Препроцессор и макросы

Максим Бакиров
C++ - Разработчик в Яндекс

Проверка связи

Поставьте “+”, если меня видно и слышно

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включен звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Максим Бакиров

О спикере:

- В C++ разработке с 2017 года
- С 2019 года работает в команде разработки Яндекс Браузера

Вспоминаем прошлое занятие

Вопрос: как создать новый файл кода в Visual Studio?

Вспоминаем прошлое занятие

Вопрос: как создать новый файл кода в Visual Studio?

Ответ: щёлкнуть правой кнопкой мыши по проекту в Обозревателе решений, выбрать Добавить -> Новый элемент

Вспоминаем прошлое занятие

Вопрос: зачем нужен заголовочный файл?

Вспоминаем прошлое занятие

Вопрос: зачем нужен заголовочный файл?

Ответ: для объявления глобальных переменных, функций и классов (структур)

Вспоминаем прошлое занятие

Вопрос: как подключить заголовочный файл?

Вспоминаем прошлое занятие

Вопрос: как подключить заголовочный файл?

Ответ: с помощью директивы #include

Вспоминаем прошлое занятие

Вопрос: как определить члены класса, объявленные в заголовочном файле, в файле исходного кода?

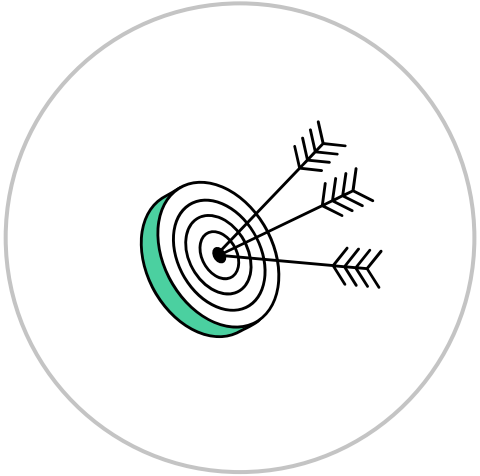
Вспоминаем прошлое занятие

Вопрос: как определить члены класса, объявленные в заголовочном файле, в файле исходного кода?

Ответ: с помощью названия класса, оператора области видимости :: и названия члена

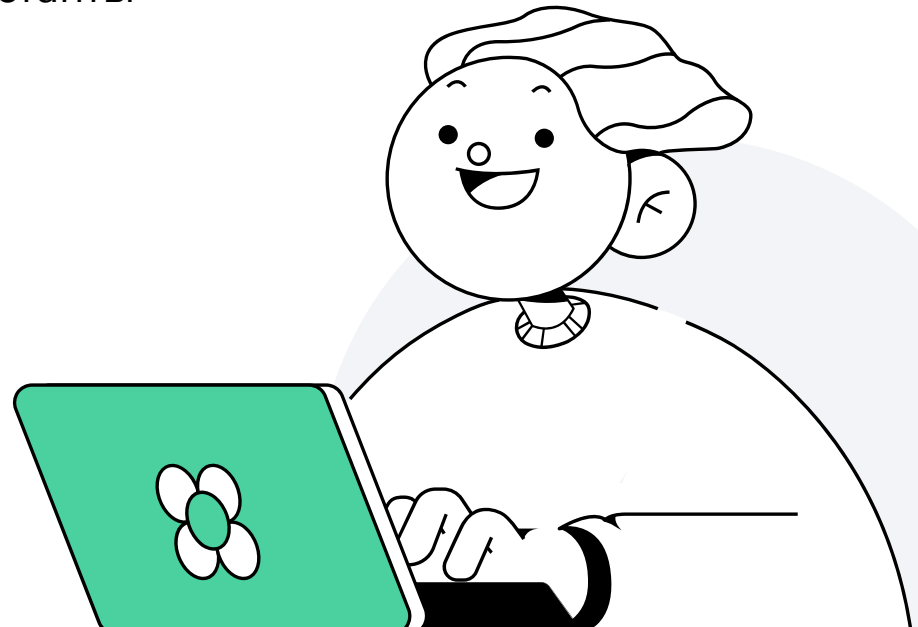
Цели занятия

- Разберёмся, как собирается программа
- Познакомимся с препроцессорными директивами
- Узнаем, что такое макросы
- Выясним, какие бывают предопределённые символьные константы



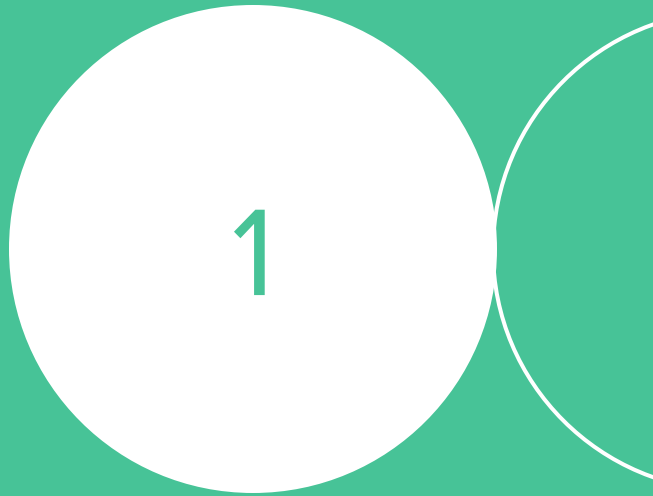
План занятия

1. Препроцессор
2. Макросы
3. Предопределённые символьные константы
4. Итоги
5. Домашнее задание



*Нажми на нужный раздел для перехода

Препроцессор

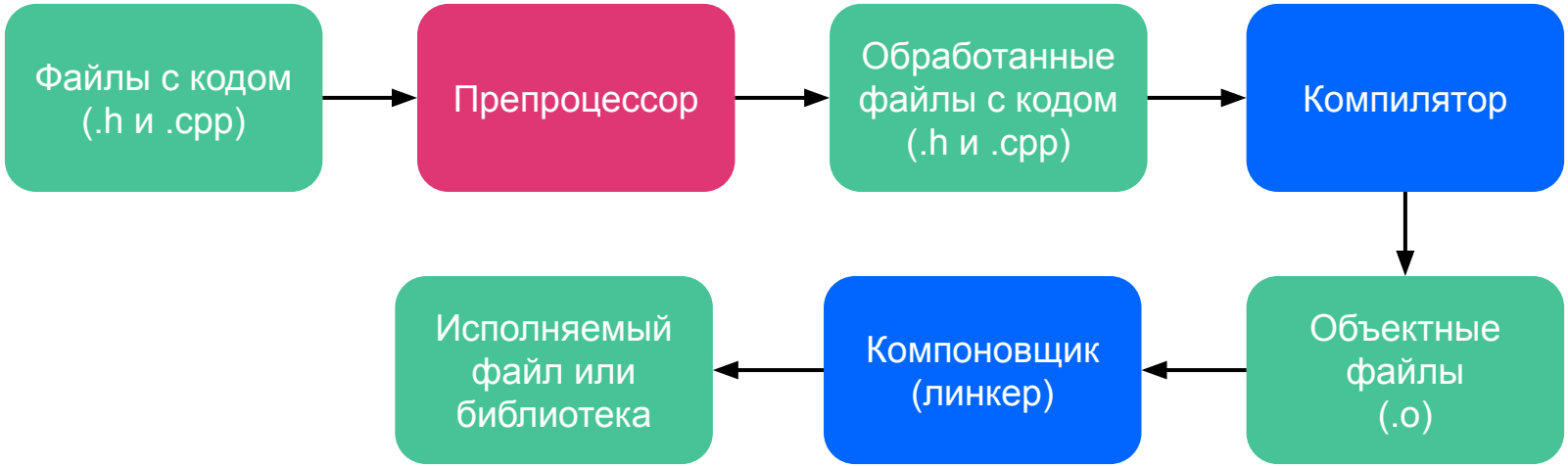


Препроцессор - это инструмент, который обрабатывает файлы с кодом перед компиляцией программы

Препроцессор ищет в файлах с кодом специальные куски кода - **директивы** - и обрабатывает эти куски, после чего обработанные файлы с кодом поступают на вход компилятору, который и превращает их в программу

Процесс сборки программы

Схематично процесс сборки программы выглядит следующим образом:



В рамках своей работы препроцессор работает с **препроцессорными директивами**, которые находятся в файлах с кодом

Как вы думаете

с какой препроцессорной директивой
мы уже знакомы?
Что она делает?

Напишите в чат

Одну директиву мы уже знаем

Это директива **#include**

Препроцессор, встречая эту директиву, выполняет копирование содержимого файла, указанного после директивы #include, на то место в файле, где написана эта директива

Как вы думаете

как препроцессор при сканировании файлов
с кодом отличает препроцессорные директивы
от обычного кода?

Напишите в чат

Препроцессорные директивы

Все препроцессорные директивы начинаются с символа **#** (решётка) - так препроцессор их находит и обрабатывает

Посмотрим, какие директивы существуют, и разберём каждую по отдельности:

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

Директива #define

Директива **#define** создаёт **макрос** - это ассоциация между **идентификатором** или **идентификатором с параметрами** и произвольной строкой (которой может и не быть)

С макросами разберёмся более подробно позже, а сейчас посмотрим на наиболее простой способ использования директивы #define

Синтаксис директивы #define

Директива #define имеет два варианта синтаксиса:

- первый вариант для обычного идентификатора (a.k.a. символьная константа)
- второй вариант для идентификатора с параметрами (a.k.a. макрос):

#define <идентификатор> [<строка>]

#define <идентификатор> ([<идентификатор>], ..., [<идентификатор>]) [<строка>]

Когда препроцессор встречает директиву #define, он запоминает, что **<идентификатор>** определен, и в дальнейшем тексте кода **заменяет** встреченные **<идентификаторы>** **<строкой>**, даже если она пустая - тогда **<идентификатор>** просто исчезнет

Вариант идентификатора с параметрами разберём, когда будем изучать макросы

Как вы думаете

где и зачем мы уже использовали директиву #define?

Напишите в чат

Использование

Мы использовали директиву #define для создания include guard’ов заголовочных файлов

Пустые идентификаторы обычно определяют для того, чтобы далее другими директивами проверить, определены они уже или нет

Также с помощью #define иногда объявляют константы

Идентификаторы, объявляемые с помощью #define, принято именовать заглавными буквами

Пример #define

Посмотрим на простой пример кода с директивой #define

```
#define PI 3.14 // Определили идентификатор PI значением 3.14
int main(int argc, char** argv)
{
    std::cout << "Число пи:" << PI << std::endl;            // Число пи: 3.14
    std::cout << "Число пи + 1:" << PI + 1 << std::endl;    // Число пи + 1: 4.14
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    std::cout << "Число пи:" << 3.14 << std::endl;          // Число пи: 3.14
    std::cout << "Число пи + 1:" << 3.14 + 1 << std::endl;  // Число пи + 1: 4.14
}
```

Ещё пример

Вопрос: как вы думаете, что выведет этот код? Напишите в чат

```
#define NAME John // Определили идентификатор NAME значением John
int main(int argc, char** argv)
{
    std::cout << "Привет, " << NAME << std::endl; // ???
}
```

Ещё пример

Ответ: такой код не скомпилируется, потому что после обработки препроцессором код будет выглядеть вот так:

```
int main(int argc, char** argv)
{
    std::cout << "Привет, " << John << std::endl; // Ошибка
}
```

Важно помнить о том, что вместо идентификатора подставляется **ровно тот текст**, который определён директивой #define

При этом препроцессор **НЕ** заменяет идентификаторы, которые находятся в комментариях и в строках

А как тогда сделать?

Посмотрите на два варианта решения проблемы
Вопрос: как вы думаете, какой из них правильный? Напишите в чат

```
#define NAME John // Определили идентификатор NAME значением John
int main(int argc, char** argv)
{
    std::cout << "Привет, " << "NAME" << std::endl; // ???
}
```

```
#define NAME "John" // Определили идентификатор NAME значением "John"
int main(int argc, char** argv)
{
    std::cout << "Привет, " << NAME << std::endl; // ???
}
```

А как тогда сделать?

Ответ: правильный вариант - второй

```
#define NAME John // Определили идентификатор NAME значением John
int main(int argc, char** argv)
{
    std::cout << "Привет, " << "NAME" << std::endl; // Привет, NAME
}
```

```
#define NAME "John" // Определили идентификатор NAME значением "John"
int main(int argc, char** argv)
{
    std::cout << "Привет, " << NAME << std::endl; // Привет, John
}
```

Директива #undef

Директива **#undef** удаляет идентификатор, созданный ранее с помощью директивы #define

Синтаксис директивы #undef очень прост:

#undef **<идентификатор>**

После обработки этой директивы препроцессор забывает о существовании указанного **<идентификатора>**. Если **<идентификатор>** ранее определён не был, то ошибки не возникает

Пример #undef

Посмотрим на простой пример кода с директивой #undef

```
#define PI 3.14 // Определили идентификатор PI значением 3.14
int main(int argc, char** argv)
{
    std::cout << "Число пи:" << PI << std::endl;
#undef PI // Забыли идентификатор PI
    std::cout << "Число пи:" << PI << std::endl;
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    std::cout << "Число пи:" << 3.14 << std::endl;
    std::cout << "Число пи:" << PI << std::endl; // Ошибка
}
```

Директивы #if и #endif

Директивы **#if** и **#endif** используется для **условной компиляции**

Она проверяет выражение, которое указывается после директивы - если выражение имеет ненулевое значение, оно считается истинным. Если оно равно 0 - считается ложным

В случае, если выражение ложно, то **весь текст**, находящийся **между** этой директивой **#if** и следующей по тексту директивой **#endif**, **исключается** и **не компилируется** - то есть попросту удаляется

Если выражение истинно, то директива #if не оказывает никакого эффекта

Директивы #if и #endif. Синтаксис

Синтаксис использования директив выглядит следующим образом:

#if <константное выражение>
<текст - код, комментарии, директивы>
#endif

<константное выражение> должно удовлетворять следующим условиям:

- имеет целочисленный тип, может включать в себя только целочисленные константы, символьные константы и оператор **defined**
- нельзя использовать оператор sizeof и оператор приведения типа

Директивы #if и #endif. Оператор defined

Препроцессорный оператор **defined** имеет следующий синтаксис:

defined(**<идентификатор>** **)**

defined **<идентификатор>**

Оператор возвращает ненулевой (истинный) результат, если **<идентификатор>** определён (с помощью директивы #define), в противном случае возвращает 0

Результат выполнения оператор defined можно инвертировать - для этого перед оператором defined нужно написать восклицательный знак

Директивы #if и #endif. Пример

Посмотрим на простой пример кода с директивами #if и #endif

```
#define PI 3.14 // Определили идентификатор PI значением 3.14
int main(int argc, char** argv)
{
#if defined PI
    std::cout << "Мы знаем число пи:" << PI << std::endl;
#endif
#if defined E
    std::cout << "Мы знаем число e:" << E << std::endl;
#endif
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    std::cout << "Мы знаем число пи:" << 3.14 << std::endl;
}
```

Директивы #else и #elif

Директивы **#else** и **#elif** являются дополнением для директивы #if.
Они являются аналогами операторов else и else if в C++

Директива **#elif** позволяет задать условие для препроцессора, альтернативное тому, что указано в директиве #if. Количество директив #elif не ограничено

Директива **#else** считается истинной в случае, если ни одно условие из предыдущих директив #if и #elif не выполнилось. Директива #else может быть всего одна

Директивы #elif и #else могут располагаться **только между** директивами #if и #endif

Директивы #else и #elif. Синтаксис

Синтаксис использования директив выглядит следующим образом:

```
#if <константное выражение 1>
<текст - код, комментарии, директивы>
#elif <константное выражение 2>
<текст - код, комментарии, директивы>
...
#elif <константное выражение N>
<текст - код, комментарии, директивы>
#else
<текст - код, комментарии, директивы>
#endif
```

Директивы #else и #elif. Пример

Посмотрим на простой пример кода с директивами #else и #elif

```
#define PI 3 // Определили идентификатор PI значением 3
int main(int argc, char** argv)
{
#if PI == 3
    std::cout << "Сегодня пи равно 3" << std::endl;
#elif PI == 4
    std::cout << "Сегодня пи равно 4" << std::endl;
#else
    std::cout << "Мы не знаем пи" << std::endl;
#endif
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    std::cout << "Сегодня пи равно 3" << std::endl;
}
```

Директивы #ifdef и #ifndef

Директивы **#ifdef** и **#ifndef** - это просто сокращённый вариант директивы #if с операторами defined и !defined соответственно, поэтому после них тоже должна следовать директива #endif

Директивы #ifdef и #ifndef. Синтаксис

Синтаксис:

#ifdef <идентификатор>
<текст - код, комментарии, директивы>
#endif

#ifndef <идентификатор>
<текст - код, комментарии, директивы>
#endif

#ifdef **<идентификатор>** эквивалентно #if defined **<идентификатор>**

#ifndef **<идентификатор>** эквивалентно #if !defined **<идентификатор>**

Директивы #ifdef и #ifndef. Пример

Посмотрим на простой пример кода с директивой #ifndef - include guard

```
#ifndef MY_HEADER_FILE
#define MY_HEADER_FILE
class MyClass
{
public:
    int method();
}
#endif
```

Директива #error

Директива **#error** сигнализирует об ошибке при компиляции. Её назначение - привлечь внимание разработчика к нарушению какого-либо ограничения (установленного другими препроцессорными директивами)

Если препроцессор встречает директиву #error, то он испускает сообщение об ошибке, которое следует за этой директивой, и прекращает компиляцию

Часто её используют для проверки того, что важная символьная константа или макрос были определены

Директива #error. Синтаксис

Синтаксис:

#error **<сообщение об ошибке>**

В **<сообщение об ошибке>** не будут подставляться символьные константы и макросы

Директива #error. Пример

Посмотрим на простой пример кода с директивой #error

Попытка собрать проект не увенчается успехом, будет выведено сообщение об ошибке: “Important macro required”

```
#ifndef MY_HEADER_FILE
#define MY_HEADER_FILE

#ifndef MY_IMPORTANT_MACRO
#error Important macro required
#endif
class MyClass
{
public:
    int method();
}
#endif
```

Директивы #import и #include

Директива **#import** используется в специфических случаях, поэтому подробно рассматривать её не будем. Вкратце - она инкорпорирует информацию из библиотеки типов

С директивой **#include** мы с вами уже знакомы, и синтаксис её изучали, поэтому останавливаться на ней не будем

Директива #line

Директива **#line** устанавливает информацию о номере текущей строки и (опционально) названии файла, в котором находится эта директива

Синтаксис:

#line <номер строки> [“<имя файла>”]

Номер строки и название файла доступны с помощью специальных предопределённых символьных констант **__LINE__** и **__FILE__** (предопределённые константы мы с вами рассмотрим позже)

Директива #line. Пример

```
#include <iostream>
int main(int argc, char** argv)
{
    std::cout << "Это строка номер " << __LINE__ << " в файле " << __FILE__ << std::endl;
#line 20
    std::cout << "Это строка номер " << __LINE__ << " в файле " << __FILE__ << std::endl;
    std::cout << "Это строка номер " << __LINE__ << " в файле " << __FILE__ << std::endl;
#line 30 "hello.cpp"
    std::cout << "Это строка номер " << __LINE__ << " в файле " << __FILE__ << std::endl;
    std::cout << "Это строка номер " << __LINE__ << " в файле " << __FILE__ << std::endl;
}
```

```
Это строка номер 4 в файле C:\MyApp\source.cpp
Это строка номер 20 в файле C:\MyApp\source.cpp
Это строка номер 21 в файле C:\MyApp\source.cpp
Это строка номер 30 в файле C:\MyApp\hello.cpp
Это строка номер 31 в файле C:\MyApp\hello.cpp
```

Директивы #pragma и #using

Директива **#pragma** - это многофункциональная директива, работа которой различается для разных компиляторов. У директивы #pragma есть, в свою очередь, много своих директив, которые определяются строкой, следующей за #pragma

Одну мы с вами знаем - это **#pragma once**

Директива **#using** имеет также специфическое применение - её мы тоже подробно рассматривать не будем, скажем только, что предназначена она для импортирования метаданных из файла

Макросы

2

Макросы

Как мы уже выяснили, макросы определяются с помощью директивы #define и являются параметризованными идентификаторами. Вспомним синтаксис:

#define <идентификатор> ([<идентификатор>], ..., [<идентификатор>]) [<строка>]

Параметры, которые вы объявляете (**<идентификаторы>**), затем используются в **<строке>** и подставляются туда при вызове макроса. Это *похоже* на функцию, но ей не является, так как у параметров здесь нет типов - это всего лишь текстовая подстановка

Макросы. Простой пример

Рассмотрим простой пример макроса MAX

```
#define MAX(a, b) a >= b ? a : b // Определили макрос MAX
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    int max = MAX(num1, num2);
    std::cout << "Максимальное число: " << max << std::endl;
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    int max = num1 >= num2 ? num1 : num2;
    std::cout << "Максимальное число: " << max << std::endl;
}
```

Макросы. Тот же пример

Однако давайте теперь рассмотрим другой пример с макросом MAX

```
#define MAX(a, b) a >= b ? a : b // Определили макрос MAX
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимальное число: " << MAX(num1, num2);
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимальное число: " << num1 >= num2 ? num1 : num2; // Ошибка!
}
```

Как вы думаете

почему возникла ошибка?

Напишите в чат

Причина ошибки

Причина ошибки кроется в синтаксисе - из-за него действия пытаются выполниться в неправильном порядке

Чтобы программа заработала, нужно заключить тернарный оператор в скобки. И это является общей рекомендацией к макросам - всегда заключайте все аргументы макроса и сам макрос при объявлении в скобки, чтобы избежать подобных ошибок

Макросы. Правильный вариант

Вот правильный вариант

```
#define MAX(a, b) ((a) >= (b) ? (a) : (b)) // Определили макрос MAX
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимальное число: " << MAX(num1, num2);
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимум: " << ((num1) >= (num2) ? (num1) : (num2)); // 10
}
```

Макросы. Ещё пример с MAX

Рассмотрим ещё один пример с макросом MAX
Как вы думаете, что выведет этот код?

```
#define MAX(a, b) ((a) >= (b) ? (a) : (b)) // Определили макрос MAX
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимальное число: " << MAX(++num1, ++num2); // ???
}
```

Макросы. Ещё пример с MAX

Внезапно, на консоль будет выведено 12

```
#define MAX(a, b) ((a) >= (b) ? (a) : (b)) // Определили макрос MAX
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимальное число: " << MAX(++num1, ++num2); // 12
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    std::cout << "Максимум: " << ((++num1) >= (++num2) ? (++num1) : (++num2)); // 12
}
```

Ещё одно правило безопасности

Такое поведение возникло из-за того, что макрос раскрыл наш префиксный инкремент в два префиксных инкремента (чего бы не произошло при использовании функции)

Такие ошибки *очень* сложно находить, поэтому при использовании макросов надо **избегать** использования в качестве параметров макроса **выражений и вызовов функций**

Макрос SWAP

Макросы могут состоять из нескольких выражений:

```
#define SWAP(type, a, b) type tmp = (a); (a) = (b); (b) = tmp;
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    SWAP(int, num1, num2)
}
```

Вот как будет выглядеть код после обработки препроцессором:

```
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    int tmp = (num1); (num1) = (num2); (num2) = tmp;
}
```

Макрос SWAP

Макрос можно объявить на нескольких строках, но тогда в конце каждой переносимой строки надо ставить символ “\” (обратный слэш):

```
#define SWAP(type, a, b) \
  type tmp = (a); \
  (a) = (b); \
  (b) = tmp;
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = 10;
    SWAP(int, num1, num2)
}
```

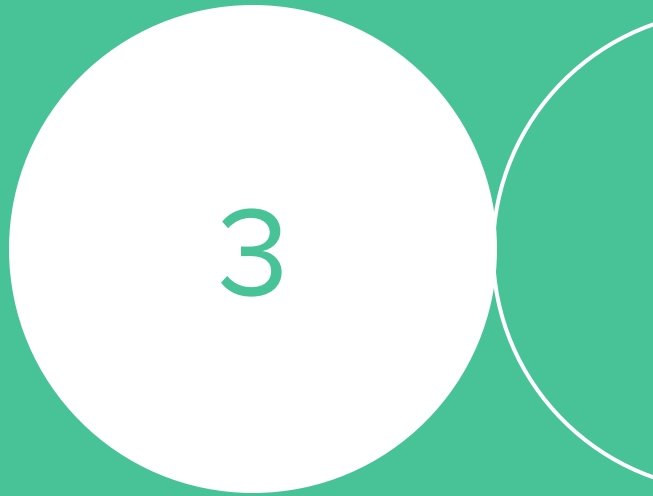
Макросы. Итог

Мы рассмотрели далеко не все возможности применения макросов, но нужно помнить одно - в C++ **не рекомендуется** использование макросов. В языке C альтернатив макросам не было, но в C++ для этих задач есть более безопасные инструменты

Если вы всё-таки используете макросы, соблюдайте следующие правила безопасности:

- Не используйте в качестве параметров выражения и функции
- Все аргументы макроса и сам макрос по возможности должны быть заключены в скобки
- Многострочные макросы должны иметь свою область видимости

Предопределённые символьные константы



Предопределённые символьные константы

Компиляторы языка C++ автоматически определяют некоторые символьные константы, которыми можно пользоваться в своём коде

Почти все предопределённые константы начинаются и заканчиваются символами ___ (два нижних подчёркивания). Поэтому при определении своих констант **не** используйте такой стиль наименования, чтобы не спутать свои константы с предопределёнными

Некоторые константы определяются только некоторыми компиляторами, другие же являются стандартными и доступны для использования во всех компиляторах

Далее мы рассмотрим стандартные константы

Стандартные символьные константы

- **__func__** - название функции, **внутри** которой расположен идентификатор
- **__cplusplus** - определён, если файл компилируется компилятором C++. Иначе не определён
- **__DATE__** - дата компиляции текущего исходного файла. Строка в формате “Mm dd yyyy”
- **__FILE__** - имя текущего файла
- **__LINE__** - номер строки текущего файла
- **__STDC__** - определён как 1, если файл компилируется компилятором C. Иначе не определён. У этой символьной константы есть несколько “сестёр”, которые также начинаются с __STDC и принимают разные значения в зависимости от компилятора и способов компиляции
- **__TIME__** - время компиляции текущего файла. Строка в формате “hh:mm:ss”

Итоги

Итоги занятия

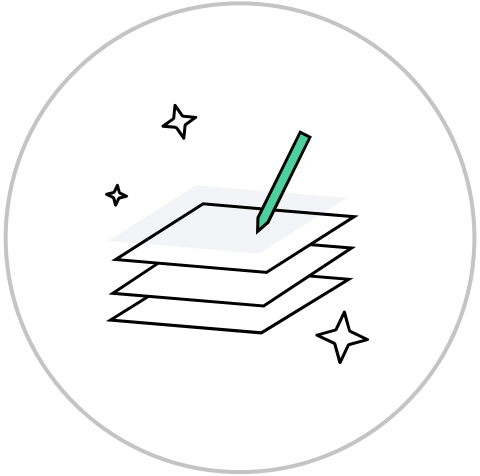
Сегодня мы

1. Разобрались, как собираются программы
2. Познакомились с препроцессорными директивами
3. Узнали, что такое макросы
4. Выяснили, какие бывают символьные константы

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Препроцессор и макросы
- Документация Microsoft

Задавайте вопросы и пишите отзыв о лекции

Максим Бакиров
C++ - разработчик в Яндекс