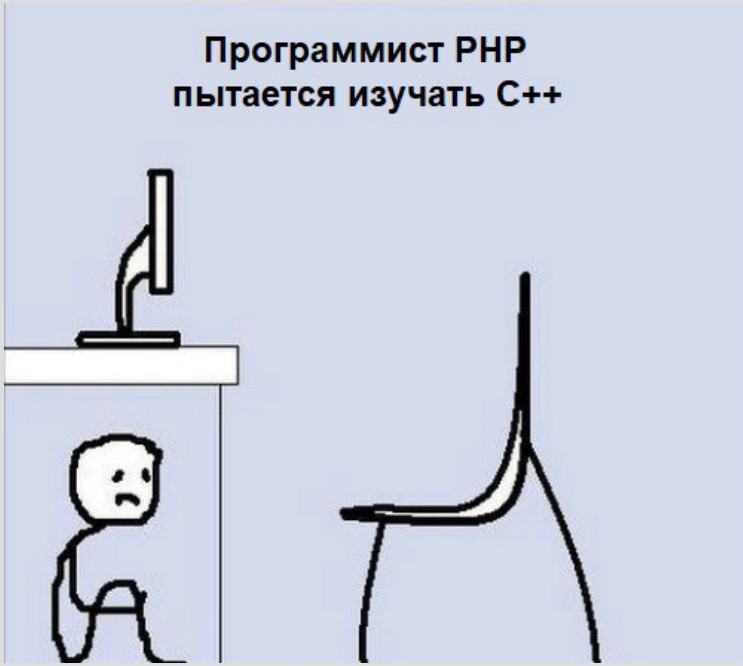
Основы C++. Вебинар №3.

Длительность: 1.5 - 2 ч.

Что будет на уроке

- Рассмотрим все операции, доступные программисту на языке C++.
- Подробно изучим битовые операции.
- Изучим указатели в C++.
- Узнаем в чём разница между ссылкой и указателем.
- Научимся арифметике указателей.

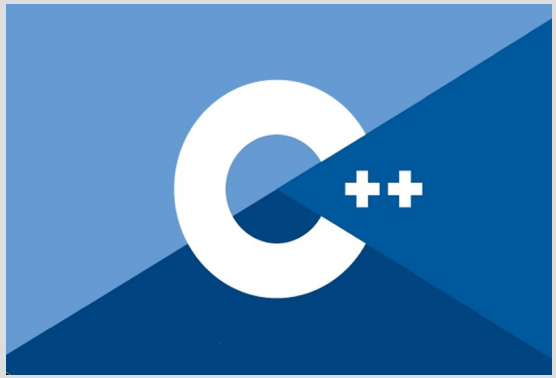


Операторы и операции в C++

В C++ большое количество разных операций или операторов. Операции делятся на **унарные**, **бинарные** и **тернарную** по количеству участвующих в них операндов.
В C++ **только один** тернарный оператор: ? :

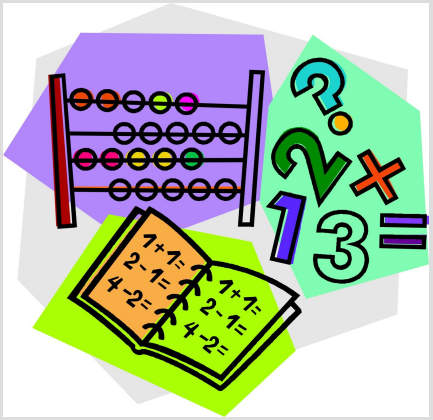
- Арифметические (+, -, /, *, ++, --, %).
- Побитовые (|, &, ^, ~, <<, >>).
- Логические (&&, ||, !, >, <, ==, !=, <=, >=).
- Присваивания (=, +=, -=, *=, /=, %=, <<=, >>=, |=, &=, ^=).
- Специальные (new, delete, throw, sizeof, typeid, static_cast, dynamic_cast, const_cast, reinterpret_cast, ::, -> и др.).

Подробнее и полный список всех операторов:
http://cpp-cpp.blogspot.com/2013/10/c_4.html



Арифметические операции

- Тривиальные: +, -, *, /
- Более интересные: ++, --, %



Взятие по модулю или остаток от деления (%):

```
int a = 30 % 27;
std::cout << a << std::endl;  // напечатает 3

a = 30 % 10;
std::cout << a << std::endl;  // напечатает 0

a = 30 % 9;
std::cout << a << std::endl;  // напечатает 3
```

Инкремент (++) и декримент (--) увеличивает или уменьшает целочисленную переменную на 1 но бывают двух видов: префиксный (до переменной) и постфиксный (после переменной).

```
int a = 10, b = 20;
a++;      // после этой строки a будет 11
++a;      // после этой строки a будет 12
b = a++; // b будет 12, a будет 13
b = ++a; // b будет 14, a будет 14
```

То есть префексный инкремент **сначала увеличивает** значение a **потом использует** его в выражении, а постфиксный инкремент действует наоборот **сначала использует** значение переменной в выражении, а **потом увеличивает** эту переменную.

Побитовые операции

Побитовые операции используются для работы с целыми числами на низком уровне (маски, криптография и пр.)
& - побитое «И», конъюнкция
| - побитовое «ИЛИ», дизъюнкция
~ - побитовое «НЕ»
^ - сложение по модулю 2 (исключающее «ИЛИ», XOR, «сумма по модулю 2»)
<< - побитовый сдвиг числа в сторону старших разрядов (увеличивает целое число в 2 раза).
>> - побитовый сдвиг числа в сторону младших разрядов (уменьшает целое число в 2 раза).

Операцию XOR часто используют в шифровании.

Побитовые сдвиги (<<, >>) можно использовать для очень быстрого умножения и деления на 2.
Обычная операция умножения и деления требует много тактов CPU.

Примеры использования:

```
int a, b = 0;
int c = 0b0000'1111; // 15, 0xF

a = b & c;
std::cout << a << std::endl;   // 0
a = b | c;
std::cout << a << std::endl;   // 15
a = b ^ c;
std::cout << a << std::endl;   // 15
a = ~c;
std::cout << a << std::endl;   // -16 неожиданно?
a = c << 1;
std::cout << a << std::endl;   // 30
```

Побитовые операции

A	B	A И B	A ИЛИ B	A исключающее ИЛИ B	НЕ A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Логические операции

Логические операции (&&, ||, !, >, <, ==, !=, <=, >=) применяются в логических условиях if или при инициализации переменных типа bool.

Примеры:

```
int a = 10, b = 20;
if (a > 0 && b > 0)       // Если a > 0 и b > 0 то напечатай текст:
{
    std::cout << "a and b > 0" << std::endl;
}

if (a > 0 || b > 0)       // Если a > 0 или b > 0 то напечатай текст:
{
    std::cout << "a or b > 0" << std::endl;
}

if (a != b)               // Если a не равно b то напечатай текст:
{
    std::cout << "a is not equal b" << std::endl;
}

const bool AnotEqualB = !(a == b);       // true
```

Подробнее про условия if будет на следующих вебинарах.

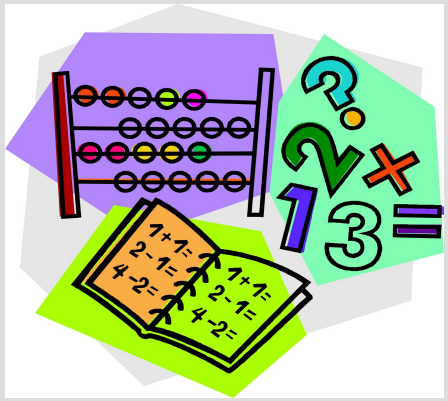
a	b	&&
0	0	0
0	1	0
1	0	0
1	1	1

a	b	||
0	0	0
0	1	1
1	0	1
1	1	1

a	!
0	1
1	0

Операции присваивания

- Операции присваивания в C++: =, +=, -=, *=, /=, %=, <<=, >>=, |=, &=, ^=.
- Операция = тривиальная
- Сокращенные варианты (+=, -=, *=, /=, %=, <<=, >>=, |=, &=, ^=).



```
int a = 100;
a = a + 5;  // можно написать короче a += 5;
a = a - 5;  // можно написать короче a -= 5;
a = a * 5;  // можно написать короче a *= 5;
a = a / 5;  // можно написать короче a /= 5;

a = a & 5;  // можно написать короче a &= 5; (побитовое И)
a = a << 2;  // можно написать короче a <<= 2; (побитовый сдвиг, умножение на 4)
```

Тернарный оператор ? :

Он напоминает логическое условие if
И имеет 3 секции:

```
X =  (условие)  ?  значение1  :  значение2;
```

```
int a = 100, b;

b = (a > 50)  ?  a  :  -a;          // b = 100, так как условие истинно (выбирается первое значение)
std::cout << b << std::endl;


b = (a > 150)  ?  a  :  -a;         // b = -100, так как условие ложно (выбирается второе значение)
std::cout << b << std::endl;
```

В ДЗ будет одно задание на использование этого оператора

Указатели в C++

Указатель это переменная которая хранит адрес другой переменной.

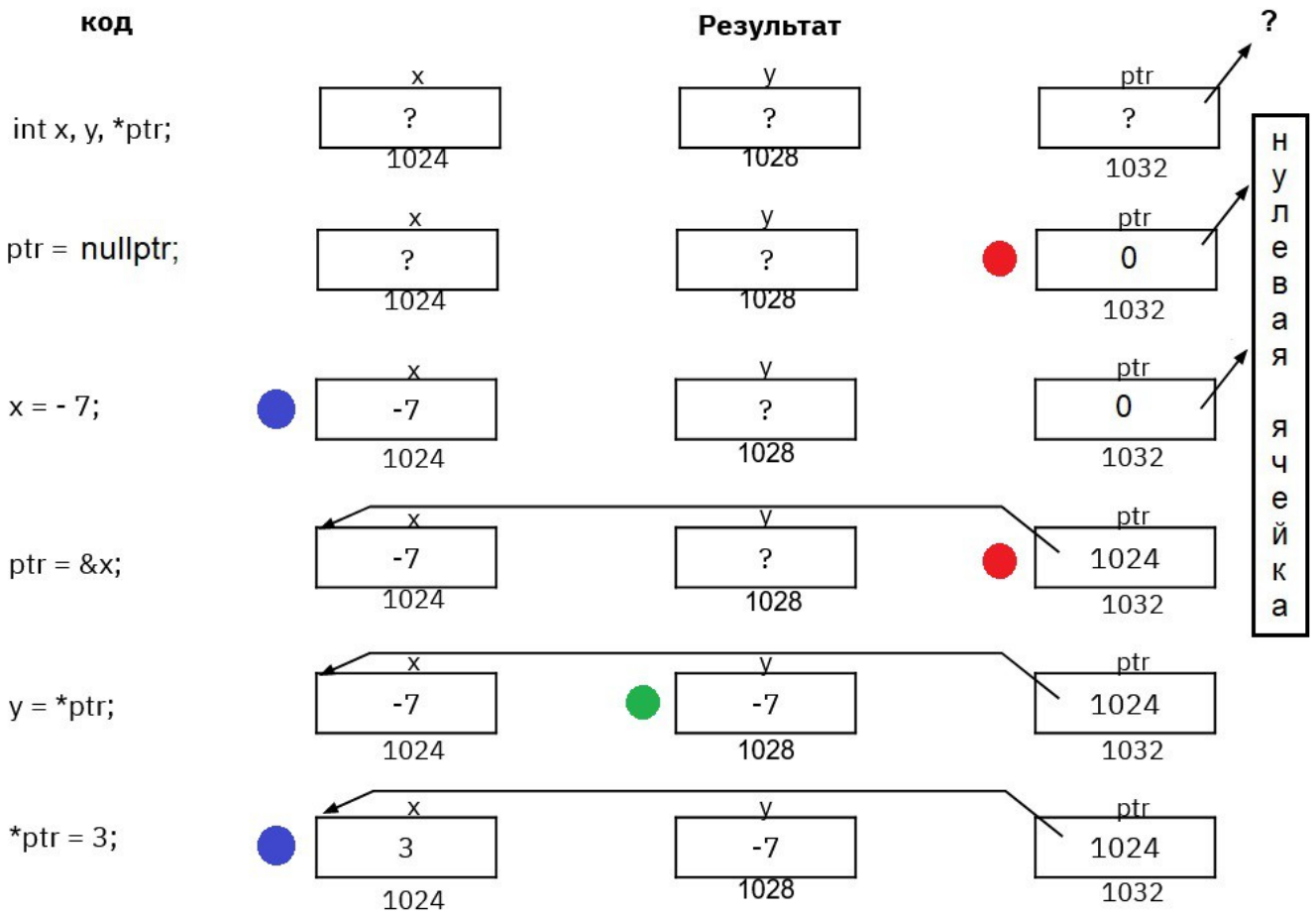
Для работы с указателями используются операции * и &.

```
int main() {

    int x, y, * ptr;  // объявляем 3 переменные

    ptr = nullptr;  // инициализируем указатель null, нулевым значением
                    // на всякий случай =)
    x = -7;

    ptr = &x;   // адрес переменной x записываем в переменную ptr

    y = *ptr;   // Записываем в y значение на которое указывает указатель ptr

    *ptr = 3;   // Записываем в ячейку (x) на которую ссылается указатель ptr число 3

    std::cout << "x = " << x << " y = " << y;  // вывод на экран x = 3 y = -7

    return 0;
}
```

Инициализатор nullptr для указателей появился в стандарте C++11.
До него использовался макрос NULL.

Указатели в С++

Графическое пояснение операции & и *



А какой размер в байтах занимает указатель?

При сборке своего проекта вы можете выбирать 32-х разрядную сборку она называется еще x86 или 64-х разрядную. Во втором режиме вы можете адресовать гораздо больше оперативной памяти (RAM).

В зависимости от этого указатель будет занимать 4 или 8 байтов.

Этот вопрос иногда задают на собеседованиях.
Теперь вы к нему готовы =)

Указатель на структуру

У любой переменной можно взять ее адрес (операция &) и сохранить его в указателе, в том числе и у структуры.

```
struct Employee  {  // Новый тип данных Сотрудник
    long id;                    // ID сотрудника
    unsigned short age;    // его возраст
    double salary;             // его зарплата
};

int main() {

    Employee em1 = { 1234567, 30, 17'000.0 };   // переменная Сотрудник

    Employee * ptrEm = & em1;    // настроем указатель на переменную em1;

    (*ptrEm).age = 31;          // обновим в структуре поле age

    ptrEm->id = 9876543;    // обновим в структуре поле id

    return 0;
}
```

У нас есть 2-а способа обращаться к полям структуры через указатель (через * и .) или через (->). Второй вариант более предпочтителен.

Указатель на массив

Можно настроить указатель на массив, и тогда нам доступна арифметика указателей (+, -, ++, --).

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

int main() {
    int Array [5], * pArr;

    pArr = & Array[0];  // адрес нулевого элемента сохраняем в указателе pArr
    *pArr = 10;          // изменяем первый элемент массива

    pArr = pArr + 1;    // увеличиваем указатель на 1 (чтоб указывал на 2й элемент массива)
    *pArr = 20;          // изменяем второй элемент массива

    pArr += 1;           // увеличиваем указатель на 1 (чтоб указывал на 3й элемент массива)
    *pArr = 30;          // меняем третий элемент массива

    pArr++;              // увеличиваем указатель на 1 с помощью операции инкремент
    *pArr = 40;         // меняем четвертый элемент массива

    pArr = & Array[4];  // настраиваем на пятый элемент массива
    *pArr = 50;          // меняем пятый элемент массива через указатель

    cout << Array[0] << endl << Array[1] << endl << Array[2] << endl << Array[3] << endl << Array[4];
    return 0;
}
```

Индексация указателей

Имя массива на самом деле является константным указателем. Поэтому для указателей тоже применима операция индексации (квадратные скобки).

```
#include <iostream>

using namespace std;   // разрешаем использовать пространство std во всем файле

int main()
{
    int Array [5];
    int * pArr = NULL;  // в старых проектах вы можете встретить NULL при инициализации указателей
                        // но после стандарта C++11 рекомендуется использовать nullptr

    pArr = & Array[0];  // адрес нулевого элемента сохраняем в указателе pArr
    pArr[0] = 10;       // изменяем первый элемент массива
    pArr[1] = 20;       // изменяем второй элемент массива
    pArr[2] = 30;       // меняем третий элемент массива
    pArr[3] = 40;       // меняем четвертый элемент массива
    pArr[4] = 50;       // меняем пятый элемент массива через указатель

    cout << Array[0] << endl << Array[1] << endl << Array[2] << endl << Array[3] << endl << Array[4];
    return 0;
}
```

Как видно мы можем работать с указателем как с обычным массивом если мы его на массив настроили.

Константные указатели

1. Указатель на константные данные

```
int a, b;
const int * ptr = & a;
*ptr = 10; // ОШИБКА компиляции нельзя менять константные данные
ptr = & b;  // но можно менять сам адрес на который указывает указатель
```

2. Константный указатель (нельзя менять сам указатель, но можно менять данные)

```
int a, b;
int * const ptr = & a;
*ptr = 20;   // записали в переменную a 20
ptr = & b;   // ОШИБКА компиляции (нельзя менять константный указатель)
```

3. Константный указатель на константные данные

```
int a, b;
const int * const ptr = & a;
*ptr = 10;   // ОШИБКА компиляции нельзя менять константные данные
ptr = & b;   // ОШИБКА компиляции (нельзя менять константный указатель)
```

В программировании, в реальных проектах **чаще всего встречается 1-й вариант**, указатель на константные данные.

Отличие указателя от ссылки (reference)

Ссылка это тоже адрес переменной как и указатель. Но
1) мы работаем с ссылкой как с обычной переменной по имени, не требуется разыменование.
2) ссылка обязана быть инициализирована при объявлении в отличи от указателя, который можно инициализировать позднее.

```
int a = 1000;
int & refA = a;  // ссылка на переменную a
refA = 20;       // изменяем переменную a через ссылку на нее
                 // не требуется * как с указателями

std::cout << a;  // вывод на экран 20
```

В C++ присутствует **определенная путаница** с операциями, усложняющая изучение:

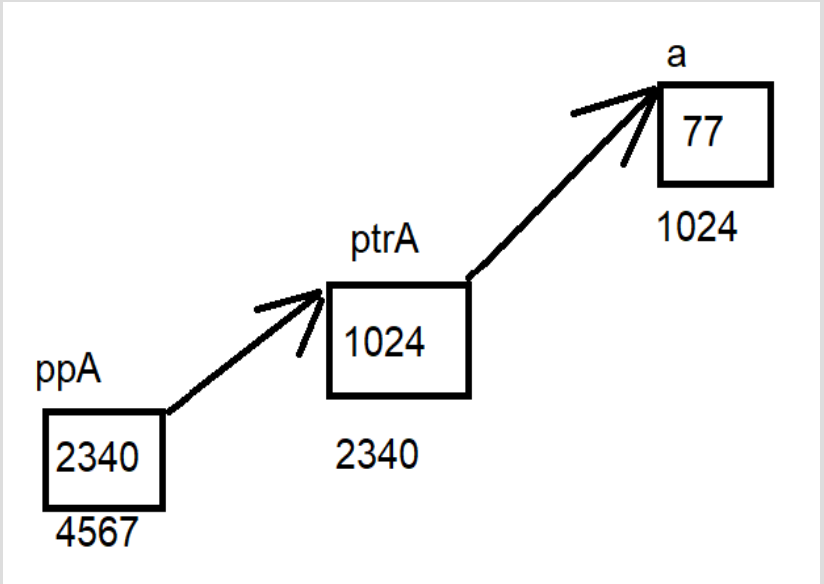
Операция * - используется
1) при объявлении указателя.
2) для разыменования указателя, чтоб перейти на то место куда он указывает.

Операция & - используется
1) при взятии адреса у переменной чтоб сохранить его в указатель.
2) при объявлении ссылки (см выше).

Указатель на указатель

Можно объявлять указатели на указатели:

```
int a = 77;
int *ptrA = &a;
int** ppA = &ptrA;

*ptrA = 88;
std::cout << a << std::endl; // 88

**ppA = 99;
std::cout << a << std::endl; // 99
```



Где применяются указатели?

Указатели применяют:

- Для возврата нескольких значений из функции. В качестве аргумента передаётся указатель на переменную, функция записывает туда значение. Такой подход очень распространён в DirectX, OpenGL, Windows API и других библиотеках в стиле C. Для этого можно использовать и ссылки, но не рекомендуется, так как синтаксис передачи и возврата неотличим.

- Для хранения адреса динамически выделенной памяти. Она отличается от обычной тем, что программист сам регулирует время жизни объектов, и её больше (а размер стека всего порядка мегабайта). Если адрес будет потерян, то память нельзя будет ни использовать, ни освободить. Возникнет утечка памяти.

- C-строка представляет собой указатель на её первый символ, а также имена массивов.

- Для создания различных структур данных: связанных списков, деревья и т. д.

- Для передачи аргумента в функцию без копирования (и вызова конструктора для объектов), которое может оказаться долгим для сложных объектов. Правда, здесь лучше использовать константные ссылки.

Таким образом, применений указателей очень много.

Получаем доступ к другой единице трансляции

С помощью ключевого слова extern можно получить доступ к глобальным переменным объявленным в другом cpp файле. Для функций объявленных в другом модуле использовать extern не нужно они и так доступны из main.

foo.cpp

```
int a = 1000;
float b = 10.0;
```

main.cpp

```
extern int a;     // говорим что будем использовать переменную a из другого модуля
extern float b;  // хотим иметь доступ к b из другой единицы трансляции

int main() {

    a = 2000;
    b = 20.0;
    std::cout << a << " " << b << std::endl;

    return 0;
}
```

В ДЗ будет одно задание на доступ к переменным в другом cpp файле.

Приведение (изменение) типа данных - cast

Проблема:

```
int a = 20, b = 30;
float c = b / a;
std::cout << c;   // вывод на экран 1
```

На экране мы увидим 1 так как 30 делится на 20 целочисленным образом хоть переменная c и объявлена как дробная (float).

1. Решение проблемы — привести переменную b перед делением к типу float.

```
int a = 20, b = 30;
float c = static_cast < float > (b) / a;        // static_cast один из 4 операторов
                                                // приведения типа в C++
std::cout << c;   // вывод на экран 1.5
```

2. Или можно использовать C-Style cast из языка C:

```
int a = 20, b = 30;
float c = float (b) / a;        // или можно вот так (float)b
std::cout << c;                  // вывод на экран 1.5
```

В ДЗ будет целочисленное деление, лучше аналогичным образом привести к float чтоб правильно его выполнить.

STL тип данных для строк

В библиотеке STL (Standard Template Library) есть очень удобный и дружелюбный тип данных string.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name, message;  // Очень удобный тип данных string в отличии от char str [255]

    cout << "Hi, guy! Enter your name: ";
    cin >> name;                                  // Считываем с клавиатуры имя пользователя

    message = "Have a good day, " + name;  // такие строки можно даже складывать

    cout << message << endl;    // Выводим на экран сообщение

    return 0;
}
```

Поскольку string это STL тип данных то он тоже как и cout требует перед собой std:: в том случае если вы не использовали ранее using namespace std.

Страшные rvalue и lvalue

В C++11 появились понятия **rvalue** и **lvalue**.

Отличный вопрос на собеседовании чем они отличаются и что обозначают. =)

Иногда программисты отвечают, что rvalue (right) справа от знака равенства а lvalue слева (left), **но это не верно**!

lvalue — то что имеет свой свой адрес в памяти, свое местоположение, с которым мы работаем через имя этой переменной.

```
int a, b;
a = b;     // и a и b это lvalue так как имеют
           // конкретное место в RAM (на стеке или в сегменте данных) для хранения значения

a = 100;  // 100 это rvalue, временное значение
           // (нет своего места в памяти в которое мы можем записать что-то другое).
```

rvalue — это то, что не lvalue. =)

Домашнее задание

1. Написать программу, вычисляющую выражение a * (b + (c / d)) и выводящую результат с плавающей точкой, где a, b, c, d – целочисленные константы. Используйте static_cast или C-Style cast к float чтобы выполнить точное деление.

2. Дано целое число. Если оно меньше или равно 21, то выведите на экран разницу между этим числом и числом 21. Если же заданное число больше, чем 21, необходимо вывести удвоенную разницу между этим числом и 21. При выполнении задания следует использовать тернарный оператор (?:).

3. * Описать трёхмерный целочисленный массив, размером 3x3x3 и указатель на значения внутри массива и при помощи операции разыменования вывести на экран значение центральной ячейки получившегося куба [1][1][1].

4. ** Написать программу, вычисляющую выражение из первого задания, а переменные для неё объявлены и инициализировать в другом cpp файле. Используйте extern.

Предлагаю использовать 2-а проекта в IDE. Первые 3 задания в проекте номер 1. 4-е задание в отдельном проекте. Или можно 1-н проект если вы сделаете 2, 3, 4 задание в нем. То есть если вы сделали 4-е задание, то 1-е можно не делать. =)

Основы C++. Вебинар №3.

Успеха с домашним заданием!

GeekBrains