Лекция 5

Структурное программирование
Особенности работы с функциями

Методология структурного программирования

- Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Предложена в 1970-х годах Э. Дейкстрой и др
- В соответствии с данной методологией любая программа строится без использования оператора goto из трёх базовых управляющих структур: последовательность, ветвление, цикл; кроме того, используются подпрограммы.
- При этом разработка программы ведётся пошагово, методом «сверху вниз».
- Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственно, усложнения программного обеспечения.
- **Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование»**.

Метод «сверху вниз»

- Сначала пишется **текст основной программы**, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент.
- Вместо настоящих, работающих подпрограмм, в программу вставляются фиктивные части — ***заглушки***, которые, говоря упрощенно, ничего не делают.
- Если говорить точнее, заглушка удовлетворяет ***требованиям интерфейса*** заменяемого фрагмента (модуля), но не выполняет его функций или выполняет их частично.
- Затем заглушки заменяются или дорабатываются до настоящих ***полнофункциональных фрагментов*** (модулей) в соответствии с планом программирования.
- На каждой стадии процесса реализации уже созданная программа должна правильно работать по отношению к более низкому уровню. Полученная программа проверяется и отлаживается.
- После того, как программист убедится, что подпрограммы вызываются в правильной последовательности (то есть общая структура программы верна), ***подпрограммы-заглушки последовательно заменяются на реально работающие,*** причём разработка каждой подпрограммы ведётся тем же методом, что и основной программы.
- Разработка заканчивается тогда, когда не останется ни одной заглушки.
- Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело ***с обозримым и понятным ему множеством фрагментов***, и может быть уверен, что общая структура всех более высоких уровней программы верна.
- При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры нужно внести изменения. Они вносятся, не затрагивая части программы, непосредственно не связанные с ними.
- Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

Функции.Введение

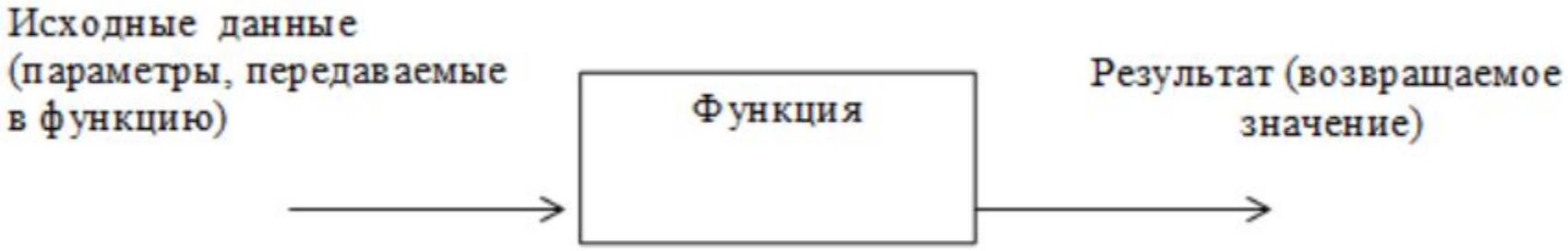
С увеличением объема программы становится невозможно удерживать в памяти все детали. Чтобы уменьшить сложность программы, ее разбивают на части. В C++ задача может быть разделена на более простые подзадачи с помощью функций. Разделение задачи на функции также позволяет избежать избыточности кода, т. к. функцию записывают один раз, а вызывают многократно. Программу, которая содержит функции, легче отлаживать.

Часто используемые функции можно помещать в библиотеки. Таким образом, создаются более простые в отладке и сопровождении программы.

Объявление и определение функций

Функция – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т. д.

Функция, во-первых, является одним из производных типов C++, а, во-вторых, минимальным исполняемым модулем программы.



Любая функция должна быть объявлена и определена.

Объявление функции (прототип, заголовок) задает имя функции, тип возвращаемого значения и список передаваемых параметров.

Определение функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов.

```
тип имя_функции([список_формальных_параметров])
{
тело_функции
}
```

Тело_функции – это блок или составной оператор. Внутри функции нельзя определить другую функцию.

В теле функции должен быть оператор, который возвращает полученное значение функции в точку вызова. Он может иметь две формы:

1) return выражение;

2) return;

Первая форма используется для возврата результата, поэтому выражение должно иметь тот же тип, что и тип функции в определении. Вторая форма используется, если функция не возвращает значения, т. е. имеет тип void. Программист может не использовать этот оператор в теле функции явно, компилятор добавит его автоматически в конец функции перед }.

! Тип возвращаемого значения может быть любым, кроме массива и функции, но может быть указателем на массив или функцию.

Список формальных параметров – это те величины, которые требуется передать в функцию. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя. В объявлении имена можно не указывать.

Для того, чтобы выполнялись операторы, записанные в теле функции, функцию необходимо вызвать. При вызове указываются: имя функции и фактические параметры. Фактические параметры заменяют формальные параметры при выполнении операторов тела функции.

! Фактические и формальные параметры должны совпадать по количеству и типу.

Объявление функции должно находиться в тексте раньше вызова функции, чтобы компилятор мог осуществить проверку правильности вызова. Если функция имеет тип не void, то ее вызов может быть операндом выражения.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

// объявление функции нахождения n!
int faktorial(int numb)// заголовок функции
{
    int rezult = 1; // инициализируем переменную rezult значением 1
    for (int i = 1; i <= numb; i++) // цикл вычисления значения n!
        rezult *= i; // накапливаем произведение в переменной rezult
    return rezult; // передаём значение факториала в главную функцию
}

int main()
{
    setlocale(LC_ALL, "Russian");
    int digit; // переменная для хранения значения n!
    cout << "Введите число : ";
    cin >> digit;
    cout << digit << "! = " << faktorial(digit) << endl;// запуск функции нахождения факториала
    system("pause");
    return 0;
}
```

```
Введите число : 15
15! = 2004310016
Для продолжения нажмите любую клавишу . . .
```

Прототипы функций

В C++ вы не можете вызвать функцию до объявления самой функции. Все потому, что компилятор не будет знать полное имя функции (имя функции, число аргументов, типы аргументов).

```
int main() {
    cout << Sum_numbers(1, 2) << endl;
    system("PAUSE");
    return 0;
}

int Sum_numbers(int a, int b) {
    return a + b;
}
```

Так, при вызове функции Sum_numbers() внутри функции main() компилятор не знает ее полное имя.

Конечно компилятор C++ мог просмотреть весь код и определить полное имя функции, но этого он делать не умеет и нам приходится с этим считаться.

Поэтому мы обязаны проинформировать компилятор о полном имени функции. Для этого мы будем использовать прототип функции.

Прототип функции — это функция, в которой отсутствует блок кода (тело функции). В прототипе функции находятся:

1)Полное имя функции.

2)Тип возвращаемого значения функции.

```
int Sum_numbers(int a, int b);    // прототип функции

int main() {
    cout << Sum_numbers(1, 2) << endl;
    system("PAUSE");
    return 0;
}

int Sum_numbers(int a, int b) {   // сама функция
    return a + b;
}
```

Область видимости переменных

Область видимости переменных — это те части программы, в которой пользователь может изменять или использовать переменные в своих нуждах.

Если переменная была создана в каком-либо блоке, то ее **областью видимости** будет являться этот блок от его начала (от открывающей скобки — {) и до его конца (до закрывающей скобки — }) включая все дочерние блоки созданные в этом блоке.

! Глобальные переменные также можно использовать для передачи данных между функциями, но этого не рекомендуется делать, т. к. это затрудняет отладку программы и препятствует помещению функций в библиотеки. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

Локальные переменные

Переменные, которые используются внутри данной функции, называются локальными. Память для них выделяется в стеке, поэтому после окончания работы функции они удаляются из памяти. Нельзя возвращать указатель на локальную переменную, т. к. память, выделенная такой переменной, будет освобождаться.

```
int* f()
{
int a;
…
return &a;// ОШИБКА!
}
```

Глобальные переменные

Глобальные переменные – это переменные, описанные вне функций. Они видны во всех функциях, где нет локальных переменных с такими именами.

```
int a,b;        //глобальные переменные
void change()
{
     int r;     //локальная переменная
     r=a;
     a=b;
     b=r;
}

void main()
{
     cin>>a,b;
     change();
     cout<<"a="<<a<<"b="<<b;
}
```

Статические переменные

Статическая переменная (или еще «переменная со статической продолжительностью») сохраняет свое значение даже после выхода из блока, в котором она определена. То есть она создается (и инициализируется) только один раз, а затем сохраняется на протяжении всей программы.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

void incrementAndPrint()
{
    int value = 1;
    ++value;
    cout << value << endl;
} // переменная value уничтожается здесь

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();
    system("PAUSE");
    return 0;
}
```

```
2
2
2
```

```
#include "stdafx.h"
#include <iostream>
using namespace std;

void incrementAndPrint()
{
    static int s_value = 1;// переменная s_value - статическая.
    ++s_value;
    cout << s_value << endl;
} // переменная s_value не уничтожается здесь, но становится недоступной
int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();
    system("PAUSE");
    return 0;
}
```

```
2
3
4
```

Параметры функций

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Существует два способа передачи параметров в функцию: по адресу и по значению.

При передаче по значению выполняются следующие действия:

1)Вычисляются значения выражений, стоящие на месте фактических параметров;

2)В стеке выделяется память под формальные параметры функции;

3)Каждому фактическому параметру присваивается значение формального параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

Таким образом, операторы функции работают с копиями фактических параметров. Доступа к самим фактическим параметрам у функции нет, следовательно, нет возможности их изменить.

```
void Change(int a, int b)//передача по значению
{
    int r = a;
    a = b;
    b = r;
}
```

При передаче по адресу в стек заносятся копии адресов параметров, следовательно, у функции появляется доступ к ячейке памяти, в которой находится фактический параметр и она может его изменить.

```
void Change(int* a, int* b) //передача по адресу
{
    int r = *a;
    *a = *b;
    *b = r;
}
```

Для работы с функцией необходимо выполнить следующие этапы

- Описание функции (*Прототип*)
- Вызов функции
- Определение функции

Общая форма определения функции

Тип_функции Имя_функции ([Список_параметров])

{

Операторы; *// Тело_функции*

return [значение]**;**

}

Здесь ***Тип_функции*** определяет тип величины, возвращаемого функцией. Функция может возвращать любой тип за исключением массива. Если функция ничего не возвращает, то тип возврата должен быть void **– пустой***.

Имя_функции – любой допустимый идентификатор.

Список параметров представляет собой **последовательность пар** *типов* и *идентификаторов*, разделяемых запятыми.

Параметры – это переменные, которые получают значения аргументов, передаваемых функции при ее вызове. Если функция не требует параметров, то список параметров будет пуст.

Фигурные скобки окружают **тело функции**. Тело функции состоит из операторов, определяющих, что именно эта функция делает.

*** Функции с типом void аналогичны sub на VBA, procedure – Python, Pascal**

```
#include <iostream>
using namespace std;
void myfunc( ); // прототип функции myfunc
int main( ) // главная функция
{
    myfunc( );  // обращение к функции
    return 0;
}
void myfunc( ) // определение функции
{
    cout << " inside the myfunc" << endl;
    return;      // оператор, осуществляющий возврат
                 // в место вызова функции
}
```

Передача значений в функцию

В функцию можно передать одно или несколько значений.

Значение, передаваемое в вызываемую функцию называется аргументом или фактическим параметром и указывается *в обращении* к функции.

Соответствующие параметры в функции называются формальными параметрами.

Формальные параметры объявляются *в определении* функции.

Составим программу с функцией, которая вычисляет объем коробки задаваемой тремя параметрами: длиной, шириной и высотой.

```
#include <iostream>
using namespace std;
void box(int length, int width, int height);      // прототип функции
int main( ) // главная функция
{
      box(7, 20, 4);   // первое обращение к функции
      box(50, 3, 2);   // второе обращение к функции
      box(8, 6, 9);    // третье обращение к функции
      return 0;
}
void box(int length, int width, int height) // определение функции
{
      cout << " volume = " << length * width * height << endl;
            return; // оператор возврата из функции
}
```

```
volume = 560
volume = 300
volume = 432
```

Возврат одного значения из функции

Функция может вернуть значение в вызывающий ее код*. Возвращаемое значение указывается в операторе **return**:

return значение.

В определении функции должен быть указан ***тип возвращаемого значения*.** Этот тип должен совпадать с типом значения в операторе **return**.

Доработаем программу с функцией для вычисления объема коробки.

В этом варианте **box( )** возвращает вычисленный объем, который является величиной целого типа.
Поэтому функция **box( )** определена как функция целого типа.

*** Функции с возвращаемым значением эквиваленты Function на VBA, Python, Pascal.**

```
#include <iostream>
using namespace std;
int box(int  length,   int  width,   int  height);
int main( )
{
      cout << box(7, 20, 4) << endl;
      cout << box(50, 3, 2) << endl;
      cout << box(8, 6, 9)  << endl;
          int z = box(3,6,4) + box(4,5,8);
      return 0;
}
int box(int  length,   int  width,   int  height)
{
      return   length * width * height;
}
```

Необязательные аргументы функций

```
double expnt (double x, unsigned int e = 2);
int main( )
{
    double y = expnt(3.8);          // здесь рассчитывается 3.8^2
    double x = expnt(2.9, 5);       // здесь рассчитывается 2,9^5
    return 1;
}
//Определение функции:
double expnt (double x, unsigned int e = 2)
{
    double result = 1;
    for (int i = 0; i < e; i++)
        result *= x;
    return result;
}
```

Необязательные аргументы функций: возможные ошибки

попытка определения двух функций

```
double expnt (double x, unsigned int e = 2);
```

```
double expnt (double x);
```

приведет к ошибке компиляции – неоднозначности определения функции. Это происходит потому, что вызов

```
double x = expnt(4.1);
```

подходит как для первой, так и для второй функции

Указатели и ссылки

Взаимоотношение имени переменной и её адреса в ОП

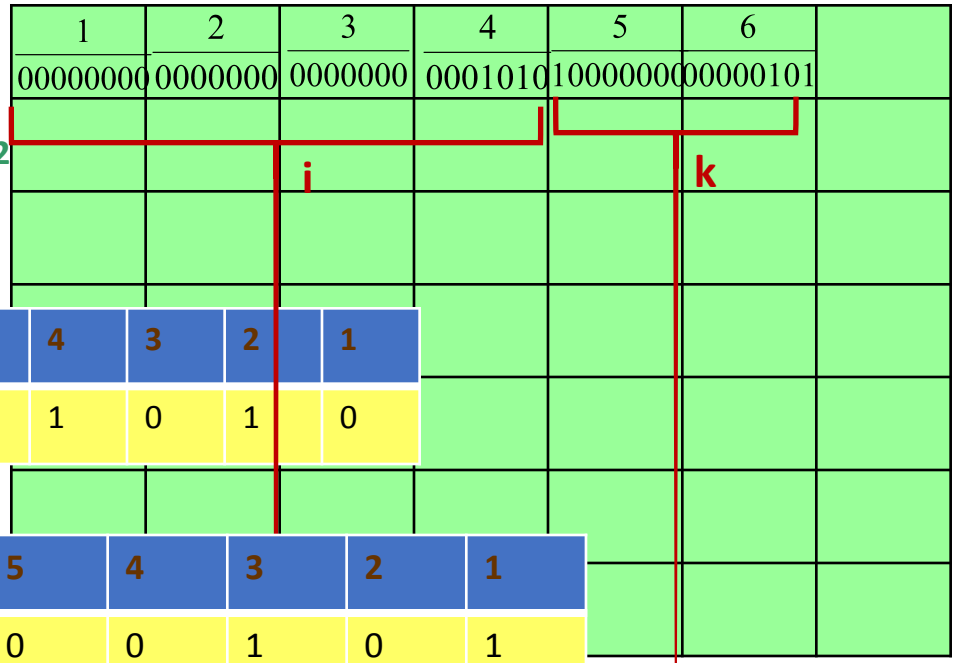
Текст программы

```
int i = 10;
short int k = -5;
```

// $10_{10} = 1010_2$

// $|5_{10}| = 101_2$

Оперативная память, каждая ячейка – 1 байт



32	31	30	…	9	8	7	6	5	4	3	2	1
0	0	0	…	0	0	0	0	0	1	0	1	0

16	15	14	…	8	7	6	5	4	3	2	1
1	0	0	…	0	0	0	0	0	1	0	1

Тип «указатель»

Когда компилятор обрабатывает оператор определения переменной, например int i = 10; он выделяет память в соответствии с типом (int) и инициализирует её указанным значением (10). Все обращения к переменной по её имени (i) *заменяются компилятором на адрес области памяти*, в которой хранится значение переменной.

Программист может определить собственные переменные для хранения адресов областей памяти. **Такие переменные называются *указателями*.**

Указатели*, следовательно, *предназначены для хранения адресов областей памяти.

В C++ различают три вида указателей – указатели на объект, на функцию и на void, отличающиеся свойствами и набором допустимых значений.

Описание указателя

Указатель – это объект, содержащий адрес ***начала*** области памяти, где хранится значение переменной.

Так, если **p** содержит адрес **x**, то говорят, что **p** **указывает** на **x**.

Переменная-указатель *объявляется* следующим образом:

тип *имя_указателя;

Здесь ***тип*** определяет, тип данных на которые будет указывать этот указатель.

Пример:

int * ip; // ip указатель на int

float * fp; // fp указатель на float

Указатель хранит:
1) адрес начала памяти объекта,
2) количество байт, выделенных на объект;
3) способ представления объекта (целочисленный, вещественный, пользовательский и т.п).

Машинный адрес	0012FF48 байт	0012FF49 байт	0012FF4A байт	0012FF4B байт	0012FF54 байт	0012FF55 байт	0012FF56 байт	0012FF57 байт	0012FF63 байт
Значение в памяти	$2.015*10^{-6}$				1937				'G'
Имя	summa				date				ch

Размер указателя зависит от модели памяти, для VS C++ указатель занимает 4 байта, можно определять указатель на указатель, может быть константой или переменной, а также указывать на константу или переменную.

Например:

```
int L;                          // целая переменная
const int ci = L;               // целая константа
int * pi ;                      // указатель на целую переменную
const int * pci;                // указатель на целую константу
int * const cp = & i;           // указатель-константа на целую переменную
const int * const cpc = & ci;   // указатель-константа на целую константу
```

С указателями можно выполнять следующие операции:

- *разыменование* (*) – получение значения величины, адрес которой хранится в указателе;

- взятие адреса (&);

Эта операция позволяет получить объект по адресу, который хранится в указателе.

Указатель p будет ссылаться на адрес, по которому располагается переменная x, напр. на адрес 0x60FE98.

Но так как указатель хранит адрес, то мы можем по этому адресу получить хранящееся там значение, то есть значение переменной x.

Для этого применяется операция * (операция разыменования), то есть та операция, которая применяется при определении указателя.

Результатом этой операции всегда является объект (значение), на который указывает указатель.

- присваивание;

- арифметические операции

- **- сложение указателя только с константой**
- **- вычитание: допускается разность указателей и разность указателя и константы**
- **- инкремент (++) увеличивает значение указателя на величину**

sizeof(тип);

- **- декремент (--) уменьшает значение указателя на величину**

sizeof(тип);

-сравнение;

-приведение типов

Пример на применение операции разыменования

Это означает, что по адресу переменной в памяти мы переходим к действиям над значением, хранимом по данному адресу. Это операция и называется разыменованием.

```
int main()
{
    int  i_val = 7;
    int* i_ptr = &i_val; //Используя унарную операцию
              //взятия адреса &, мы извлекаем адрес переменной
              //i_val и присваиваем ее указателю.

    // выведем на экран значение переменной i_val
    cout <<  i_val << endl; //используем саму переменную
    cout << *i_ptr << endl; //обращаемся к значению переменной
                            // i_val через указатель: здесь используется
                            //операция разыменования:
                            //она позволяет перейти от адреса к значению.
        system("pause");
  return 0;
}
```

Краткие итоги:

- *Для экономии памяти и времени, затрачиваемого на обращение к данным, в программах используют указатели на объекты.*
- *Указатель не является самостоятельным типом, он всегда связан с другим типом.*
- *Указатель может быть константой или переменной, а также указывать на константу или переменную.*
- *Указатель типа void указывает на область памяти любого размера. Разыменование такого указателя необходимо проводить с операцией приведения типов.*
- *До первого использования в программе объявленный указатель необходимо проинициализировать.*
- *Над указателями определены операции: разыменование, взятие адреса, декремент, инкремент, увеличение (уменьшение) на целую константу, разность, определение размера.*
- *Над указателями определены операции сравнения.*

Пример сравнения указателей

```
int x=10;
int y=10;
int *xptr=&x;
int *yptr=&y;
```

При этом сравниваются значения указателей, а не значения величин, на которые данные указатели ссылаются.

```
//сравниваем указатели
 if (xptr == yptr)
   cout << "Указатели равны\n";
    else
   cout << "Указатели не равны\n";
```

```
//сравниваем значения, на которое указывает указатель
 if (*xptr == *yptr) {
   cout << "Значения равны\n";
   } else {
   cout << "Значения неравны\n";}
```

Какие результаты будут на консоли?

Пример демонстрация ситуации, когда указатели различных типов указывают на одно и то же место в памяти. Однако при разыменовании получаются разные результаты.

```
// Выбор данных из памяти с помощью разных указателей
// Использование функций приведения типов

unsigned long L=12345678;
  char *cp=(char*)&L;
  int *ip=(int*)&L;
  long *lp=(long*)&L;

  cout <<"\n&L = "<<&L;
  cout <<"\nL = "<<L;
  cout <<"\n*cp = "<<*cp;
  cout <<"\n*ip = "<<*ip;
  cout <<"\n*lp = "<<*lp;

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

```
&L = 012FFBF8
L = 12345678
*cp = N
*ip = 12345678
*lp = 12345678
```

```
&L = 00CFFBD0
L = 102345678
*cp = ╬
*ip = 102345678
*lp = 102345678
```

```
&L = 007FFE84
L = 12345678
*cp = N
*ip = 12345678
*lp = 12345678
```

```
&L = 00CFFB94
L = 12345678
*cp = N
*ip = 12345678
*lp = 12345678
```

Докажите, что для L=12345678 *cp="N"

Ключевые термины

- **Адрес объекта** – это *адрес* области оперативной памяти, по которому хранится *объект* в соответствии с особенностями представления типа.
- **Косвенная адресация** – это обращение к области памяти не напрямую, по адресу, а через *объект*, которому в памяти соответствует определенный участок.
- **Разыменование** – это операция получения значения объекта, *адрес* которого хранится в указателе;
- **Указатель** – это именованный *объект*, предназначенный для хранения адреса области памяти.
- **Указатель на константу** – это *указатель* на такой *объект*, *значение* которого нельзя изменить в процессе выполнения программы.
- **Указатель-константа** – это *указатель*, *значение* которого нельзя изменить в процессе выполнения программы.
- **Указатель-константа на константу** – это *указатель*, для которого невозможно изменение как самого указателя, так и значения адресуемого объекта.

Контрольные вопросы

1. Почему указатель не может существовать как самостоятельный тип?
2. С какой целью в программе может быть использован указатель типа `void`?
3. Как изменится значение указателя после применения к нему операции инкремента (декремента)?
4. Почему для указателей определены сложение и вычитание только с целыми константами?
5. В чем отличие указателя на константу от указателя-константы?
6. Два указателя разных типов указывают на одно и то же место в памяти. Сравните результаты операций разыменования и взятия адреса с такими указателям. Сравните значения указателей.
7. Если объект занимает в памяти несколько байтов, то какой адрес является значением указателя на этот объект?
8. Каким образом при разыменовании указателей становится известно, сколько байтов памяти доступно?

Тип «ссылка»

Ссылки представляют собой *синоним имени*, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

Тип & Имя;

где Тип это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним Имя является именем переменной ссылочного типа, например:

int kol;	
int & pal = kol;	**// ссылка pal - альтернативное имя для kol**
const char& CR = ' \ n ';	**// ссылка на константу**

Правила работы со ссылками

- Переменная-ссылка должна явно инициализироваться при ее описании, кроме некоторых случаев, когда она является, например, параметром функции.
- После инициализации ссылке не может быть присвоена другая переменная.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений.

ПРИМЕР: программа выполняет последовательность описанных выше операций.

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

int main( )
{
    int  total;
    int  *ptr;      // указатель на int
    int  val;
    total = 3200;  // присвоим total значение 3200
    ptr = &total;   // получим адрес total
    val = *ptr;     // получим значение по этому адресу
    cout  <<  "val  =  "   << val   << '\n';
    return 0;
}
```

Возврат нескольких значений из функции

Для того чтобы и в вызывающей программе и в функции работать с одной и той же переменной необходимо осуществлять передачу в функцию адреса памяти, где размещена данная переменная.

Такая передача называется **передачей по ссылке** (вместо передачи по значению). Это достигается с помощью **параметра-ссылки.**

Для этого в определении функции и в прототипе перед именем соответствующей переменной необходимо поставить знак операции **&**, возвращающей адрес переменной, перед которой она указана.

При использовании параметра-ссылки в функцию передается адрес (а не значение) аргумента.

Внутри функции при операциях над параметром-ссылкой автоматически выполняется снятие ссылки, поэтому нет необходимости указывать при аргументе оператор &.

В следующей программе в функции вычисляются объем и

```
#include <iostream>
using namespace std;
void box(int  length,  int  width,  int  height,  int  &vol,  int &ar);
int main( )
{     int volume;
      int area;
      box(7, 20, 4, volume, area); cout << volume << "  " << area << endl;
      box(50, 3, 2, volume, area);  cout << volume << "  " << area << endl;
      box(8, 6, 9, volume, area);    cout << volume << "  " << area << endl;
      return 0;
}
void box(int  length,  int  width,  int  height,  int   &vol,  int &ar)
{
      vol = length * width * height;
      ar = length * width;
      return;
}
```

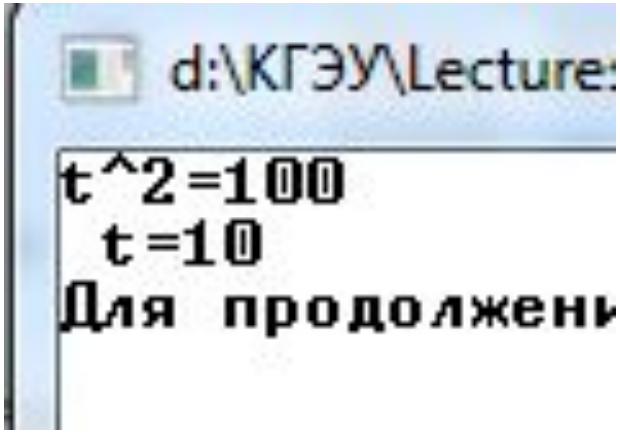
Резюме:

При передаче фактических аргументов *по значению* в вызываемой функции создаются *копии* передаваемых значений. Поэтому любые операции с соответствующими формальными параметрами в теле функции *не изменят фактические значения в вызывающей функции*.

При передаче *по ссылке*, вызываемая функция принимает адрес той переменной, которая описана в вызывающей программе, поэтому все операции в теле функции *приводят к изменению значения переменной*, переданной в качестве фактического значения.

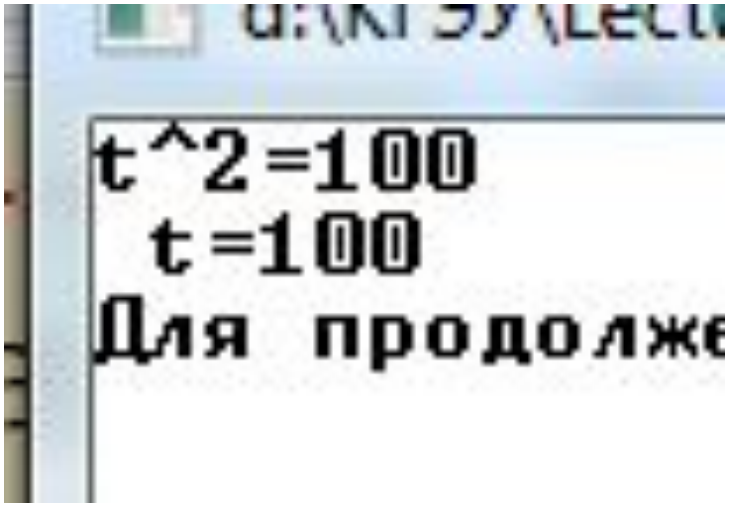
Пример передачи параметров по значению

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

int sqr (int  x);
int main(void)
{
int t=10;
cout<<"t^2="<<sqr(t)<<endl;
cout<<" t=" <<t<<endl;
system("pause");
return 0;
}
int sqr (int  x) {
x = x*x; return x;
}
```



Пример передачи параметров по ссылке

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

int sqr (int & x);
int main(void)
{
int t=10;
cout<<"t^2="<<sqr(t)<<endl;
cout<<" t=" <<t<<endl;
system("pause");
return 0;
}
int sqr (int & x) {
x = x*x; return x;
}
```



Задача: 1. **Написать функцию Power234(A, B, C, D), вычисляющую вторую, третью и четвёртую степени числа A и возвращающую эти степени соответственно в переменных B, C, D. Все параметры вещественные. Найти все степени пяти любых чисел.**

```
 9  #include <iostream>
10
11  using namespace std;
12  void Power234(int A, int &B, int &C, int &D);
```

```
x = 4 x^2 = 16 x^3 = 64 x^4 = 256
x = 7 x^2 = 49 x^3 = 343 x^4 = 2401
x = 8 x^2 = 64 x^3 = 512 x^4 = 4096
x = 6 x^2 = 36 x^3 = 216 x^4 = 1296
x = 4 x^2 = 16 x^3 = 64 x^4 = 256
```

Задача: 2. **Даны два вектора с координатами {1,-2,0}, {2, 7,-4}. Найти модуль каждого вектора, сумму векторов и их скалярное произведение.**

```
int main()
{
    int a,b,c, x,y,z, u,v,w;//переменные для координат3-х векторов
    float mod_1, mod_2, scal_pr;// модули векторов и их скалярное произведение
cout<<" Input 3 coordinates of the vector:";cin >>a>>b>>c;
cout<< "Vector: "<<'\t'<<a<<'\t'<<b<<'\t'<<c<<endl;
cout<<" Input 3 coordinates of the vector:";cin >>x>>y>>z;
cout<< "Vector: "<<'\t'<<x<<'\t'<<y<<'\t'<<z<<endl;
```

```
mod_1= sqrt(float(a*a+b*b+c*c));// модуль первого вектора
mod_2= sqrt(float(x*x+y*y+z*z));// модуль второго вектора
     cout<<"\nmod_1 = "<< mod_1<<endl;
     cout<<"\nmod_2 = "<< mod_2<<endl;

     u=a+x; v=b+y;w= c+z;// определение суммы векторов
    cout<<”\n new”;
    cout<< "Vector: "<<'\t'<<u<<'\t'<<v<<'\t'<<w<<endl;


     scal_pr=(a*x+b*y+c*z)/mod_1/mod_2;
    cout<< "\nscal_pr: "<<scal_pr<<endl;
system("pause");
     return 0;
}
     scal_pr=(a*x+b*y+c*z)/mod_1/mod_2;
   cout<< "\nscal_pr: "<<scal_pr<<endl;
system("pause");
     return 0;
}
```

Решение задачи в рамках структурного подхода

```
#include <iostream>
#include <cmath>
using namespace std;
 void inp_vect(int & a, int & b, int & c)// функция для
      ввода вектора с консоли
 {cout<<"\n Input 3 coordinates of the vector:";cin >>a>>b>>c;}
 void print(int a, int b, int c)// функция для вывода
      вектора на консоль
 {cout<< "\nVector: "<<'\t'<<a<<'\t'<<b<<'\t'<<c<<endl;}
 float modul(int a, int b, int c)// функция  вычисления
                 модуля вектора
 {return sqrt(float(a*a+b*b+c*c));}
```

```
 int main()
{
int a,b,c, x,y,z, u,v,w; //переменные для координат3-х
      векторов
float mod_1, mod_2, scal_pr; // модули векторов и их скалярное
                произведение


      inp_vect(a,b,c); print(a,b,c); // ввод и вывод векторов
      inp_vect(x,y,z); print(x,y,z);

      mod_1= modul(a,b,c); // модуль первого вектора
      mod_2= modul(x,y,z); // модуль первого вектора

      cout<<"\nmod_1 = "<< mod_1<<endl;
      cout<<"\nmod_2 = "<< mod_2<<endl;

u=a+x; v=b+y;w= c+z; // определение суммы векторов
    cout<<"\n new "; print(u,v,w);
```

```
scal_pr=(a*x+b*y+c*z)/(modul(a,b,c)*modul(x,y,z)); // обращение к функции внутри вычисляющего оператора

	cout<< "\nscal_pr: "<<scal_pr<<endl;
system("pause");
	return 0;
}
```

```
 Input 3 coordinates of the vector: 1 -2 0
Vector:          1       -2       0
 Input 3 coordinates of the vector: 2 7 -4
Vector:          2       7        -4
mod_1 = 2.23607
mod_2 = 8.30662
 new
Vector:          3       5        -4
scal_pr: -0.646058
Для продолжения нажмите любую клавишу . . .
```

Задания для самостоятельной работы

Задача: 2. Написать процедуру Mean(X,Y,Amean,Gmean), вычисляющую средне арифметическое Amean =(X+Y)/2 и среднее геометрическое Gmean = двух положительных чисел X, Y. С помощью функции найти среднее арифметическое и среднее геометрическое для пар (A,B), (A,C), (A,D), если заданы A,B,C,D.

Задача: 3. Написать функцию DigitCountSum(K, S), находящую количество цифр целого положительного числа K, а также их сумму S. Применить процедуру к пяти случайным числам K, содержащим не менее 5 и не более 9 цифр. Для определения числа цифр в числе создать соответствующую функцию.