Лекция 1. Как начать программировать на C++ и не утонуть в нем

CS Club Novosibirsk, 2019

Знакомство

- Валерий Лесин
  - Технический директор SimLabs (sim-labs.com)
  - Преподаватель C++ в СПбАУ, CS Center, ИТМО КТ
  - valery.lesin@sim-labs.com

- Для кого эта лекция:
  - Для того, кто изучал C++, но не стал им пользоваться из-за сложности
  - Для того, кто изучает сейчас, и пока много непонятного – вероятно, лекция вопросов прибавит. Но надеюсь, и ответов.
  - Для тех, кто используется C++ сейчас и хочет улучшить навыки.
  - Есть ли новички? А профессионалы?

- По ходу лекции:
  - Задавайте вопросы (представляйтесь, пожалуйста)
  - Более продвинутые слайды помечены (*)
  - Иногда встречаются ветвления от основных тем

План лекций

- План лекций на 14-16 ноября:
  - Как начать программировать на C++ и не утонуть в этом. Библиотеки, утилиты, средства разработки, сборка
  - Работа с памятью. Утечки ресурсов и как их избежать. RAII, умные указатели
  - Move semantics, rvalue reference, perfect forwarding
  - Callbacks: lambda, bind & function
  - Multithreading in C++ (потоки, блокировки, задачи, …)
  - Обзор возможностей современных стандартов (auto/decltype, generics, concepts, ranges, modules, constexpr)
- На закуску: две домашние задачи с их разбором.
- Примечание: в примерах используются типы из std, иногда из boost – отдельно оговаривается.

Часть 1. Введение

Что почитать?

- Книги
  - Серия C++ In-Depth
    - Б. Му, Э. Кениг. Эффективное программирование на C++
    - Стандарты программирования на C++
    - Решение сложных/новые сложные задачи на C++
    - …
  - Скотт Мейерс. Эффективный и современный C++

- Блоги, сайты:
  - cppreference.com – справочник
  - C++ Core Guidelines
  - moderncpp by Rainer Grimm
  - simplify C++ (arne-mertz.de)
  - лекции и доклады конференций CppRussia, CppCon

Суперкраткая история языка

- Появление
  - Bjarne Stroustrup (Bell Labs), начало 80-х
  - транслятор программ в язык C

- Стандарты
  - базовые элементы для реализации компиляторами
  - 1998 / 2003 / 2005 Technical Report 1
  - Период затишья
  - 2011 / 2014 / 2017, C++11/14/17
  - 2020 – много фундаментально новых возможностей

Философия языка

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off"

Bjarne Stroustrup

Философия языка

- Большая свобода выбора, даже если выбор неверен – «язык ответственных программистов»
- Универсальный язык со статической типизацией
- Максимальная обратная совместимость с C (на уровне компиляции и линковщика), но не полная
- Множество стилей
  - процедурное программирование
  - ООП
  - обобщенное программирование (STL алгоритмы)
  - метапрограммирование (e.g. boost.spirit)
- Не платить за то, что не используешь*
- Переносимый, избегать платформозависимых особенностей

Применимость. За!

- C++ - всего лишь инструмент, весьма универсальный, поэтому не всегда удобный

- Программы с высокими требованиями к ресурсам компьютера: процессору и памяти, с жестким контролем памяти
  - OS и сопутствующие программы (Windows, KDE)
  - драйверы/встроенные системы, IoT
  - научные программы (иногда в связке с Python, ...)
  - игры/симуляторы
  - сервера с высокой нагрузкой (google search, maps)
  - многие компиляторы
  - нагруженный финтех и т.д.

Применимость. Против!

- Возможно, не лучший выбор для:
  - клиентских частей web-приложений (для любителей есть wasm)
  - небольших легкопереносимых программ-сценариев (лучше скрипты)
  - некоторых приложений, не очень требовательных к ресурсам
  - областей, где существенно богаче окружения других языков

- Универсальность порождает сложность
  - более высокий порог вхождения чем Java, C# и тем более Python – для первого языка программирования непросто

- Выход: хладнокровно выбирать язык; иногда комбинировать с другими языками программирования (например, с Python). Но лучше не в одном процессе (обмен, например, через Protobuf)

Библиотеки

- Стандартные: CRT, STL
  - наиболее используемые структуры данных и алгоритмы
  - работа с примитивами OS (файлы, многопоточность, время, …)
  - качественная реализация и оптимизация компилятором

- Работа с OS (WinAPI, POSIX, …)

- Общего назначения (Boost, Qt, Poco, …)

- Boost. Программировать на C++ без boost – деньги на ветер!
  - Smart pointers, optional, any,
  - Bind, function, signals2,
  - Format, string algorithms, lexical cast,
  - Test,
  - Thread, interprocess, asio, filesystem,
  - Boost python,
  - Program options, preprocessor,
  - И многие-многие другие…

Средства разработки

- Компиляторы: **g++**, **MSVC**, **Clang** (LLVM), icc (intel)
- Автоматизация сборки: **cmake,** make (Makefile), autotools , SCons, qmake, qbs
- Package managers: conan, vcpkg
- Отладчик : gdb, MSVS, LLDB
- Тестирование: boost.test , **google.test**, CUTE
- Системы контроля версий: CVS, SVN, git , hg, …
- Integrated development environment (IDE): **Clion, MSVS, QtCreator,** VS Code, Eclipse CDT, NetBeans
- Доп. рефакторинг для MSVC: Visual Assist, ReSharper
- Online compilers: ideone, wandbox
- Статический анализ кода: PVS Studio, cppcheck, Clang
- Профайлеры: AQTime, VTune, MSVC, Valgrind
- Инструментальная отладка: Address/Thread/Memory/UB Sanitizers, Valgrind

Часть 2. Вспомним базовые возможности

Та самая программа

```
1.  // main.cpp
2.  #include <iostream>
3.
4.  int main(int argc, char* argv[])
5.  {
6.      using namespace std;
7.      cout << "Hello, " << argv[1] << endl;
8.
9.      return 0;
10. }
```

```
1.  $ g++ main.cpp -o hw_app
2.  $ ./hw_app World!
3.  Hello, World!
```

Объявление переменных

```
1.  #include <string>
2.
3.  int main()
4.  {
5.      using namespace std;
6.
7.      int answer = 42;
8.      const double pi(3.14);
9.
10.     string language{"C++"};
11.
12.     string foo;
13.     foo = "bar";
14.
15.     return 0;
16. }
```

Функции, условия, циклы

```
1.  string to_string(int value) {
2.      if (value == 0) return "0";
3.
4.      auto sign = value < 0;
5.      value = abs(value);
6.
7.      string res;
8.      while (value > 0) {
9.          res.push_back('0' + value % 10);
10.         value /= 10;
11.     }
12.
13.     if (sign) res.push_back('-');
14.
15.     reverse(begin(res), end(res));
16.     return res;
17. }
18. //...
19. to_string(-15);
20.
```

Классы значений

```
1.  struct dyn_array
2.  {
3.      dyn_array() noexcept;
4.      explicit dyn_array(size_t num, double def = 0);
5.
6.      dyn_array(dyn_array const& other);
7.      dyn_array& operator=(dyn_array const& other);
8.
9.      double& operator[](size_t i);
10.     double  operator[](size_t i) const;
11.
12.     size_t size() const;
13.     bool   empty() const;
14.     /* more code */
15. private:
16.     void fill_with(double value);
17.
18. private:
19.     double* data_;
20.     size_t  size_;
21. };
```

Динамический полиморфизм

```
1.  struct game_object
2.  {
3.      virtual ~game_object() {}
4.      virtual void render(engine&, context const&) = 0;
5.  };
6.
7.  struct car: game_object {
8.      /*...*/
9.      void render(engine&, context const&) override {/*...*/ }
10. };
11.
12. struct prize: game_object {
13.     /*...*/
14.     void render(engine&, context const&) override {/*...*/ }
15. };
16.
17. void render_objects(vector<game_object*> const& objs/*, ...*/)
18. {
19.     for (auto ptr : objs)
20.         ptr->render(/*...*/); // real object’s render
21. }
```

Статический полиморфизм

```
1.  template<class FwdIt, class Comp = less<>>
2.  constexpr void sort(FwdIt begin, FwdIt end, Comp comp = Comp())
3.  {
4.      /*...*/
5.  }
6.  //...
7.  int main() {
8.      vector items = { 3, 14, 15, 92, 6, 53, 5, 89 }; // error?
9.      sort(begin(items), end(items));
10.
11.     string arr[] = { "one", "two", "three" };
12.     sort(arr, arr + 5);
13. }
```

Lambda functions

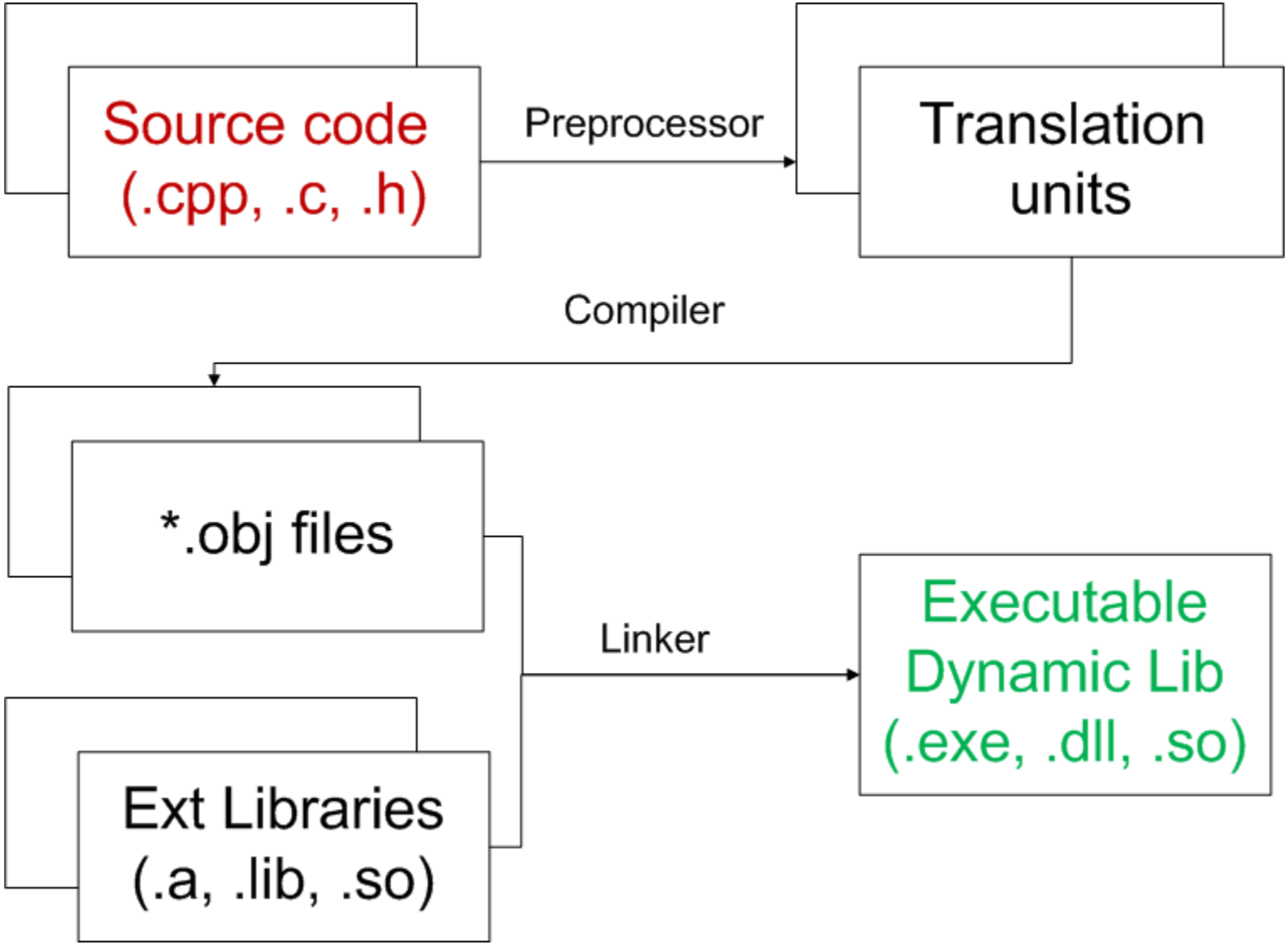
```
1.  template<class FwdIt, class Comp = less<>>
2.  constexpr void sort(FwdIt begin, FwdIt end, Comp comp = Comp())
3.  {
4.      /*...*/
5.  }
6.  //...
7.  int main() {
8.      vector<string> strings = { "one", "two", "three" };
9.      auto case_insensitive_comp =
10.         [](auto const& lhs, auto const& rhs)
11.     {
12.         /* case insensitive string comparison */
13.     };
14.
15.     sort(begin(strings), end(strings), case_insensitive_comp);
16. }
```

Concepts, constrained placeholders*

```
1.  constexpr void mysort(
2.      bidirectional_iterator begin,
3.      bidirectional_iterator end)
4.  {/*sort code*/}
5.
6.  constexpr void mysort(
7.      random_access_iterator begin,
8.      random_access_iterator end)
9.  {/*sort code*/}
10.
11.
12. //...
13. int main() {
14.     std::vector<string> vs = { "one", "two", "three" };
15.     mysort(begin(vs), end(vs));
16.
17.     std::list<string> ls = { "one", "two", "three" };
18.     mysort(begin(ls), end(ls));
19.     return 0;
20. }
```

Часть 3. Компоновка (пока без модулей)

Компиляция


Source code
(.cpp, .c, .h)
Preprocessor
Translation units
Compiler
*.obj files
Linker
Executable
Dynamic Lib
(.exe, .dll, .so)
Ext Libraries
(.a, .lib, .so)

Объявления и определения

```
1.  // declarations
2.  void foo(int bar);
3.  extern int value;
4.
5.
6.  // definitions
7.  void foo(int bar)
8.  {
9.      return bar + 5;
10. }
11.
12. int value;
13. double e = 2.71;
```

- Объявлений в программе может быть сколько угодно. Но одно на translation unit (TU)
- Определение должно быть ровно одно на программу (или пользуемся ODR)

Объявление функции

```
1.  //-- file1.cpp
2.  #include <iostream>
3.
4.  void greet()
5.  {
6.      std::cout << "Hello, World!" << std::endl;
7.  }
```

```
1.  //-- file2.cpp
2.  void greet();
3.
4.  int main()
5.  {
6.      greet();
7.      return 0;
8.  }
```

Примеры объявлений/определений

```
1.  //-- file1.cpp
2.  int uno = 1;
3.  int due = 2;
4.  extern int tre;
```

```
1.  //-- file2.cpp
2.  int uno;
3.  extern double due;
4.  extern int tre;
5.
6.  int main()
7.  {
8.      due = 2.71;
9.      tre = 3;
10.     return 0;
11. }
```

Примеры объявлений/определений

```
1.  //-- file1.cpp
2.  int uno = 1;
3.  int due = 2;
4.  extern int tre;
```

```
1.  //-- file2.cpp
2.  int uno;
3.  extern double due;
4.  extern int tre;
5.
6.  int main()
7.  {
8.      due = 2.71;
9.      tre = 3;
10.     return 0;
11. }
```

```
1.  1>file2.obj : error LNK2005: "int uno" (?uno@@3HA) already defined in file1.obj
2.  1>file2.obj : error LNK2001: unresolved external symbol "int tre" (?tre@@3HA)
3.  1>file2.obj : error LNK2001: unresolved external symbol "double due" (?due@@3NA)
```

Внешняя компоновка (external linkage)

```
1.  //-- file1.cpp
2.  extern int uno;
3.  int foo(int bar);
```

```
1.  //-- file2.cpp
2.  int uno = 5;
3.  int foo(int bar)
4.  {
5.      return bar + 5;
6.  }
```

- Имя используется в другой единице трансляции нежели ее определение

Внутренняя компоновка (internal linkage)

```
1.  //-- file1.cpp
2.  static int uno = 5; // be careful with ‘static’ – too many meanings
3.  const double due = 2.71;
4.
5.  namespace
6.  {
7.      int answer()
8.      {
9.          return 42;
10.     }
11. }
```

```
1.  //-- file2.cpp
2.  extern double due;
3.  int answer()
4.  {
5.      return 43;
6.  }
7.
8.  int main()
9.  {
10.     due = 5; // error LNK2001: unresolved external symbol "double due" (?due@@3NA)
11.     return 0;
12. }
```

Заголовочные файлы (headers)

```
1.  //-- header1.h
2.  int factorial(int n);
```

```
1.  //-- source1.cpp
2.  #include "header1.h"
3.  int factorial(int n)
4.  {
5.      int res = 1;
6.      while (n > 1)
7.          res *= n--;
8.
9.      return res;
10. }
```

```
1.  //-- source2.cpp
2.  #include <iostream>
3.  #include "header1.h"
4.
5.  int main()
6.  {
7.      std::cout << factorial(6) << std::endl;
8.      return 0;
9.  }
```

Что включать в header?

- Включать все то, что планируется использовать в нескольких единицах трансляции:
  - объявления типов `struct vector{int x; int y;};`
  - определение/объявление шаблонов
  - объявление функций
  - определение встроенных (`inline`) функций и переменных (C++17)
  - объявления (только!) переменных
  - определения констант, …

А что нет?

- Может привести к ошибкам:
  - определение функций
  - определение данных
  - анонимные пространства имен

- Типичная ошибка (вывод MSVC):
  - `error LNK2005: "int a" (?a@@3HA) already defined in source1.obj`

Встроенные функции

```
1.  //-- header1.h
2.  inline int factorial(int n)
3.  {
4.      int res = 1;
5.      while (n > 1)
6.          res *= n--;
7.
8.      return res;
9.  }
```

- Выбирается одна из всех единиц трансляции (поэтому должны быть одинаковы)
- Может встраиваться в код для оптимизации (по усмотрению компилятора)

One Definition Rule

- Два определения одной и той же встроенной функции (класса или шаблона) в разных единицах трансляции:
  - должны совпадать лексема за лексемой
  - смыл лексем должен быть одинаков

One Definition Rule

- Два определения одной и той же встроенной функции (класса или шаблона) в разных единицах трансляции:
  - должны совпадать лексема за лексемой
  - смыл лексем должен быть одинаков
- Пишите самодостаточные заголовочные файлы

```
1.  //-- source1.cpp
2.  struct vector {int x; int y;};
3.  #include "header1.h"
```

```
1.  //-- source2.cpp
2.  struct vector {double x; double y;};
3.  #include "header1.h"
```

```
1.  //-- header1.h
2.  inline double length(vector v) {/*...*/ }
```

“Глобальные переменные” в хидере*

- Что, если у Вас нет cpp-файла для определения переменной?

“Глобальные переменные” в хидере*

- Популярный трюк, если у Вас нет cpp-файла для определения переменной:

```
1.  //-- header1.h
2.  inline double& get_global_time()
3.  {
4.      static double time = init_time();
5.      return time;
6.  }
7.
8.  // or no trick from C++17
9.  inline double global_time = init_time();
```

Стражи включения

- Лучше делать самодостаточные хедеры
  - не переусердствуйте – увеличиться время компиляции
  - библиотечные хедеры включайте в precompiled header*
- Избегайте повторного включения:

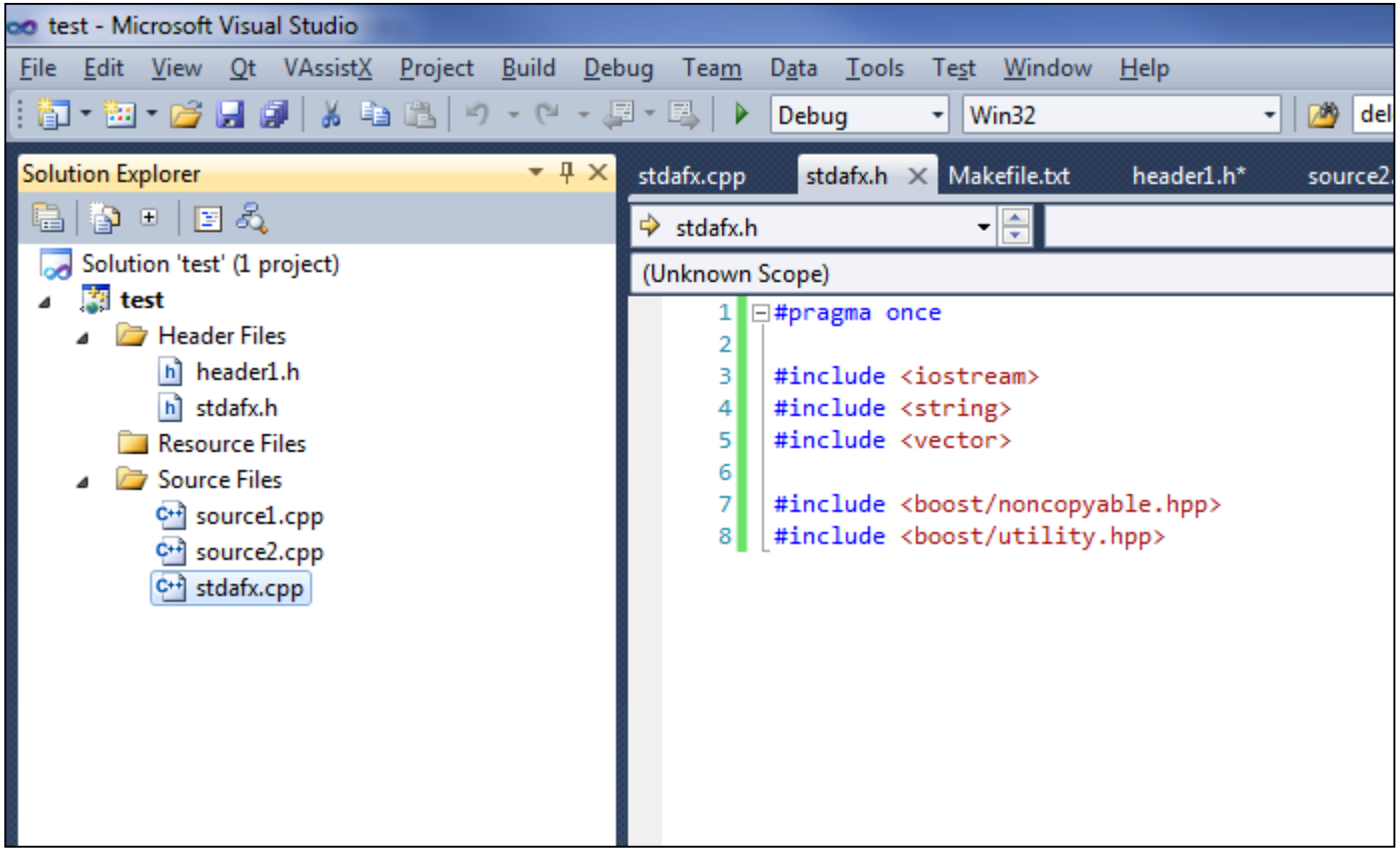
```
1.  //-- header1.h
2.  #ifndef __HEADER_1__
3.  #define __HEADER_1__
4.
5.  ...
6.
7.  #endif
8.
9.  //-- header2.h
10. #pragma once
11. ...
```

Инициализация глобальных переменных

- Если не указана инициализирующая часть, присваивается значение типа по умолчанию (ноль)
- Инициализируются в порядке описания в единице трансляции
- Между разными единицами трансляции порядок инициализации не определен

- Рекомендация: старайтесь не использовать глобальные переменные. Если очень надо – лучше трюк с `inline` функцией.

Precompiled headers*



- Позволяют значительно сэкономить время компиляции для библиотечных хедеров
- По всей видимости, модули C++20/23 их заменят

Вопросы?