Многофайловые проекты и библиотеки

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: что такое наследование?

Вспоминаем прошлое занятие

Вопрос: что такое наследование?

Ответ: наследование - механизм ООП, позволяющий одним классам наследовать другие, перенимая их члены себе

Вспоминаем прошлое занятие

Вопрос: какой модификатор доступа даёт доступ к члену класса наследникам, но не внешнему коду?

Вспоминаем прошлое занятие

Вопрос: какой модификатор доступа даёт доступ к члену класса наследникам, но не внешнему коду?

Ответ: модификатор доступа protected

Вспоминаем прошлое занятие

Вопрос: что происходит с конструкторами при наследовании?

Вспоминаем прошлое занятие

Вопрос: что происходит с конструкторами при наследовании?

Ответ: конструкторы не наследуются, но вызываются при создании объекта - сначала родительские, потом дочерние

Вспоминаем прошлое занятие

Вопрос: что такое множественное наследование?

Вспоминаем прошлое занятие

Вопрос: что такое множественное наследование?

Ответ: это когда один класс наследуется сразу от нескольких. Использовать этот механизм следует с особой осторожностью и полным пониманием происходящего

Вспоминаем прошлое занятие

Вопрос: что такое полиморфизм и виртуальные методы?

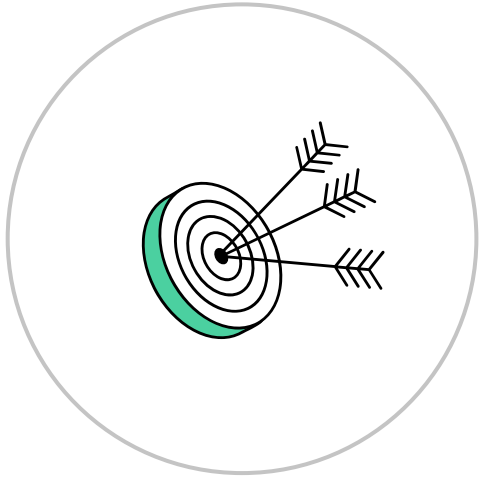
Вспоминаем прошлое занятие

Вопрос: что такое полиморфизм и виртуальные методы?

Ответ: полиморфизм - способность использовать класс-наследник вместо класса-родителя с поведением класса-наследника. Виртуальные методы могут быть переопределены в классе-потомке

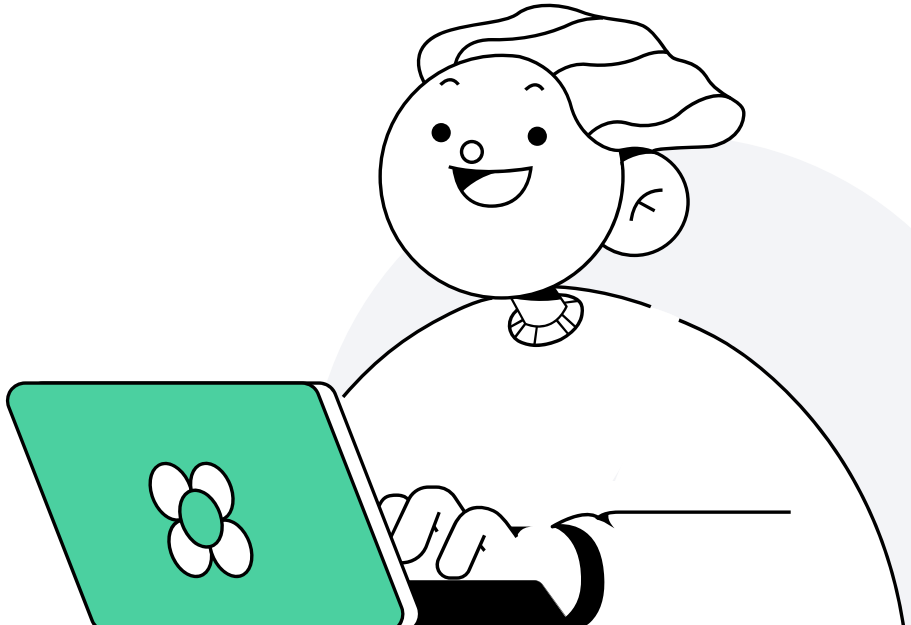
Цели занятия

- Разберёмся, что такое многофайловые проекты
- Познакомимся с директивой #include
- Узнаем, что такое заголовочный файл и файл исходного кода
- Выясним, как выносить класс в отдельный файл



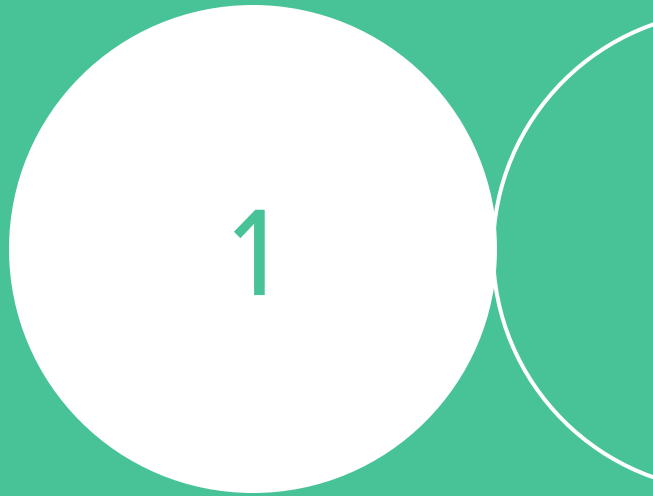
План занятия

1. Многофайловые проекты
2. Заголовочные файлы
3. Вынесение класса в отдельный файл
4. Итоги
5. Домашнее задание



*Нажми на нужный раздел для перехода

Многофайловые проекты

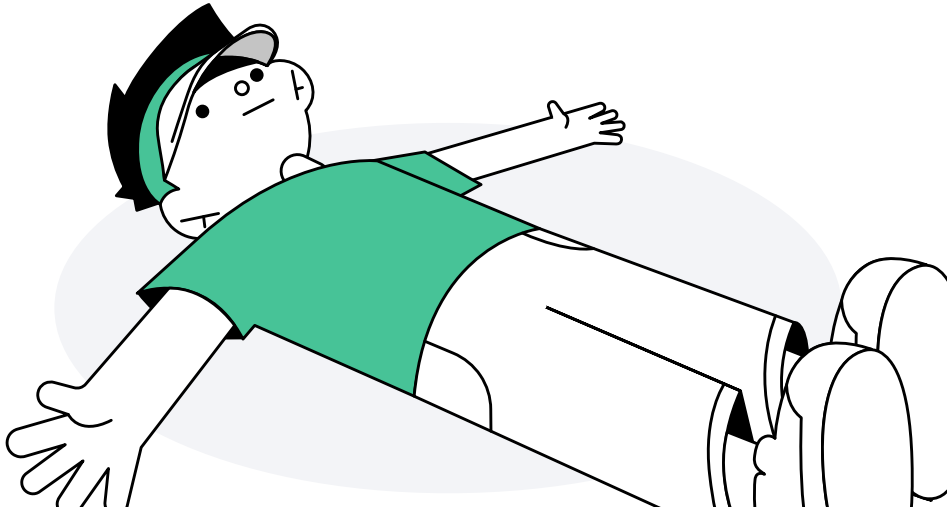


В чём проблема?

Вы уже увидели, что хоть сколько-нибудь сложные программы занимают много места - и когда счёт строк кода идёт на сотни, с программой уже становится сложно работать

Ведь нужно помнить, какие классы и функции вы создали, при необходимости изменения нужно пользоваться поиском или просматривать файл, чтобы найти нужные участки кода

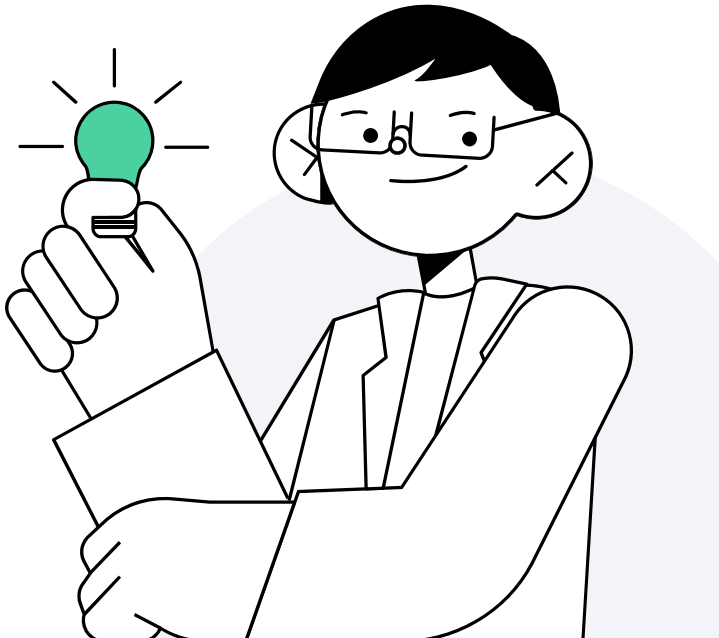
Чем больше кода, тем сложнее это делать



Как с этим справляться?

Когда программа становится сложнее, в ней можно увидеть разные части, которые выполняют **разные** задачи

Для упрощения разработки кода и его поддержки люди разносят части программы, выполняющие разные задачи, по разным **файлам**



Какие бывают файлы для кода?

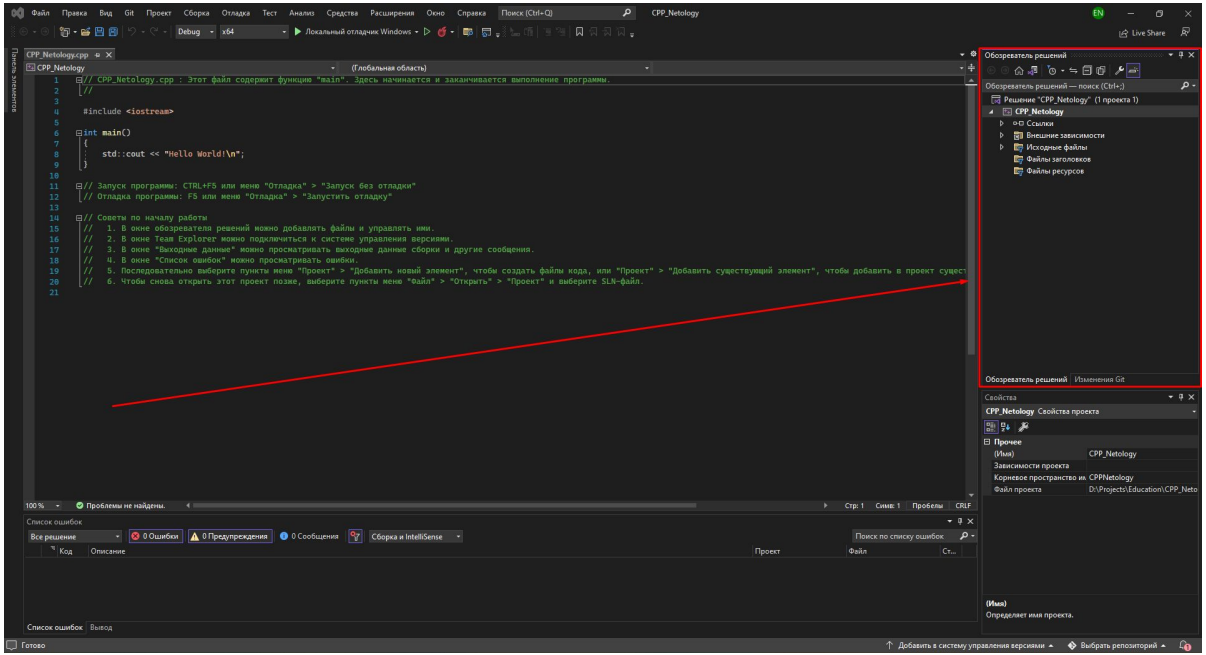
В C++ в основном используется два вида файлов для кода:

- **Исходные файлы** (**Source files**)
  это файлы с расширением **.cpp**. В таких файлах размещается основная часть кода. Кстати, при создании нового консольного проекта точка входа в программу (функция main) размещается в автоматически созданном файле .cpp

- **Файлы заголовков** (**Header files**, ещё известны как **заголовочные файлы**)
  это файлы с расширением **.h**. С ними подробно познакомимся позднее

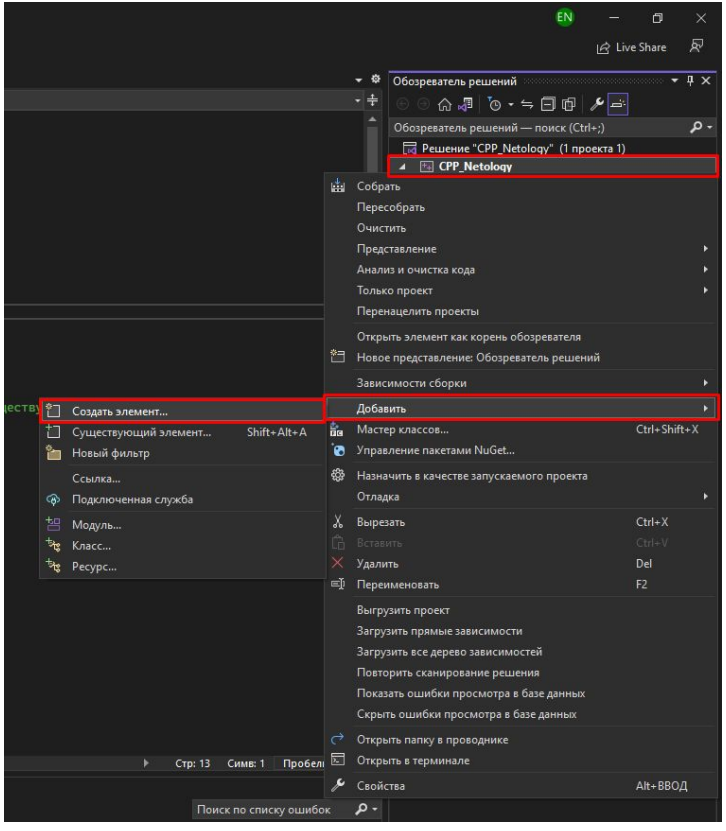
Как создать новый файл кода?

Microsoft Visual Studio предоставляет удобный интерфейс для создания новых файлов. В секции **“Обозреватель решений”** можно увидеть несколько “папок” - это не папки в смысле файловой системы, а фильтры для разделения файлов кода



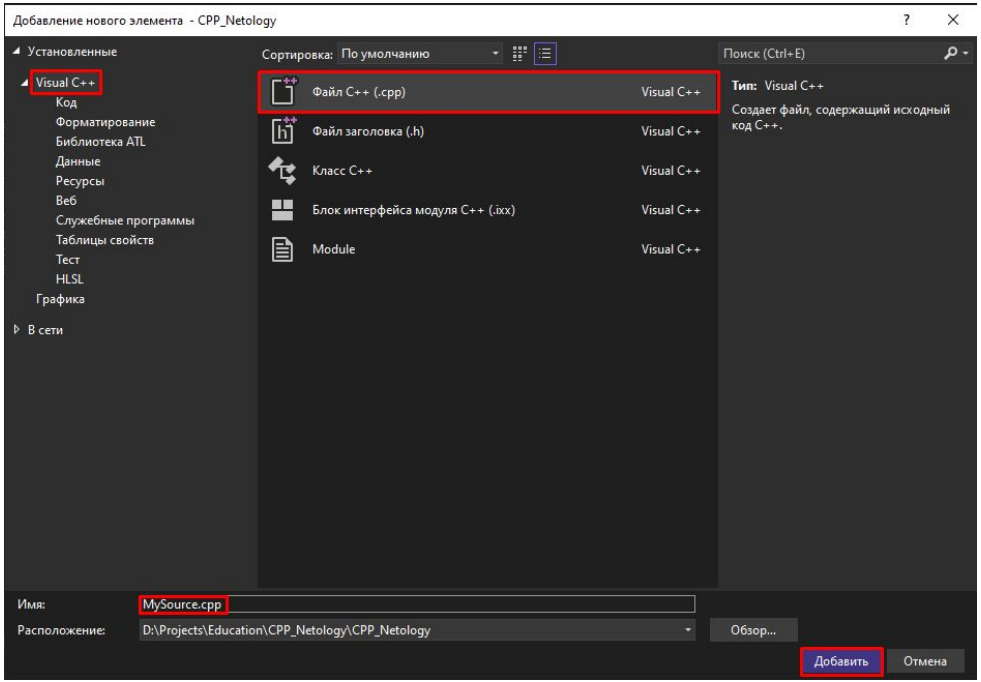
Как создать новый файл кода?

Для создания нового файла щёлкните правой кнопкой мыши по **названию проекта**, в появившемся контекстном меню выберите пункт **“Добавить”** => **“Создать элемент”**



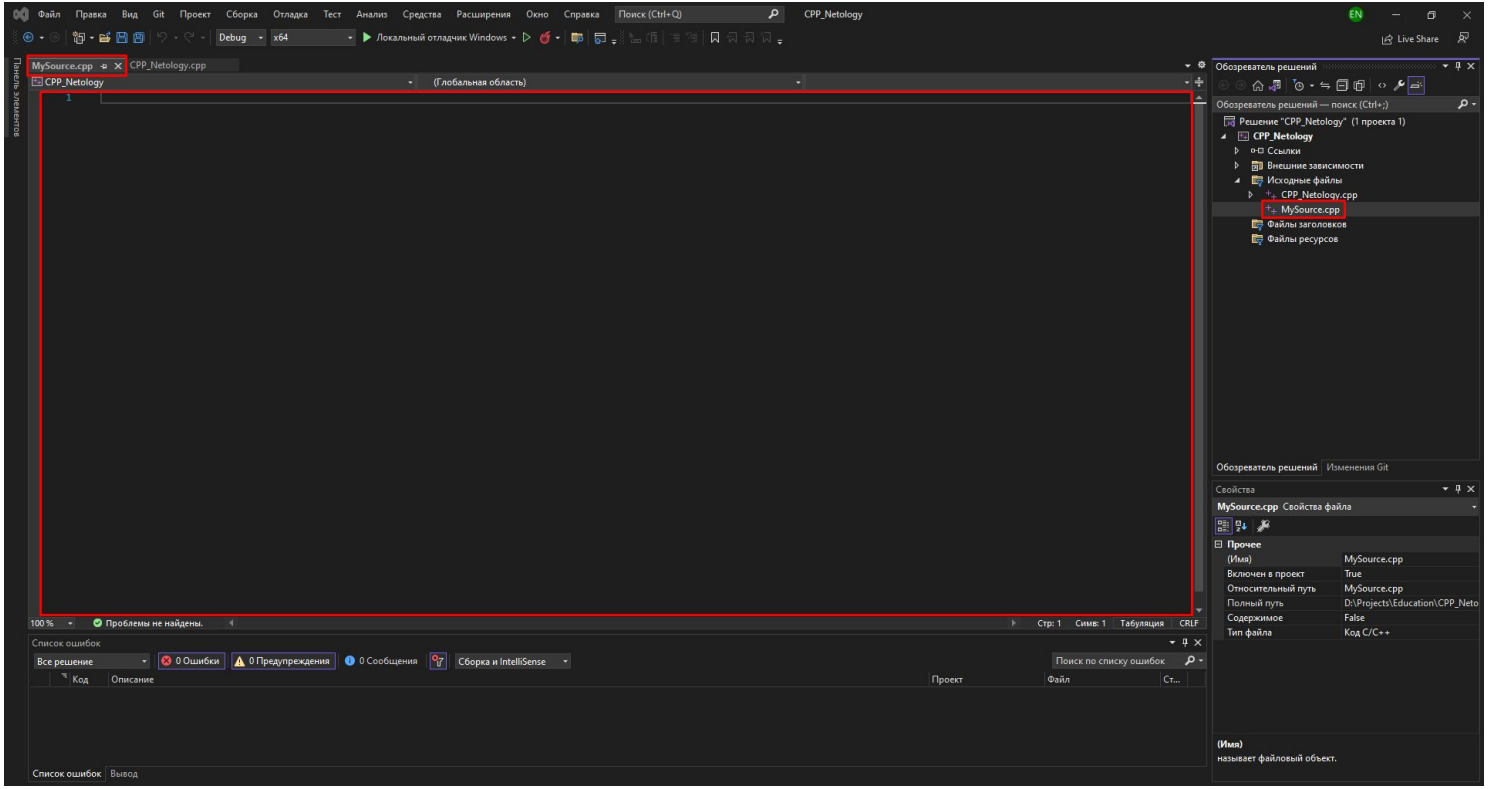
Как создать новый файл кода?

Перед вами появится окно для добавления нового элемента. В меню, расположенном слева, выберите **“Visual C++”** (должен быть уже выбран), в списке по центру выберите **“Файл C++ (.cpp)”**, внизу можете ввести имя для вашего нового файла (по умолчанию Source.cpp). Назовём наш файл **MySource.cpp**
В конце нужно щёлкнуть **“Добавить”** или нажать Enter



Как создать новый файл кода?

После этого в центральной области для написания кода у вас должна открыться новая пустая область, появиться вкладка “MySource.cpp”. Также в “Обозревателе решений” в фильтре “Исходные файлы” появится новый файл “MySource.cpp”



Попробуем перенести код

Представим, что есть функция, которую мы хотим вынести в новый файл. Например, функция power_2, которая вычисляет квадрат числа

```
int power_2(int x) // вот эту функцию мы хотим вынести в новый файл
{
    return x * x;
}
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // 25
}
```

Попробуем перенести код

Создадим файл “math_sqrt.cpp”, перенесём функцию туда и попробуем собрать

```
// файл math_power.cpp
int power_2(int x)
{
    return x * x;
}
```

```
// основной файл (например, source.cpp - далее это будет основной файл)
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // ???
}
```

Как вы думаете

что будет выведено при вызове функции power_2?

Напишите в чат

Попробуем перенести код

Программа не соберётся
Будет выведена ошибка “идентификатор power_2 не определён”

```
// файл math_power.cpp
int power_2(int x)
{
    return x * x;
}
```

```
// основной файл (например, source.cpp - далее это будет основной файл)
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // ???
}
```

Почему так?

Так происходит потому, что в файле source.cpp нет объявления функции power_2. Объявим эту функцию (напишем её прототип):

```
// файл math_power.cpp
int power_2(int x)
{
    return x * x;
}
```

```
// файл source.cpp
int power_2(int x); // объявляем функцию
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // ???
}
```

Не лучший вариант

Код соберётся и отработает правильно. Однако, так никто не делает

```
// файл math_power.cpp
int power_2(int x)
{
    return x * x;
}
```

```
// файл source.cpp
int power_2(int x); // объявляем функцию
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // 25
}
```

Как вы думаете

почему программисты не пользуются таким подходом?

Напишите в чат

Причины

Программисты не используют такой подход по следующим причинам:

- Высока вероятность ошибки
  Представьте, что вы хотите воспользоваться написанными кем-то функциями и классами - вам придётся вручную писать определения для всех функций, которые вам нужны, со всеми типами аргументов, перегрузками и пр.

- Очень плохо масштабируется
  Часто код пишется в отдельных файлах для того, чтобы можно было его потом легко переиспользовать

  Высокоуровневое программирование построено на грамотном переиспользовании, а при таком подходе оно практически невозможно - ведь почти весь ваш код будет состоять из прототипов. Непонятно, где определены ваши прототипы. И есть потенциальная проблема с названиями - они должны быть уникальны

А как правильно?

Общепринятым подходом для разнесения кода по разным файлам является использование **заголовочных файлов** (**header files**)

Заголовочные файлы

2

Что такое заголовочные файлы?

Заголовочные файлы - это файлы с расширением **.h** (иногда можно встретить расширение **.hpp**) - сокращение от слова header

Заголовочный файл практически всегда идёт в паре с файлом исходного кода с таким же названием. При этом в заголовочном файле размещаются **объявления** функций и переменных, а также **определения** классов **без реализаций** их методов

Заголовочные файлы **подключаются** с помощью директивы **#include** (мы с ней уже немного знакомы)

Директива #include

Директива **#include** (англ. include - включать в себя) позволяет после себя указать название файла и приводит к **включению** содержимого этого файла в то место, где указана директива #include. Попросту говоря, на место директивы копируется указанный файл как он есть

Этот механизм используется в сочетании с заголовочными файлами, в которых программисты пишут **объявления** и **определения** функций, переменных и классов. При этом заголовочный файл должен содержать только ту информация, которая необходима для **использования** функций, классов и переменных, а **реализация** должна быть вынесена в соответствующий .cpp файл

Директива #include

Существует два вида синтаксиса для директивы #include:

1. **#include "путь до файла"**
2. **#include <путь до файла>**

Два вида синтаксиса различаются тем, **где** компилятор ищет требуемый файл. При использовании угловых скобок (< >) компилятор ищет файлы в специальных местах, а при использовании кавычек он также ищет файлы в директориях проекта

В общем случае при использовании стандартных библиотек (таких как iostream) используется синтаксис с угловыми скобками, а при использовании своих локальных заголовочных файлов используется синтаксис с кавычками

Почему так сложно?

Если вы знакомы с каким-либо современным высокоуровневым языком программирования, вы зададите справедливый вопрос - зачем всё это? Можно ведь проще

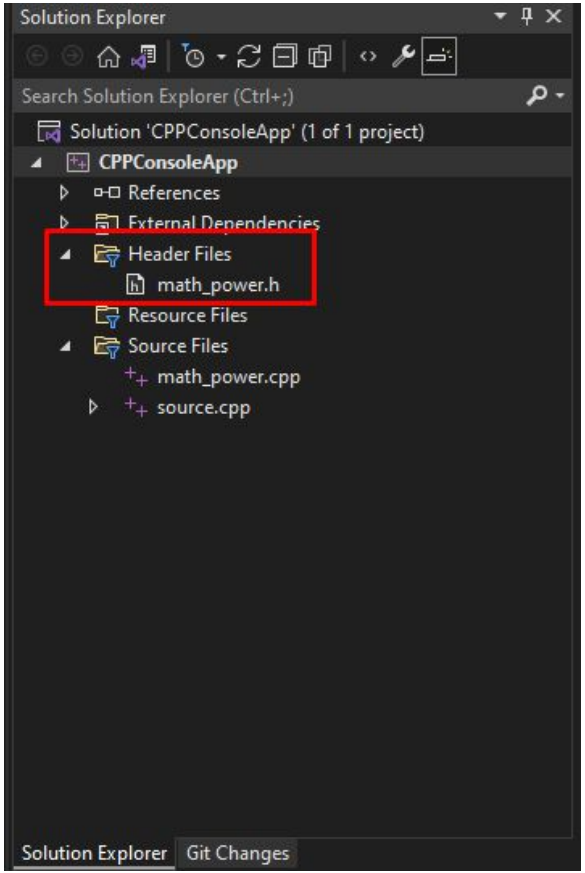
Проще действительно можно, но такая система существует по историческим причинам. Во время создания C++ возможности компьютеров были много меньше, чем сейчас, поэтому часть работы по связыванию разнесённых по разным файлам кусков кода (которую современные языки выполняют сами) была возложена на плечи программистов

Как создать заголовочный файл?

Заголовочный файл создаётся аналогично файлу с кодом - но на этапе выбора нового элемента вы выбираете не “Файл C++ (.cpp)”, а **“Файл заголовка (.h)”**

Давайте в пару к нашему файлу “math_power.cpp” создадим заголовочный файл “math_power.h”. Он должен появиться в “Обозревателе решений”

Заметьте, что во вновь созданном заголовочном файле уже написана строчка - #pragma once. Чуть позже мы разберёмся, что это, пока оставьте её



Пример

Рассмотрим простейший пример вынесения нашей функции

```
// файл math_power.h
int power_2(int x); // объявление функции power_2
```

```
// файл math_power.cpp
#include "math_power.h" // в cpp файлах обычно включают свой заголовочный файл
int power_2(int x)
{
    return x * x;
}
```

```
// файл source.cpp
#include "math_power.h"
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // 25
}
```

Преимущества заголовочных файлов

Заголовочные файлы решают проблему потенциальных ошибок и масштабирования - заголовочный файл со всеми объявлениями пишется один раз, далее нужно лишь его подключать

Заголовочный файл также выступает в роли некого интерфейса - он сообщает, какие функции, переменные и классы доступны для внешнего использования

Как вы думаете

в чём заключается
потенциальная проблема использования
заголовочных файлов с директивой #include?

Напишите в чат

Двойное подключение

Существует проблема двойного подключения

Если файл “a.h” содержит в себе определение структуры “foo” с одним полем, файл “b.h” включает в себя файл “a.h”, а файл “source.cpp” включает в себя файлы “a.h” и “b.h” - в файле “source.cpp” в итоге окажется два определения структуры “foo”, и программа не соберётся

Пример двойного подключения

При сборке возникнет ошибка: “Переопределение структуры foo”

```
// файл a.h
struct foo
{
    int field;
};
```

```
// файл b.h
#include "a.h"
```

```
// файл source.cpp
#include "a.h"
#include "b.h"
int main(int argc, char** argv)
{ }
```

Как вы думаете

как решить проблему двойного подключения?

Напишите в чат

Решение проблемы

Решение проблемы двойного подключения заключается в том, чтобы как-то определить, подключался ли заголовочный файл уже в других наших подключениях или ещё нет

Для этого существует два инструмента:

- Include guards
- Директива #pragma once

Include guards

Include guards (можно примерно перевести как “стражи подключения”) - это языковая конструкция, использующая в себе директивы #define, #ifndef и #endif (позже мы узнаем, что они значат)

Заголовочный файл с include guard выглядит так:

```
#ifndef УНИКАЛЬНАЯ_СТРОКА_ВАШЕГО_Н_ФАЙЛА
#define УНИКАЛЬНАЯ_СТРОКА_ВАШЕГО_Н_ФАЙЛА
<содержимое заголовочного файла>
#endif
```

Уникальная строка для вашего файла не должна содержать пробелы и русские буквы

Include guards - это более старый механизм, при его использовании нужно соблюдать уникальность используемой строки

Директива #pragma once

Директива **#pragma once** - более простой инструмент, его минус заключается в том, что не все старые компиляторы могут его поддерживать

Заголовочный файл с директивой выглядит так:

#pragma once
<содержимое заголовочного файла>

Использование директивы является более предпочтительным, чем include guards

Пример

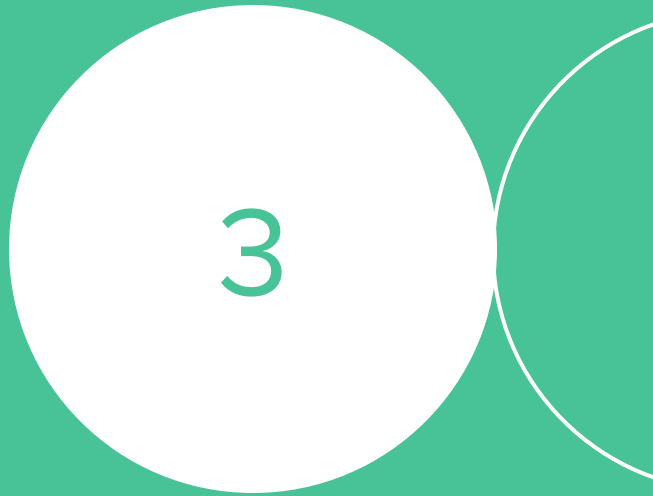
Рассмотрим полностью правильный пример вынесения нашей функции

```
// файл math_power.h
#pragma once
int power_2(int x); // объявление функции power_2
```

```
// файл math_power.cpp
#include "math_power.h" // в cpp файлах обычно включают свой заголовочный файл
int power_2(int x)
{
    return x * x;
}
```

```
// файл source.cpp
#include "math_power.h"
int main(int argc, char** argv)
{
    std::cout << power_2(5) << std::endl; // 25
}
```

Вынесение класса в отдельный файл



Вынесение класса

Классы логично выносить в отдельные файлы, потому что они должны являться самостоятельными единицами

Представим, что у нас есть класс “Calculator” с методом int power_2(int x), который возводит аргумент в квадрат. Создаём для этого класса два файла - “calculator.h” и “calculator.cpp”

Вопрос: Как вы думаете, что мы должны написать в заголовочный файл и что мы должны написать в файл .cpp?

Напишите в чат

Попробуем вынести

Можно предположить что-то такое:

```
// файл calculator.h
#pragma once
class Calculator; // объявление класса Calculator
```

```
// файл calculator.cpp
#include "calculator.h"
class Calculator // определение класса Calculator
{
    int power_2(int x) { return x * x; }
};
```

```
// файл source.cpp
#include "calculator.h"
int main(int argc, char** argv)
{
    Calculator c; // Ошибка: тип Calculator неполный
    std::cout << c.power_2(5) << std::endl;
}
```

Как вы думаете

почему возникла ошибка?

Напишите в чат

Причина ошибки

Ошибка возникла потому, что в заголовочном файле о классе Calculator ничего не известно - просто известно, что он существует. Соответственно, компилятор не может определить, как его правильно конструировать, какие методы и поля у него есть

Поэтому при вынесении класса в заголовочный файл нужно указывать, какие члены у него есть. При этом **определение** этих членов может находиться в .cpp файле

Определение в .cpp файле

Но тогда как правильно написать определение отдельных членов класса в .cpp файле, если класс уже объявлен?

Для этого используется синтаксис **внешнего определения членов класса** - снаружи класса перед названием члена указывается название класса, за которым следуют два двоеточия:

```
class MyClass
{
public:
    int field;
    MyClass(); // объявление конструктора без параметров
    void method(); // объявление метода
};

MyClass::MyClass() { ... } // определение конструктора без параметров
void MyClass::method() { ... } // определение метода
```

Выносим класс

В итоге класс в отдельном файле будет выглядеть вот так:

```
// файл calculator.h
#pragma once
class Calculator
{ public: int power_2(int x); };
```

```
// файл calculator.cpp
#include "calculator.h"
int Calculator::power_2(int x) { return x * x; }
```

```
// файл source.cpp
#include "calculator.h"
int main(int argc, char** argv)
{
    Calculator c;
    std::cout << c.power_2(5) << std::endl; // 25
}
```

Итоги

Итоги занятия

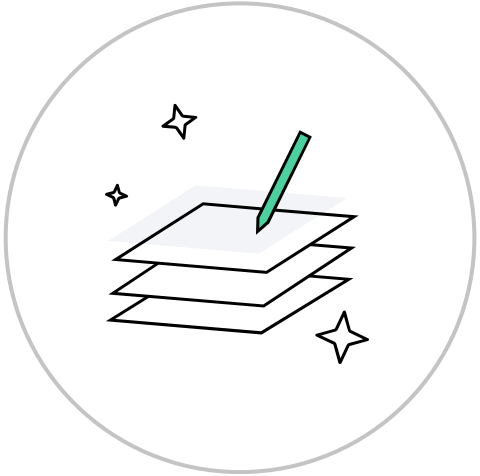
Сегодня мы

1. Разобрались, что такое многофайловые проекты
2. Познакомились с директивой #include
3. Узнали, что такое заголовочный файл и файл исходного кода
4. Выяснили, как вынести класс в отдельный файл

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

- Многофайловые проекты

Задавайте вопросы и пишите отзыв о лекции

Максим Бакиров
C++ - разработчик в Яндекс