполнение) × | change (Выполнение) ×

```
Вот исходные значения: 1.1 2.2 3.3 4.4 5.5
Вот измененные значения: 1.1 5298.23 3.3 -98.8 5.5
```

- Функция *change_it()* объявлена как возвращающая ссылку на значение типа *double*. Говоря более конкретно, она возвращает ссылку на элемент массива *vals*, который задан ей в качестве параметра *i*. Таким образом, при выполнении следующей инструкции функции main()
- change_it(1) = 5298.23; // Изменяем 2-йэлемент.
- функция *change_it()* возвращает ссылку на элемент *vals[1]*. Через эту ссылку элементу *vals[1]* теперь присваивается значение *5298,23*. Аналогичные события происходят при выполнении и этой инструкции.
- change_it(3) = -98.8; //Изменяем4-йэлемент.
- Поскольку функция *change_it()* возвращает ссылку на конкретный элемент массива *vals*, ее можно использовать в левой части инструкции для присвоения нового значения соответствующему элементу массива.
- Организуя возврат функцией ссылки, необходимо позаботиться о том, чтобы объект, на который она ссылается, не выходил за пределы действующей области видимости. Например:
- //Здесь ошибка: нельзя возвращать ссылку
- //на локальную переменную.
- int &f()
- {
- int i=10;
- return i;
- }
- При завершении функции *f()* локальная переменная *i* выйдет за пределы области видимости. Следовательно, ссылка на переменную *i*, возвращаемая функцией *f()*, будет неопределенной. В действительности некоторые компиляторы не скомпилируют функцию *f()* в таком виде, и именно по этой причине. Однако проблема такого рода может быть создана опосредованно, поэтому нужно внимательно отнестись к тому, на какой объект будет возвращать ссылку ваша функция.

Создание ограниченного массива

- Ссылочный тип в качестве типа значения, возвращаемого функцией, можно с успехом применить для создания ограниченного массива. Как вы знаете, при выполнении C++-кода проверка нарушения границ при индексировании массивов не предусмотрена. Это означает, что может произойти выход за границы области памяти, выделенной для массива. Другими словами, может быть задан индекс, превышающий размер массива. Однако путем создания *ограниченного*, или *безопасного*, массива выход за его границы можно предотвратить. При работе с таким массивом любой выходящий за установленные границы индекс не допускается для индексирования массива.
- Один из способов создания ограниченного массива иллюстрируется в следующей программе.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int &put(int i); // Помещаем значение в массив.
int get(int i); // Считываем значение из массива.
int vals[10];
int error = -1;
int main()
{
put(0) = 10; // Помещаем значения в массив.
put(1) = 20;
put(9) = 30;
cout << get(0) << ' ';
cout << get(1) << ' ';
cout << get(9) << ' ';
// А теперь специально генерируем ошибку.
put(12) =1; // Индекс за пределами границ массива.
return 0;
}
// Функция занесения значения в массив.
int &put(int i)
{
if((i>=0) && (i<10))
return vals[i]; // Возвращаем ссылку на i-йэлемент.
else {
cout << "Ошибка нарушения границ!\n";
return error; // Возвращаем ссылку на error.
}
}
// Функция считывания значения из массива.
int get(int i)
{
if(i>=0 && i<10)
return vals[i]; // Возвращаем значение i-гоэлемента.
else {
cout << "Ошибка нарушения границ!\n";
return error; // Возвращаем значение переменной error.
}
}
```

put

Терминал | Уведомления | Вывод ×

mas_ogran (Собрать, Выполнение) × | mas_ogran (Выполнение) ×

```
10 20 30 Ошибка нарушения границ!
```

В этой программе создается безопасный массив, предназначенный для хранения десяти целочисленных значений. Чтобы поместить в него значение, используйте функцию *put()*, а чтобы прочитать нужный элемент массива, вызовите функцию *get()*. При использовании обеих функций индекс интересующего вас элемента задается в виде аргумента. Как видно из текста программы, функции *get()* и *put()* не допускают выход за границы области памяти, выделенной для массива. Обратите внимание на то, что функция *put()* возвращает ссылку на заданный элемент и поэтому законно используется в левой части инструкции присваивания.

Независимые ссылки

- Понятие ссылки включено в C++ главным образом для поддержки способа передачи параметров "по ссылке" и для использования в качестве ссылочного типа значения, возвращаемого функцией. Несмотря на это, можно объявить независимую переменную ссылочного типа, которая и называется *независимой ссылкой*. Однако, справедливости ради, необходимо сказать, что эти независимые ссылочные переменные используются довольно редко, поскольку они могут "сбить с пути истинного" вашу программу. Сделав (для очистки совести) эти замечания, мы все же можем уделить независимым ссылкам некоторое внимание.
- **Независимая ссылка** — *это просто еще одно название для переменных некоторого иного типа.*
- Независимая ссылка должна указывать на некоторый объект. Следовательно, независимая ссылка должна быть инициализирована при ее объявлении. В общем случае это означает, что ей будет присвоен адрес некоторой ранее объявленной переменной. После этого имя такой ссылочной переменной можно применять везде, где может быть использована переменная, на которую она ссылается. И в самом деле, между ссылкой и переменной, на которую она ссылается, практически нет никакой разницы. Рассмотрим, например, следующую программу.

```
1
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5  int main()
6  {
7  int j, k;
8  int &i = j; // независимая ссылка
9  j = 10;
10 cout << j << " " << i; // Выводится: 10 10
11 k = 121;
12 i = k; // Копирует в переменную j значение п
13 cout << "\n" << j; // Выводится: 121
14 return 0;
15 }
```

main

Терминал | Уведомления | Вывод ×

mas_ogran (Собрать, Выполнение) × | mas_ogran (Выполнение) × | nez

```
10 10
121
```

Адрес, который содержит ссылочная переменная, фиксирован и его нельзя изменить. Следовательно, при выполнении инструкции *i = k* в переменную *j* (адресуемую ссылкой *i*) копируется значение переменной *k*, а не ее адрес. В качестве еще одного примера отметим, что после выполнения инструкции *i++* ссылочная переменная *i* не станет содержать новый адрес, как можно было бы предположить. В данном случае на *1* увеличится содержимое переменной *j*. Как было отмечено выше, независимые ссылки лучше не использовать, поскольку чаще всего им можно найти замену, а их неаккуратное применение может исказить ваш код. Согласитесь: наличие двух имен для одной и той же переменной, по сути, уже создает ситуацию, потенциально порождающую недоразумения.

Ограничения при использовании ссылок

На применение ссылочных переменных накладывается ряд следующих ограничений.

- Нельзя ссылаться на ссылочную переменную.
- Нельзя создавать массивы ссылок.
- Нельзя создавать указатель на ссылку, т.е. нельзя к ссылке применять оператор *"&"*
- Ссылки не разрешено использовать для битовых полей структур

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

ele

```
#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) );
}
```

Санкт-Петербургский государственный университет
телекоммуникаций им. проф. М.А. Бонч-Бруевича

Двумерные динамические массивы

СПб ГУТ)))

Вспомним одномерные

```
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];
```

Объявляем указатели

Выделяем динамическую память и при сваиваем адреса указателям

Работаем с дин. масс. как с обычным массивом

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

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

Для обращения к элементу массива используем квадратные скобки

Освобождаем память

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

Двумерный динамический массив

```
1  #include <iostream>
2  #include<stdio.h>
3  #include<cstdlib>
4  using namespace std;
5
6  int main()
7  {
8      double *m2d[5];
9      int i,j;
10     for(i=0;i<5;i++)
11         m2d[i]=(double*)malloc(10*sizeof(double));
12     for(i=0;i<5;i++)
13         for(j=0;j<10;j++)
14             m2d[i][j]=i+j;
15     for(i=0;i<5;i++)
16     {
17         for(j=0;j<10;j++)
18             printf("%7.2lf",m2d[i][j]);
19         printf("\n");
20     }
21     for(i=0;i<5;i++)
22         free(m2d[i]);
23     return 0;
24 }
25
```

```
D:\CodeBloks\m2d\bin\Debug\m2d.exe
   0.00    1.00    2.00    3.00    4.00    5.00    6.00    7.00    8.00    9.00
   1.00    2.00    3.00    4.00    5.00    6.00    7.00    8.00    9.00   10.00
   2.00    3.00    4.00    5.00    6.00    7.00    8.00    9.00   10.00   11.00
   3.00    4.00    5.00    6.00    7.00    8.00    9.00   10.00   11.00   12.00
   4.00    5.00    6.00    7.00    8.00    9.00   10.00   11.00   12.00   13.00
```

Создаём массив из 5 указателей на double

Выделяем память для строк. Каждая строка (одномерный массив) будет содержать 10 элементов double

Работаем как с обычным массивом

Не забываем освободить память

Недостаток!

```
using namespace std;

int main()
{
    double *m2d[5];
    int i,j;
```

Необходимо указывать размер (число строк)

Вывод! Нужно сделать массив указателей динамическим!

Вместо обычного массива указателей double *a[5] создаём динамический массив указателей

```
double **a;
a=new double *[n];
```

```
1  #include <iostream>
2  #include<stdio.h>
3  #include<cstdlib>
4  using namespace std;
5
6  int main()
7  {
8      double **m2d;
9      int i,j,n;
10     printf("n= ");
11     scanf("%d",&n);
12     m2d=(double**)malloc(n*sizeof(double*));
13     for(i=0;i<5;i++)
14         m2d[i]=(double*)malloc(10*sizeof(double)
15     for(i=0;i<5;i++)
16         for(j=0;j<10;j++)
17             m2d[i][j]=i+j;
18     for(i=0;i<5;i++)
19     {
20         for(j=0;j<10;j++)
21             printf("%7.2lf",m2d[i][j]);
22         printf("\n");
23      }
24     for(i=0;i<5;i++)
25         free(m2d[i]);
26     free(m2d);
27     return 0;
28 }
29
```

```
D:\CodeBloks\m2d\bin\Debug\m2d.exe
n= 5
   0.00   1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00
   1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00  10.00
   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00  10.00  11.00
   3.00   4.00   5.00   6.00   7.00   8.00   9.00  10.00  11.00  12.00
   4.00   5.00   6.00   7.00   8.00   9.00  10.00  11.00  12.00  13.00
```

Создаём динамический массив указателей

Освобождаем память для строк

И только потом для массива указателей

```
double** init(int n,int m)
{
int i;
double **a;
a=new double *[n];
```

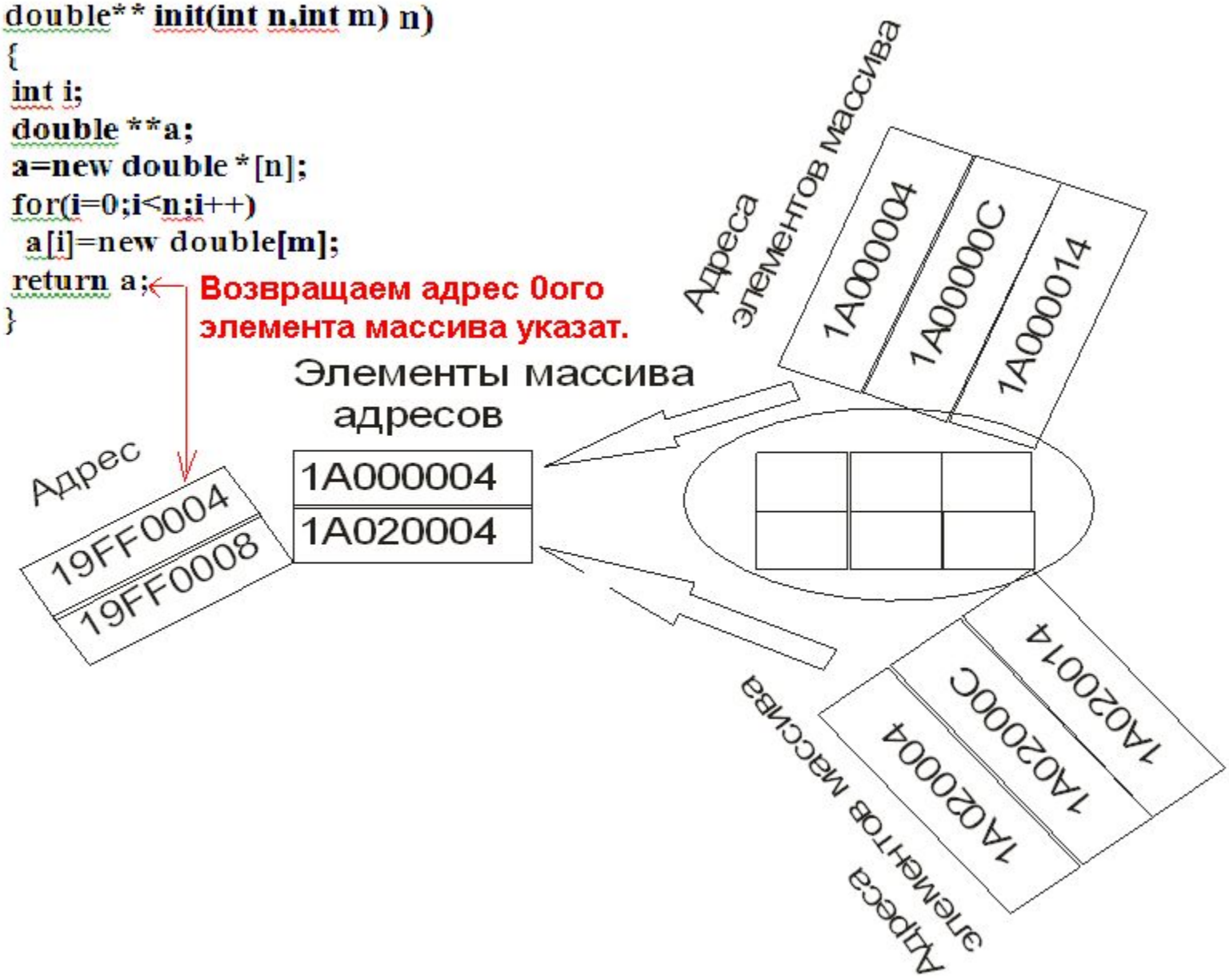
```
double** init(int n,int m)
{
 int i;
 double **a;
 a=new double *[n];
 for(i=0;i<n;i++)
  a[i]=new double[m];
```

Выделяем память для n строк (на картинке 2 строки). Каждая строка состоит из m элементов типа double (на картинке две строки по 3 элемента double)

Адреса строк записываем в массив указателей

```
double** init(int n,int m) n)
{
int i;
double **a;
a=new double *[n];
for(i=0;i<n;i++)
 a[i]=new double[m];
return a;
}
```

Возвращаем адрес 0ого элемента массива указат.



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();
}
```

Препроцессор C++

Препроцессор C++ подвергает программу различным текстовым преобразованиям до реальной трансляции исходного кода в объектный. Препроцессор обрабатывает команды, называемые *директивами препроцессора*

Препроцессор C++ включает следующие директивы.

#define	#error	#include
#if	#else	#elif
#endif	#ifdef	#ifndef
#undef	#line	#pragma

Все директивы препроцессора начинаются с символа '#'.

Директива #define

- Директива *#define* используется для определения идентификатора и символьной последовательности, которая будет подставлена вместо идентификатора везде, где он встречается в исходном коде программы. Этот идентификатор называется макроименем, а процесс замены — *макроподстановкой* (реализацией макрорасширения). Общий формат использования этой директивы имеет следующий вид.
- #define макроимя последовательность_символов
- Обратите внимание на то, что здесь нет точки с запятой. Заданная *последовательность_символов* завершается только символом конца строки. Между элементами *макроимя* (имя_макроса) и *последовательность_символов* может быть любое количество пробелов.
- Итак, после включения этой директивы каждое вхождение текстового фрагмента, определенное как *макроимя*, заменяется заданным элементом *последовательность_символов*. Например, если вы хотите использовать слово *UP* в качестве значения *1* и слово *DOWN* в качестве значения *0*, объявите такие директивы *#define*.
- #define UP 1
- #define DOWN 0
- Данные директивы вынудят компилятор подставлять *1* или *0* каждый раз, когда в файле исходного кода встретится слово *UP* или *DOWN* соответственно. Например, при выполнении инструкции:
- cout << UP << ' ' << DOWN << ' ' << UP + UP;
- На экран будет выведено следующее:
- 1 0 2

- После определения имени макроса его можно использовать как часть определения других макроимен. Например, следующий код определяет имена *ONE*,*TWO* и *THREE* и соответствующие им значения.
- #define ONE 1
- #define TWO ONE+ONE
- #define THREE ONE+TWO
- Важно понимать, что *макроподстановка* — это просто замена идентификатора соответствующей строкой. Следовательно, если вам нужно определить стандартное сообщение, используйте код, подобный этому.
- #define GETFILE "Введите имя файла"
- // ...
- Препроцессор заменит строкой *"Введите имя файла"* каждое вхождение идентификатора *GETFILE*. Для компилятора эта cout-инструкция
- cout << GETFILE;
- в действительности выглядит так.
- cout << "Введите имя файла";
- Никакой текстовой замены не произойдет, если идентификатор находится в строке, заключенной в кавычки. Например, при выполнении следующего кода
- #define GETFILE "Введите имя файла"
- // ...
- cout << "GETFILE - это макроимя\n";
- на экране будет отображена эта информация
- GETFILE - это макроимя,
- а не эта:
- Введите имя файла - это макроимя

- Если текстовая последовательность не помещается на строке, ее можно продолжить на следующей, поставив обратную косую черту в конце строки, как показано в этом примере.
- #define LONG_STRING "Это очень длинная последовательность,\
- которая используется в качестве примера."
- Среди C++ - программистов принято использовать для макроимен прописные буквы. Это соглашение позволяет с первого взгляда понять, что здесь используется макроподстановка. Кроме того, лучше всего поместить все директивы *#define* в начало файла или включить в отдельный файл, чтобы не искать их потом по всей программе.
- Макроподстановки часто используются для определения *"магических чисел«* программы. Например, у вас есть программа, которая определяет некоторый массив, и ряд функций, которые получают доступ к нему. Вместо *"жесткого"* кодирования размера массива с помощью константы лучше определить имя, которое бы представляло размер, а затем использовать это имя везде, где должен стоять размер массива. Тогда, если этот размер придется изменить, вам достаточно будет внести только одно изменение, а затем перекомпилировать программу. Рассмотрим пример.
- #define MAX_SIZE 100
- // ...
- float balance[MAX_SIZE];
- double index[MAX_SIZE];
- int num_emp[MAX_SIZE];

Макроопределения, действующие как функции

- Директива *#define* имеет еще одно назначение: макроимя может использоваться с аргументами. При каждом вхождении макроимени связанные с ним аргументы заменяются реальными аргументами, указанными в коде программы. Такие макроопределения действуют подобно функциям. Рассмотрим пример.
- /* Использование "функциональных" макроопределений. */
- #include <iostream>
- using namespace std;
- #define MIN(a, b) (((a)<(b)) ? a : b)
- int main()
- {
- int x, y;
- x = 10;
- y = 20;
- cout << "Минимум равен: " << MIN(x, y);
- return 0;
- }
- При компиляции этой программы выражение, определенное идентификатором *MIN (a, b)*, будет заменено, но *x* и *y* будут рассматриваться как операнды. Это значит, что cout инструкция после компиляции будет выглядеть так.
- cout << "Минимум равен: " << (((x)<(y)) ? x : y);
- По сути, такое макроопределение представляет собой способ определить функцию, которая вместо вызова позволяет раскрыть свой код в строке.

- **Макроопределения, действующие как функции,** — *это макроопределения, которые принимают аргументы.*
- Кажущиеся избыточными круглые скобки, в которые заключено макроопределение *MIN*, необходимы, чтобы гарантировать правильное восприятие компилятором заменяемого выражения. На самом деле дополнительные круглые скобки должны применяться практически ко всем макроопределениям, действующим подобно функциям. Нужно всегда очень внимательно относиться к определению таких макросов; в противном случае возможно получение неожиданных результатов. Рассмотрим, например, эту короткую программу, которая использует макрос для определения четности значения.
- // Эта программа дает неверный ответ.
- #include <iostream>
- using namespace std;
- #define EVEN(a) a%2==0 ? 1 : 0
- int main()
- {
- if(EVEN(9+1)) cout << "четное число";
- else cout << "нечетное число ";
- return 0;
- }
- Эта программа не будет работать корректно, поскольку не обеспечена правильная подстановка значений. При компиляции выражение *EVEN(9+1)* будет заменено следующим образом.
- 9+1%2==0 ? 1 : 0
- Напомню, что оператор *"%"* имеет более высокий приоритет, чем оператор*"+"*. Это значит,

```
// Эта программа работает корректно.
#include <iostream>
using namespace std;
#define EVEN(a) (a)%2==0 ? 1 : 0
int main()
{
if(EVEN(9+1)) cout << "четное число";
else cout << "нечетное число";
return 0;
}
```

- Теперь сумма *9+1* вычисляется до выполнения операции деления по модулю. В общем случае лучше всегда заключать параметры макроопределения в круглые скобки, чтобы избежать непредвиденных результатов, подобных описанному выше.
- Использование макроопределений вместо настоящих функций имеет одно существенное достоинство: поскольку код макроопределения расширяется в строке, и нет никаких затрат системных ресурсов на вызов функции, скорость работы вашей программы будет выше по сравнению с применением обычной функции. Но повышение скорости является платой за увеличение размера программы (из-задублирования кода функции).
- **Важно!** *Несмотря на то что макроопределения все еще встречаются в C++-коде, макросы, действующие подобно функциям, можно заменить спецификатором inline, который справляется с той же ролью лучше и безопаснее. (Вспомните: спецификатор inline обеспечивает вместо вызова функции расширение ее тела в строке.) Кроме того, inline-функции не требуют дополнительных круглых скобок, без которых не могут обойтись макроопределения. Однако макросы, действующие подобно функциям, все еще остаются частью C++-программ, поскольку многиеC/C++-программистыпродолжают использовать их по привычке.*

Директива #error

- *Директива #error отображает сообщение об ошибке.*
- Директива *#error* дает указание компилятору остановить компиляцию. Она используется в основном для отладки. Общий формат ее записи таков.
- #error сообщение
- Обратите внимание на то, что элемент *сообщение* не заключен в двойные кавычки. При встрече с директивой *#error* отображается заданное *сообщение* и другая информация (она зависит от конкретной реализации рабочей среды), после чего компиляция прекращается. Чтобы узнать, какую информацию отображает в этом случае компилятор, достаточно провести эксперимент.

Директива #include

- Директива *#include* включает заголовочный или другой исходный файл.
- Директива препроцессора *#include* обязывает компилятор включить либо стандартный заголовок, либо другой исходный файл, имя которого указано в директиве *#include*. Имя стандартных заголовков заключается в угловые скобки. Например, эта директива
- #include <vector>
- Включает стандартный заголовок для векторов.
- При включении другого исходного файла его имя может быть указано в двойных кавычках или угловых скобках. Например, следующие две директивы обязывают C++ прочитать и скомпилировать файл с именем *sample.h*:
- #include <sample.h>
- #include "sample.h"
- Если имя файла заключено в угловые скобки, то поиск файла будет осуществляться в одном или нескольких специальных каталогах, определенных конкретной реализацией.
- Если же имя файла заключено в кавычки, поиск файла выполняется, как правило, в
- текущем каталоге (что также определено конкретной реализацией). Во многих случаях это означает поиск текущего рабочего каталога. Если заданный файл не найден, поиск повторяется с использованием первого способа (как если бы имя файла было заключено в угловые скобки).

Директивы условной компиляции

- Существуют директивы, которые позволяют избирательно компилировать части исходного кода. Этот процесс, именуемый *условной компиляцией*, широко используется коммерческими фирмами по разработке программного обеспечения, которые создают и поддерживают множество различных версий одной программы.
- **Директивы #if, #ifdef, #ifndef, #else, #elif и #endif** — *это директивы условной компиляции.*
- Главная идея состоит в том, что если выражение, стоящее после директивы *#if* оказывается истинным, то будет скомпилирован код, расположенный между нею и директивой *#endif* в противном случае данный код будет опущен. Директива *#endif* используется для обозначения конца блока *#if*.
- Общая форма записи директивы *#if* выглядит так.
- #if константное_выражение
- последовательность инструкций
- #endif
- Если *константное_выражение* является истинным, код, расположенный непосредственно за этой директивой, будет скомпилирован.

- // Простой пример использования директивы #if.
- #include <iostream>
- using namespace std;
- #define MAX 100
- int main()
- {
- #if MAX>10
- cout << "Требуется дополнительная память\n";
- #endif
- // ...
- return 0;
- }
- При выполнении эта программа отобразит сообщение *Требуется дополнительная память* на экране, поскольку, как определено в программе, значение константы *MAX* больше 10. Этот пример иллюстрирует важный момент: Выражение, которое стоит после директивы *#if*, вычисляется во время компиляции. Следовательно, оно должно содержать только идентификаторы, которые были предварительно определены, или константы. Использование же переменных здесь исключено.

- Поведение директивы *#else* во многом подобно поведению инструкции *else*, которая является частью языка C++: она определяет альтернативу на случай невыполнения директивы *#if*. Чтобы показать, как работает директива *#else*, воспользуемся предыдущим примером, немного его расширив.
- // Пример использования директив #if/#else.
- #include <iostream>
- using namespace std;
- #define MAX 6
- int main()
- {
- #if MAX>10
- cout << "Требуется дополнительная память.\n");
- #else
- cout << "Достаточно имеющейся памяти.\n";
- #endif
- // . . .
- return 0;
- }
- В этой программе для имени *MAX* определено значение, которое меньше 10, поэтому *#if*- ветвь кода не скомпилируется, но зато скомпилируется альтернативная *#else*-ветвь. В результате отобразится сообщение *Достаточно имеющейся памяти.*.
- Обратите внимание на то, что директива *#else* используется для индикации одновременно как конца *#if*-блока, так и начала *#else*-блока. В этом есть логическая необходимость, поскольку только одна директива *#endif* может быть связана с директивой *#if*.

- Директива *#elif* эквивалентна связке инструкций *else-if* и используется для формирования многозвенной схемы *if-else-if*,представляющей несколько вариантов компиляции. После директивы *#elif* должно стоять константное выражение. Если это выражение истинно, следующий блок кода скомпилируется, и никакие другие *#elif*- выражения не будут тестироваться или компилироваться. В противном случае будет проверено следующее по очереди *#elif*-выражение. Вот как выглядит общий формат использования директивы *#elif*.
- #if выражение
- последовательность инструкций
- #elif выражение 1
- последовательность инструкций
- #elif выражение 2
- последовательность инструкций
- #elif выражение 3
- последовательность инструкций
- // . . .
- #elif выражение N
- последовательность инструкций
- #endif

- Например, в этом фрагменте кода используется идентификатор *COMPILED_BY*, который позволяет определить, кем компилируется программа.
- #define JOHN 0
- #define BOB 1
- #define TOM 2
- #define COMPILED_BY JOHN
- #if COMPILED_BY == JOHN
- char who[] = "John";
- #elif COMPILED_BY == BOB
- char who[] = "Bob";
- #else
- char who[] = "Tom";
- #endif

- Директивы *#if* и *#elif* могут быть вложенными. В этом случае директива *#endif*, *#else* или *#elif* связывается с ближайшей директивой *#if* или *#elif*. Например, следующий фрагмент кода совершенно допустим.
- #if COMPILED_BY == BOB
- #if DEBUG == FULL
- int port = 198;
- #elif DEBUG == PARTIAL
- int port = 200;
- #endif
- #else
- cout << "Боб должен скомпилировать код" << "для отладки вывода данных.\n";
- #endif

Директивы #ifdef и #ifndef

- Директивы *#ifdef* и *#ifndef* предлагают еще два варианта условной компиляции, которые можно выразить как *"если определено"*и *"если не определено"*соответственно.
- Общий формат использования директивы *#ifdef* таков.
- #ifdef макроимя
- последовательность инструкций
- #endif
- Если *макроимя* предварительно определено с помощью какой-нибудь директивы *#define*, то *последовательность инструкций*, расположенная между директивами *#ifdef* и *#endif*, будет скомпилирована.
- Общий формат использования директивы *#ifndef* таков.
- #ifndef макроимя
- последовательность инструкций
- #endif
- Если *макроимя* не определено с помощью какой-нибудь директивы *#define*, то *последовательность инструкций*, расположенная между директивами *#ifdef* и *#endif*, будет скомпилирована.
- Как директива *#ifdef*, так и директива *#ifndef* может иметь директиву *#else* или *#elif*.

Директива #undef

- Директива *#undef* используется для удаления предыдущего определения некоторого макроимени. Ее общий формат таков.
- #undef макроимя
- Рассмотрим пример.
- #define TIMEOUT 100
- #define WAIT 0
- // . . .
- #undef TIMEOUT
- #undef WAIT
- Здесь имена *TIMEOUT* и*WAIT* определены до тех пор, пока не выполнится директива
- *#undef*.
- Основное назначение директивы *#undef* — разрешить локализацию макроимен для тех частей кода, в которых они нужны.

Использование оператора defined

- Помимо директивы *#ifdef* существует еще один способ выяснить, определено ли в программе некоторое макроимя. Для этого можно использовать директиву *#if* в сочетании с оператором времени компиляции *defined*. Например, чтобы узнать, определено ли макроимя *MYFILE*, можно использовать одну из следующих команд препроцессорной обработки.
- #if defined MYFILE
- или
- #ifdef MYFILE
- При необходимости, чтобы реверсировать условие проверки, можно предварить оператор *defined* символом *"!"*. Например, следующий фрагмент кода скомпилируется только в том случае, если макроимя *DEBUG* не определено.
- #if !defined DEBUG
- cout << "Окончательная версия!\n";
- #endif

Директива #line

- *Директива #line изменяет содержимое псевдопеременных _ _LINE_ _ и _ _FILE_ _.*
- Директива *#line* используется для изменения содержимого псевдопеременных_ _*LINE*_ _ и_ _*FILE*_ _, которые являются зарезервированными идентификаторами (макроименами). Псевдопеременная_ _*LINE*_ _ содержит номер скомпилированной строки, а псевдопеременная_ _*FILE*_ _— имя компилируемого файла. Базовая форма записи этой команды имеет следующий вид.
- #line номер "имя_файла"
- Здесь *номер* — это любое положительное целое число, а *имя_файла* — любой допустимый идентификатор файла. Значение элемента *номер* становится номером текущей исходной строки, а значение элемента *имя_файла*— именем исходного файла. *Имя_файла*
- — элемент необязательный. Директива *#line* используется, главным образом, в целях отладки и в специальных приложениях.
- Например, следующая программа обязывает начинать счет строк с числа 200. Инструкция cout отображает номер 202, поскольку это — третья строка в программе после директивной инструкции *#line 200*.
- #include <iostream>
- using namespace std;
- #line 200 // Устанавливаем счетчик строк равным 200.
- int main() // Эта строка сейчас имеет номер 200.
- {// Номер этой строки равен 201.
- cout << _ _LINE_ _;// Здесь выводится номер 202.
- return 0;
- }

Директива #pragma

- Работа директивы *#pragma* зависит от конкретной реализации компилятора. Она позволяет выдавать компилятору различные инструкции, предусмотренные создателем компилятора. Общий формат его использования таков.
- #pragma имя
- Здесь элемент *имя* представляет имя желаемой *#pragma*-инструкции. Если указанное имя не распознается компилятором, директива *#pragma* попросту игнорируется без сообщения об ошибке.
- **Важно!** *Для получения подробной информации о возможных вариантах использования директивы #pragma стоит обратиться к системной документации по используемому вами компилятору. Вы можете найти для себя очень полезную информацию. Обычно #pragma-инструкции позволяют определить, какие предупреждающие сообщения выдает компилятор, как генерируется код и какие библиотеки компонуются с вашими программами.*

Операторы препроцессора "#" и "##"

- В C++ предусмотрена поддержка двух операторов препроцессора: *"#"* и*"##"*. Эти операторы используются совместно с директивой *#define*. Оператор*"#"* преобразует следующий за ним аргумент в строку, заключенную в кавычки. Рассмотрим, например, следующую программу.
- #include <iostream>
- using namespace std;
- #define mkstr(s) # s
- int main()
- {
- cout << mkstr(Я в восторге от C++);
- return 0;
- }
- Препроцессор C++ преобразует строку
- cout << mkstr(Я в восторге от C++);
- в строку
- cout << "Я в восторге от C++";

##

- Оператор используется для конкатенации двух лексем. Рассмотрим пример.
- #include <iostream>
- using namespace std;
- #define concat(a, b) a ## b
- int main()
- {
- int xy = 10;
- cout << concat(x, y);
- return 0;
- }
- Препроцессор преобразует строку
- cout << concat (x, y);
- в строку
- cout << xy;

Зарезервированные макроимена

- В языке C++ определено шесть встроенных макроимен.
- _ _LINE_ _
- _ _FILE_ _
- _ _DATE_ _
- _ _TIME_ _
- _ _STDC_ _
- _ _cplusplus
- Рассмотрим каждое из них в отдельности.
- Макросы _ _*LINE*_ _ и_ _*FILE*_ _ описаны при рассмотрении директивы *#line*. Они содержат номер текущей строки и имя файла компилируемой программы.
- Макрос _ _*DATE*_ _ представляет собой строку в формате *месяц/день/год*, которая означает дату трансляции исходного файла в объектный код.
- Время трансляции исходного файла в объектный код содержится в виде строки в макросе _ _*TIME*_ _. Формат этой строки следующий: *часы.минуты.секунды*.
- Точное назначение макроса _ _*STDC*_ _ зависит от конкретной реализации компилятора. Как правило, если макрос_ _*STDC*_ _ определен, то компилятор примет только стандартный C/C++-код, который не содержит никаких нестандартных расширений.
- Компилятор, соответствующий ANSI/ISO-стандарту C++, определяет макрос_ _*cplusplus* как значение, содержащее по крайней мере шесть цифр. "Нестандартные" компиляторы должны использовать значение, содержащее пять (или даже меньше) цифр.