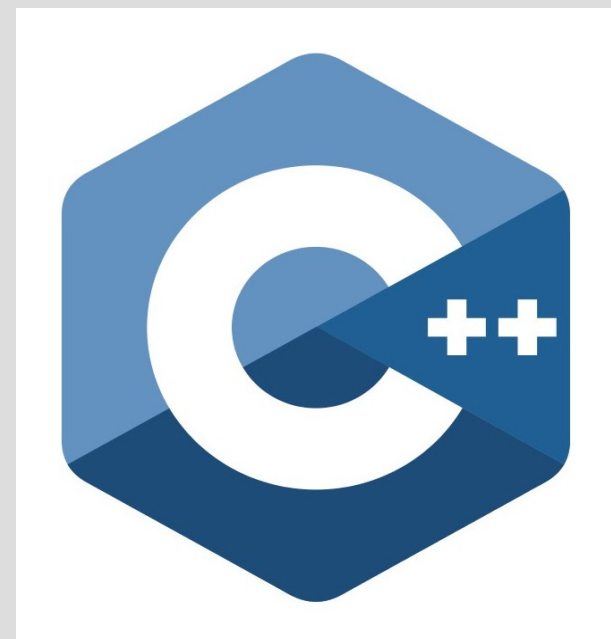
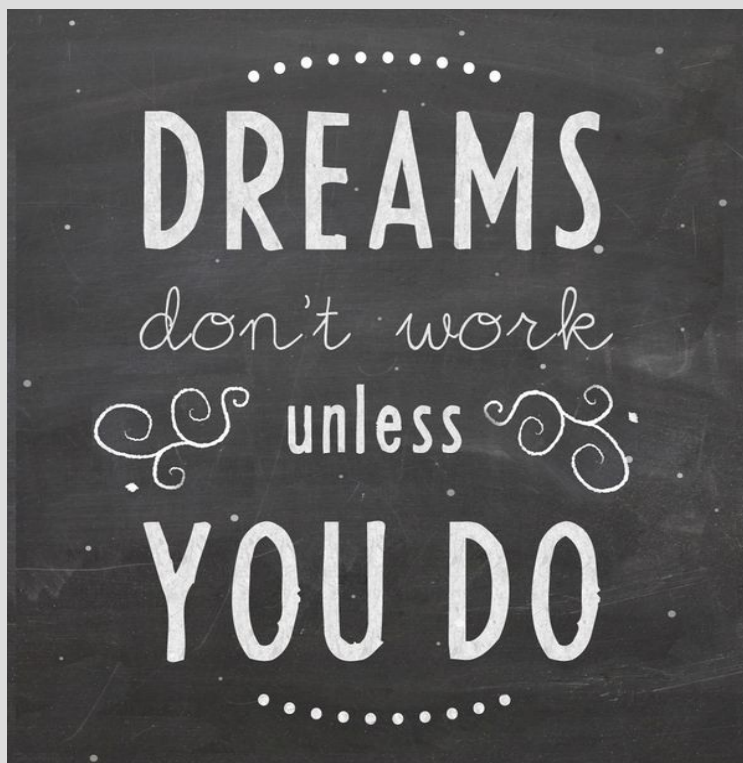


# Основы C++. Вебинар №7.

Длительность: 1-2 ч.



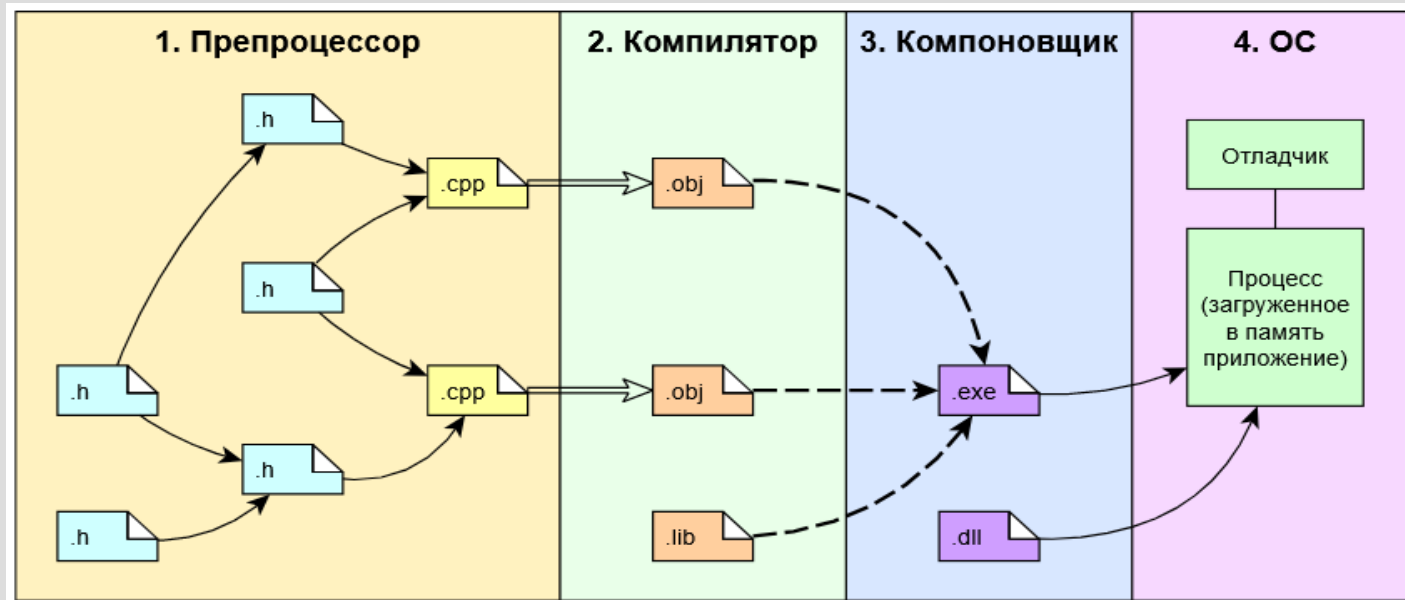
GeekBrains

## Что будет на уроке?

- Познакомимся с понятием директивы препроцессора.
- Изучим дополнительные возможности препроцессора.
- Научимся писать функциональные макросы.
- Узнаем об условной компиляции.



# Трансляция программы на C++



1. **Преппроцессинг** - это этап на котором исполняются так называемые директивы преппроцессора, (include, define и пр.).

Еще иногда этап 2 разделяют на  
2.1 **Компиляция** cpp в assembler.

2.2 **Ассемблирование** - преобразования ассемблерного кода в машинные коды. На этом этапе создаётся объектный файл (имеющий расширение \*.o или \*.obj).

3. **Компоновка** или **линковка** (linker) - линкер объединяет несколько объектных файлов и статических библиотек (\*.lib) в одно исполнимое приложение (\*.exe).

# Директивы препроцессора

Стандартные директивы препроцессора, реализованные во всех компиляторах, в алфавитном порядке:

`#define`

`#elif`

`#else`

`#error`

`#endif`

`#if`

`#ifdef`

`#ifndef`

**`#include` — уже использовали**

`#line`

`#pragma`

`#undef`



# Использование функций из других CPP модулей, заголовочные файлы

foo.cpp

```
#include <iostream>

void printArray(const int array[], int size) // Реализация функции (вывод массива на экран)
{
    for (int i = 0; i < size; i++)
        std::cout << array[i] << " ";
}
```

foo.h

```
#pragma once // include guard – защита от повторного включения заголовочного файла

void printArray(const int array[], int size); // Прототип функции
```

main.cpp

```
#include <iostream>
#include "foo.h"

int main(int argc, char* argv[])
{
    std::cout << "Invoke printArray fun" << std::endl;
    const int size = 4;
    int arr[size] = { 1, 2, 3, 4 };
    printArray(arr, size); // Вызываем функция из другого модуля
    return 0;
}
```





# namespaces в других модулях и h файлах

```
foo.cpp
#include <iostream>
namespace MyLib
{
    void printArray(const int array[], int size) // Реализация функции
    {
        for (int i = 0; i < size; i++)
            std::cout << array[i] << " ";
    }
}
```

```
foo.h
#pragma once
namespace MyLib
{
    void printArray(const int array[], int size); // Прототип функции
}
```

main.cpp

```
#include <iostream>
#include "foo.h"

int main(int argc, char* argv[])
{
    std::cout << "Invoke printArray fun" << std::endl;
    const int size = 4;
    int arr[size] = { 1, 2, 3, 4 };
    MyLib::printArray(arr, size); // Вызываем функция из другого модуля
    return 0;
}
```



# Директива #define

Директиву #define можно использовать для создания макросов. **Макрос** — это правило, которое определяет конвертацию идентификатора в указанные данные.

Есть два основных типа макросов: макросы-функции и макросы-объекты.

**Макросы-функции** ведут себя как функции и используются в тех же целях. Но их использование, как правило, считается опасным, и почти всё, что они могут сделать, можно осуществить с помощью простой C++ функции.

**Макросы-объекты** можно определить одним из следующих двух способов:

#define идентификатор

Или:

#define идентификатор текст\_замена

Верхнее определение не имеет никакого текст\_замена, в то время как нижнее — имеет. Поскольку это директивы препроцессора (а не простые C++ стейтменты), то ни одна из форм не заканчивается точкой с запятой.

# Макросы-функции

Макросы быстрее чем C++ функции

НО!

Они не подлежат отладке и увеличивают размер вашей программы.

И это просто механическая замена аргументов, без проверки их типов и пр. которую обычно делает компилятор.

Поэтому использование C++ функций более безопасно.

```
20
21 #define ABS(x) ((x) < 0 ? -(x) : (x))
22
23 int main(int argc, char* argv[])
24 {
25     std::cout << ABS(-5) << " "
26               << ABS(10) << std::endl;
27     return 0;
28 }
29
```

```
1  #include <stdio.h>
2
3  #define SUM(x, y) (x + y)
4
5  int main(int argc, char *argv[])
6  {
7      int a = 5;
8      int b = 10;
9      int sum = SUM(a, b);
10     printf("%d\n", sum);
11 }
```

Этот код преобразуется в следующий:

```
1
2  /* обработанный код опущен */
3
4  int main(int argc, char *argv[])
5  {
6      int a = 5;
7      int b = 10;
8      int sum = (a + b);
9      printf("%d\n", sum);
10 }
```

```
#define PRINT_LINE(str) std::cout << str << std::endl;

int main()
{
    PRINT_LINE("My str");
    PRINT_LINE(1000);
    return 0;
}
```



# Директива #define

ARRAY\_SIZE — играет такую же роль как и константы в C++. Но использование последних предпочтительнее чем директива препроцессора define

```
#include <iostream>

#define ARRAY_SIZE 5

int main(int argc, char* argv[])
{
    int arr[ARRAY_SIZE] = { 1, 3, 5, 7, 9 };

    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        std::cout << arr[i] << " ";
    }

    return 0;
}
```

Тут показан пример define с двумя параметрами.  
Где же используется define с одним?

# Условная компиляция

Директивы препроцессора условной компиляции позволяют определить, при каких условиях код будет компилироваться, а при каких — нет:

```
#ifdef  
#ifndef  
#endif
```

Директива **#ifdef** (сокр. от «if defined» = «если определено») позволяет препроцессору проверить, было ли значение ранее определено с помощью директивы **#define**. Если да, то код между **#ifdef** и **#endif** скомпилируется. Если нет, то код будет проигнорирован.

Директива **#ifndef** (сокр. от «if not defined» = «если не определено») — это полная противоположность к **#ifdef**, которая позволяет проверить, не было ли значение ранее определено.

```
#include <iostream>  
  
#define PRINT_JOE  
  
int main(int argc, char* argv[]) {  
  
#ifdef PRINT_JOE  
    std::cout << "Joe" << std::endl;  
#endif  
  
#ifdef PRINT_BOB  
    std::cout << "Bob" << std::endl;  
#endif  
  
    return 0;  
}
```

# Использование функций из других CPP модулей, заголовочные файлы (другой тип гарды)

**foo.cpp**

```
#include <iostream>

void printArray(const int array[], int size) // Реализация функции
{
    for (int i = 0; i < size; i++)
        std::cout << array[i] << " ";
}
```

**foo.h**

```
#ifndef H_FOO //
#define H_FOO // include guard – защита от повторного подключения этого h файла

void printArray(const int array [ ], int size); // Прототип функции

#endif
```

**main.cpp**

```
#include <iostream>
#include "foo.h" // Подключаем заголовочный файл другого модуля

int main()
{
    std::cout << "Invoke printArray fun" << std::endl;
    const int size = 4;
    int arr [size] = { 1, 2, 3, 4 };
    printArray(arr, size); // Вызываем функция из другого модуля
    return 0;
};
```

# Выравнивание полей в структурах

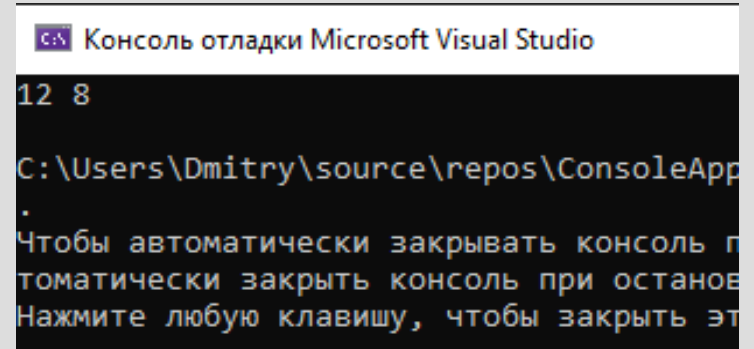
## #pragma pack

```
#include <iostream>

struct MyStruct1 {
    char var1;    // 1 byte
    short var2;   // 2 bytes
    char var3;    // 1 byte
    int var4;     // 4 bytes
};

#pragma pack(push, 1)
struct MyStruct2 {
    char var1;    // 1 byte
    short var2;   // 2 bytes
    char var3;    // 1 byte
    int var4;     // 4 bytes
};
#pragma pack(pop)

int main(int argc, char* argv[])
{
    std::cout << sizeof(MyStruct1) << " "
              << sizeof(MyStruct2) << std::endl;
    return 0;
}
```



# Предопределенные макроподстановки

Могут быть полезны при логировании или отладке:

- `__DATE__` - Дата компиляции файла в формате mm dd yyyy, например, Aug 24 2020.
- `__TIME__` - Время компиляции файла в формате hh:mm:ss.
- `__FILE__` - Имя компилируемого файла.
- `__LINE__` - Номер текущей строки исходного файла (целочисленная константа).
- `__func__` - имя функции в которой вы находитесь (есть аналог более поздний `__FUNCTION__`).
- `_WIN64`, `_WIN32` - константа индикатор сборки в среде Windows.
- `LINUX`, `__linux__` - константа-индикатор сборки Linux.
- `__APPLE__` - константа обозначающая любое устройство от Apple.

```
#ifdef _WIN64
    std::cout << "Win64" << std::endl;
#else
    std::cout << "Win32" << std::endl;
#endif
```

```
int main(int argc, char* argv[])
{
    std::cout << __func__ << std::endl
              << __FILE__ << std::endl
              << __LINE__ << std::endl
              << __DATE__ << std::endl
              << __TIME__ << std::endl;
    return 0;
}
```

Консоль отладки Microsoft Visual Studio

main

C:\Users\Dmitry\source\repos\ConsoleApplication1\ConsoleApplication1\ConsoleApplication1.cpp

28

Feb 13 2021

21:52:57

C:\Users\Dmitry\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe (процесс 31420)

# Многострочные макросы

```
#include <iostream>

#define QUOTATION "Каждый день – чудо. А ведь так и есть, если принять во внимание, \
каким огромным и насыщенным может стать любое мгновение \
нашего хрупкого существования. Пауло Коэльо"

int main(int argc, char* argv[])
{
    setlocale(LC_ALL, "Russian");

    std::cout << QUOTATION << std::endl;

    return 0;
}
```



## Директива #error

Является своего рода аналогом контрольной точки для поиска ошибок при отладке программы. А именно: приводит к выводу компилятором диагностического сообщения которое включает любой текст, указанный в директиве. Если это возможно, процесс компиляции должен приостановиться.

Директива #error используется, как правило, в сочетании с “директивами условий” #if, #elif, #else, #endif, #ifdef, #ifndef.

Синтаксис директивы предельно прост:

```
#ifndef _WIN64
#error "Необходимо скомпилировать программу на 64-разрядной ОС"
#endif
```

## Директива #undef

Мы рассмотрели применение директивы препроцессора #define. Директива #undef отменяет заданное определение #define.

Если вы хотите использовать некоторое имя, но не уверены в том, что оно не было определено ранее, на всякий случай его определение можно попытаться отменить.

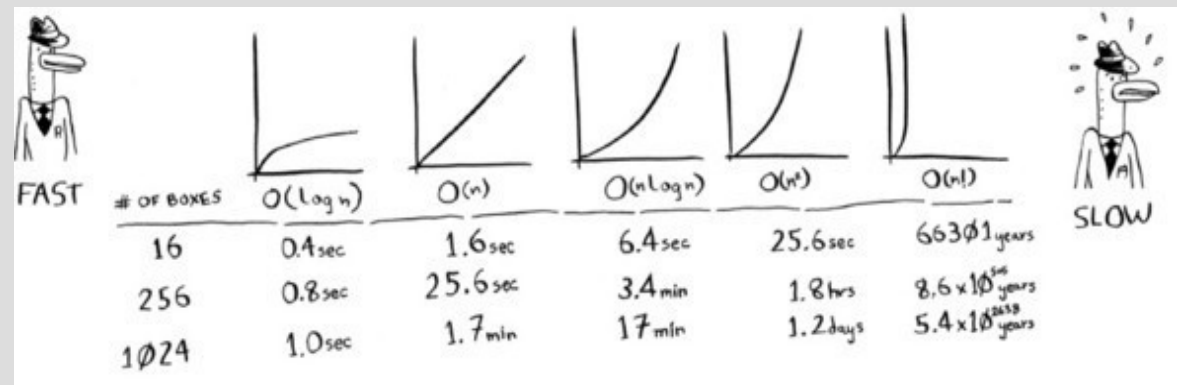
```
10
11 #define QUOTATION "Каждый день – чудо. А ведь так и есть, если принять во внимание, \
12   каким огромным и насыщенным может стать любое мгновение \
13   нашего хрупкого существования. Пауло Коэльо"
14
15 #undef QUOTATION
16
17 int main(int argc, char* argv[])
18 {
19     setlocale(LC_ALL, "Russian");
20     std::cout << QUOTATION << std::endl;
21
22     return 0;
23 }
24
```

# Оценка сложности алгоритмов — функция «О» большое

**Нотация «О» большое** используется для выражения скорости алгоритма. «О» большое говорит вам, насколько быстр ваш алгоритм. Предположим, у вас есть список с размером  $n$  (т. е., у вас  $n$  элементов в этом списке). Простому поиску нужно проверить каждый элемент, поэтому ему понадобится произвести  $n$  операций. Время работы этого алгоритма, записанное при помощи нотации «О» большого, составляет  $O(n)$ .

Некоторые известные алгоритмы:

- Перебор одномерного массива (один цикл) — сложность  $O(n)$ .
- Перебор двумерного массива (два вложенных цикла) — сложность  $O(n^2)$ .
- Сортировка пузырьком (два вложенных цикла) — сложность  $O(n^2)$ .
- Быстрая сортировка (Quick Sort) — сложность: в худшем случае  $O(n^2)$ , в среднем  $O(n \log n)$ .
- Пирамидальная сортировка (Heap Sort) — сложность и в худшем и в среднем  $O(n \log n)$ .
- Двоичное дерево поиска — сложность вставки, поиска, удаления  $O(\log n)$ .
- Черно-красное дерево — сложность вставки, поиска, удаления  $O(\log n)$ .



# Метасинтаксические переменные

**Метапеременные** — это слова-заменители, которые применяются в технических текстах для обозначения чего-либо, что может стоять на их месте. Метапеременные часто используются в программировании.

**foo** — часто используется как первая метапеременная, для обозначения неопределённого (пока) объекта: функции, процесса, и т. п.

**bar** — используется для ссылки на второй неопределённый объект в обсуждении. Например, «функция foo вызывает функцию bar» или «функция foo(bar)».

**baz** — каноническая третья метапеременная, после foo и bar.

**quux** — каноническая четвёртая метапеременная.

```
void foo()
{
    // код
}
```

```
void bar()
{
    // код
}
```

# Процедура код ревью

**Рецензирование кода, обзор кода, ревизия кода (англ. code review) или инспекция кода (англ. code inspection)** — систематическая проверка исходного кода программы другими опытными программистами (коллегами) с целью обнаружения и исправления ошибок, которые остались незамеченными в начальной фазе разработки. Целью обзора является улучшение качества программного продукта и совершенствование навыков разработчика.


В процессе инспекции кода могут быть найдены и устранены такие проблемы, как ошибки в форматировании строк, состояние гонки, утечка памяти и переполнение буфера, что улучшает безопасность программного продукта. Системы контроля версий дают возможность проведения совместной инспекции кода. Кроме того, существуют специальные инструментальные средства для совместной инспекции кода.

**Atlassian Crucible (рус. тигель)** — приложение для совместного просмотра кода, созданное австралийской компанией Senqua, позже приобретенная Atlassian в 2007 году. Проект сохранился и получил дальнейшее развитие. Как и другие продукты компании, Crucible — это веб-приложение, предназначенное в первую очередь для предприятия, которое позволяет осуществлять экспертную оценку программного кода.

Это приложение специально создано для **распределённых команд**, и облегчает **асинхронную инспекцию и комментирование** программного кода. Crucible также интегрируется с популярными системами управления версиями, такими как Git и Subversion. Crucible не является открытым программным обеспечением, но клиенты могут просматривать и модифицировать код для собственного использования.

Crucible	
	
Тип	Code review
Разработчик	Senqua (до 2007), Atlassian (с 2007)
Написана на	Java
Операционная система	Cross-platform
Последняя версия	4.5.1 <sup>[1]</sup> (27 ноября 2017; 3 года назад)
Лицензия	Proprietary
Сайт	<a href="http://www.atlassian.com/crucible">www.atlassian.com/crucible</a>

# Процедура код ревью



CR-FE-8851 109

Details

Objectives

General Comments

Number of files included: 28

FE-git

src

bundled-plugins/bundled/crucible-branch-review

src

main

java/com/atlassian/crucible/plugins/branch-review

model

ao

rest

trackedbranch

impl

resources

BranchReviewInfoAO.java

BranchReviewInfo.java 1

BranchReviewStore.java 1

BranchReviewRestResource.java

TrackedBranchRest.java 8

TrackedBranchRestBuilder.java

DefaultTrackedBranchesManager.java

TrackedBranchJson.java

TrackedBranchSerializer.java

TrackedBranch.java 1

TrackedBranchBuilder.java

TrackedBranchesManager.java

TrackedBranchesSearchCriteria.java

BranchReviewService.java 17

/src/.../trackedbranch/TrackedBranchesSearchCriteria.java Added 4

e0a7a4c

32 return new TrackedBranchesSearchCriteriaBuilder();

33 }

34 }

35 public static TrackedBranchesSearchCriteria withReview(String reviewPermaId) {

36 return builder().reviewPermaId(reviewPermaId).build();

Piotr Swiecicki

providing null/empty reviewPermaId results in criteria matching all reviews, I'd rather expect precondition failure in such circumstances

Add to favourites · Create issue · 27 Aug 14

Cezary Zawadka

Null is ok as we search for all tracked branches regardless review - auto update feature

However non null perma id means we should return empty list if there is no review with the perma id - changed to be handled at ReviewPropertiesManager level

Create issue · 29 Aug 14

Piotr Swiecicki

fine for reviewPermaId property, but if client calls withReview method to build criteria, I'd assume he wanted to filter by particular perm id and was not expecting to pass null.

Create issue · 29 Aug 14

Cezary Zawadka

My mistake - for withReview() precondition makes sense

Create issue · 29 Aug 14

37 }

38 }

39 public static class TrackedBranchesSearchCriteriaBuilder {

40 private String reviewPermaId;

20



## Социальная сеть для поиска работы - LinkedIn

Главная

Сеть

Вакансии

Сообщения

Уведомления

Профиль

Для работы

1 месяц Premium бесплатно

**Företagslån på 1 minut** - Flexibel kredit utan dolda avgifter och bindningstider. Ansök med BankID.
 Реклама

Satya Nadella · 3-й

CEO at Microsoft

Редмонд, Вашингтон, Соединенные Штаты Америки ·  
 8 787 009 отслеживающих · Контактная информация

Отслеживать

...

Общие сведения

As CEO of Microsoft, I define my mission and that of my company as empowering every person and every organization on the planet to achieve more.

Действия

8 787 009 отслеживающих

Excel formulas, the world's most popular programming language, is now Turing...

Satya поделился(ась) этим

7 172 реакции · 188 комментариев

Today, we announced new capabilities to ensure every organization has the tools ...

Satya поделился(ась) этим

Rohita Joshi

2-й  
 Entrepreneur / CEO / Diversity & Inclusion...

Jennifer Lopez

3-й и выше  
 Actor | Writer | Director | Producer | Entrepreneur

Anthony J James

3-й  
 Генеральный директор: инновации и глобальны...

Ryan Roslansky

3-й  
 CEO at LinkedIn

Richard Branson

Your dream job is closer than you think

See jobs

LinkedIn

Другие участники также смотрели

Rohita Joshi

2-й  
 Entrepreneur / CEO / Diversity & Inclusion...

Jennifer Lopez

3-й и выше  
 Actor | Writer | Director | Producer | Entrepreneur

Anthony J James

3-й  
 Генеральный директор: инновации и глобальны...

Ryan Roslansky

3-й  
 CEO at LinkedIn

Richard Branson

Сообщения

Поиск сообщений

Tatyana Penkina

Вы: «спасибо, но в данный момент наверное я не ищу...»

Наталья Рябкова

Вы: «Спасибо, Наталья»

Анастасия Андрианова

5 февр.  
 Анастасия: «Хороших выходных!»

Kseniya Milyutina

21 янв.

Julia Bredikhina

20:00

Дмитрий, добрый день! Меня зовут Юлия, я HRG специалист компании СофИТ. Разработка программно-аппаратного комплекса фото/видео фиксации. Ищем в команду C++ программиста senior или strong middle. Вилка 150-200K. Радуюсь обсудить с вами вакансию!

Спасибо, Julia

OK

👍

Напишите сообщение...

Отправить

# Статические анализаторы кода

**Synopsys Coverity** — пакет программного обеспечения, состоящий из статического и динамического анализаторов кода, принадлежащий компании Synopsys. Программное обеспечение ищет ошибки и недочёты в безопасности исходных кодах программ, написанных на Си, C++, Java, C# и JavaScript.

Использование:

- Согласно контракту с Департаментом национальной безопасности США, при помощи Coverity проверили более 150 проектов с открытым исходным кодом на наличие ошибок, было исправлено более 6 тысяч ошибок в 53 проектах.
- Администрация национальной безопасности дорожного движения США использовала инструмент в 2010 - 2011 годах, расследуя сообщения о непреднамеренном ускорении автомобилей Toyota.
- ЦЕРН использовал инструмент, проверяя программное обеспечение Большого адронного коллайдера.
- Лаборатория реактивного движения НАСА использовала ПО для тестирования исходных кодов марсохода Curiosity.

Coverity	
Основание	ноябрь 2002
Причина упразднения	Acquired by <a href="#">Synopsys</a>
Расположение	<a href="#">San Francisco, CA</a>
Ключевые фигуры	Andreas Kuehlmann (SVP & GM)
Отрасль	<a href="#">Development testing</a>
Продукция	Coverity Code Advisor, Coverity Code Advisor on Demand, Coverity Scan, Coverity Test Advisor, Seeker
Число сотрудников	250+
Сайт	<a href="#">coverity.com</a>



## Домашнее задание

1. Создайте проект из 2х cpp файлов и заголовочного (main.cpp, mylib.cpp, mylib.h) во втором модуле mylib объявить 3 функции: для инициализации массива (типа float), печати его на экран и подсчета количества отрицательных и положительных элементов. Вызывайте эти 3-и функции из main для работы с массивом.
2. Описать макрокоманду (через директиву define), проверяющую, входит ли переданное ей число (введенное с клавиатуры) в диапазон от нуля (включительно) до переданного ей второго аргумента (исключительно) и возвращает true или false, вывести на экран «true» или «false».
3. Задайте массив типа int. Пусть его размер задается через директиву препроцессора #define. Инициализируйте его через ввод с клавиатуры. Напишите для него свою функцию сортировки (например Пузырьком). Реализуйте перестановку элементов как макрокоманду SwapINT(a, b). Вызывайте ее из цикла сортировки.
4. \* Объявить структуру Сотрудник с различными полями. Сделайте для нее побайтовое выравнивание с помощью директивы pragma pack. Выделите динамически переменную этого типа. Инициализируйте ее. Выведите ее на экран и ее размер с помощью sizeof. Сохраните эту структуру в текстовый файл.
5. \* Сделайте задание 1 добавив свой неймспейс во втором модуле (первое задание тогда можно не делать).



## Основы C++. Вебинар №7.

Успеха с домашним заданием!



**GeekBrains**