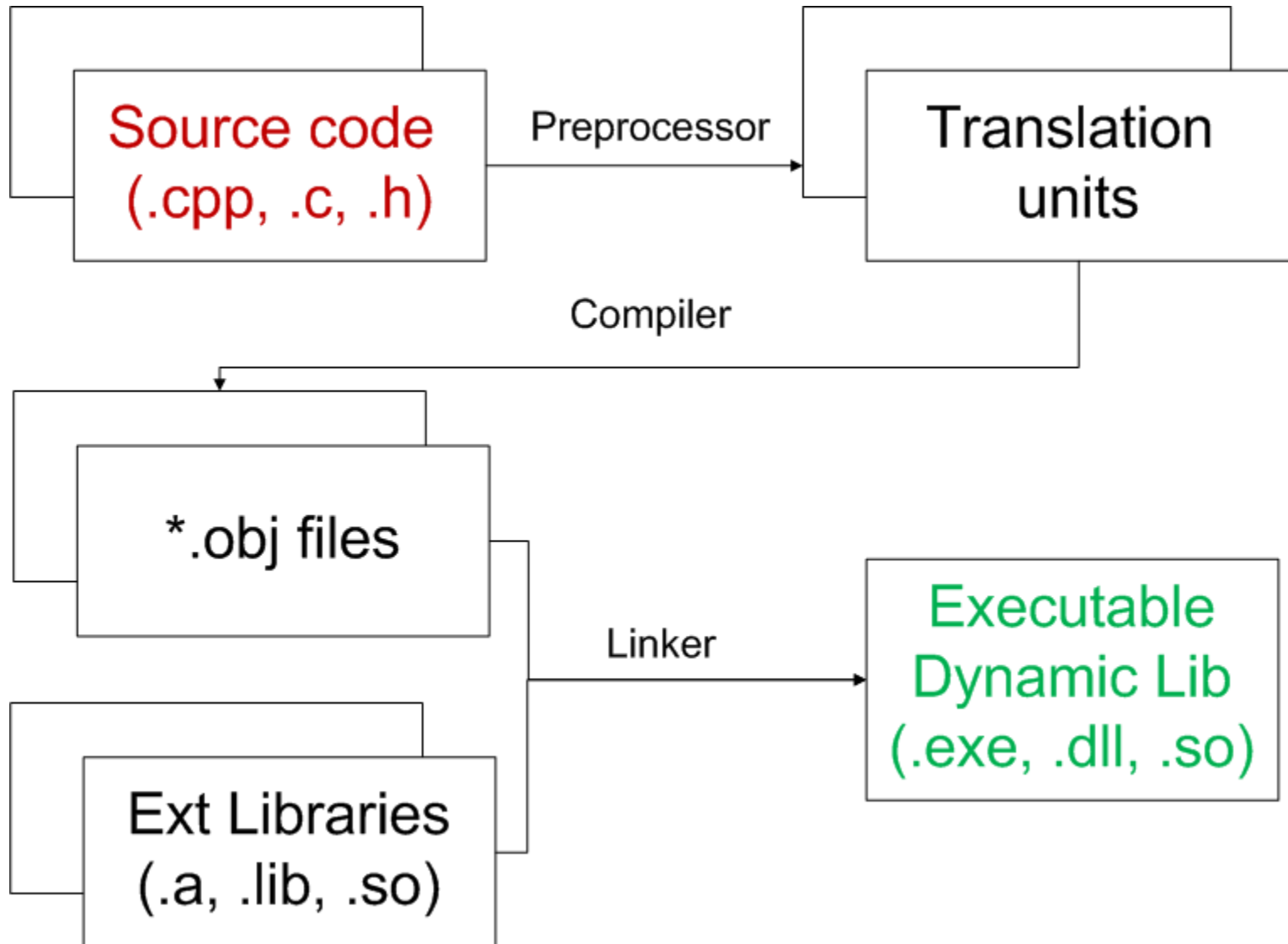


Лекция 2. Сборка C++ программ

Компиляция



Объявления и определения

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

- Объявлений должно быть **хотя бы** одно на программу, но не более одного на единицу трансляции.
- Определение должно быть **ровно** одно на программу.

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
 - 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
 - смысл 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
 - смысл 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 сrr-файла для определения переменной?

“Глобальные переменные” в хидере*

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

```
1.  //-- header1.h
2.  inline double& global_time()
3.  {
4.      static double time = init_time();
5.      return time;
6.  }
```


Компоновка с кодом не на C++

1.	<code>extern "C" int foobar(double value);</code>
2.	<code>extern "C"</code>
3.	<code>{</code>
4.	<code> void foo();</code>
5.	<code> void bar();</code>
6.	<code> extern double x;</code>
7.	<code>};</code>

- Совместимо с C, assembler
- Не меняет правил компиляции, только линковки
- Name mangling:

1.	<code>C language: _foobar@8</code>
2.	<code>C++ language: ?foobar@@YAHN@Z</code>

Пара слов про условную компиляцию

1.	<code>#if</code>
2.	<code>...</code>
3.	<code>#elif</code>
4.	<code>...</code>
5.	<code>#else</code>
6.	<code>...</code>
7.	<code>#endif</code>

1.	<code>#define PI 3.14</code>
2.	
3.	<code>#ifdef PI</code>
4.	<code>...</code>
5.	<code>#endif</code>
6.	
7.	<code>#if defined(PI)</code>
8.	<code>...</code>
9.	<code>#endif</code>

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.

Устаревший механизм модулей

- Заголовочный файл и библиотека – отдельные файлы. Один из них может потеряться или они могут разойтись.
- Включение происходит через `#include`
 - существенно замедляет компиляцию
 - добавляется влияние включаемого заголовочного файла

Новый механизм модулей*

- Быстрая компиляция
- Легко распространяемый пакеты, в т.ч. менеджер пакетов
- Вероятно, появится в C++20 (уже частично реализовано в MSVC и Clang). Ждем!

```
1. import std.io;
2. import std.string;
3.
4. module date;
5. namespace chrono
6. {
7.     export class date
8.     {
9.         /*...*/
10.    };
11.
12.    export string to_string(chrono const&){/*...*/}
13. }
```

Сборка из нескольких файлов

- Можно собрать из командной строки

```
1. g++ main.cpp hello.cpp factorial.cpp -o hello
```

- А можно сделать Makefile

```
1.  цель: зависимости
2.  [tab] команда
3.
4.  #####
5.  all:
6.      g++ main.cpp hello.cpp factorial.cpp -o hello
```

Простой Makefile

- Для сборки достаточно запустить команду `make`

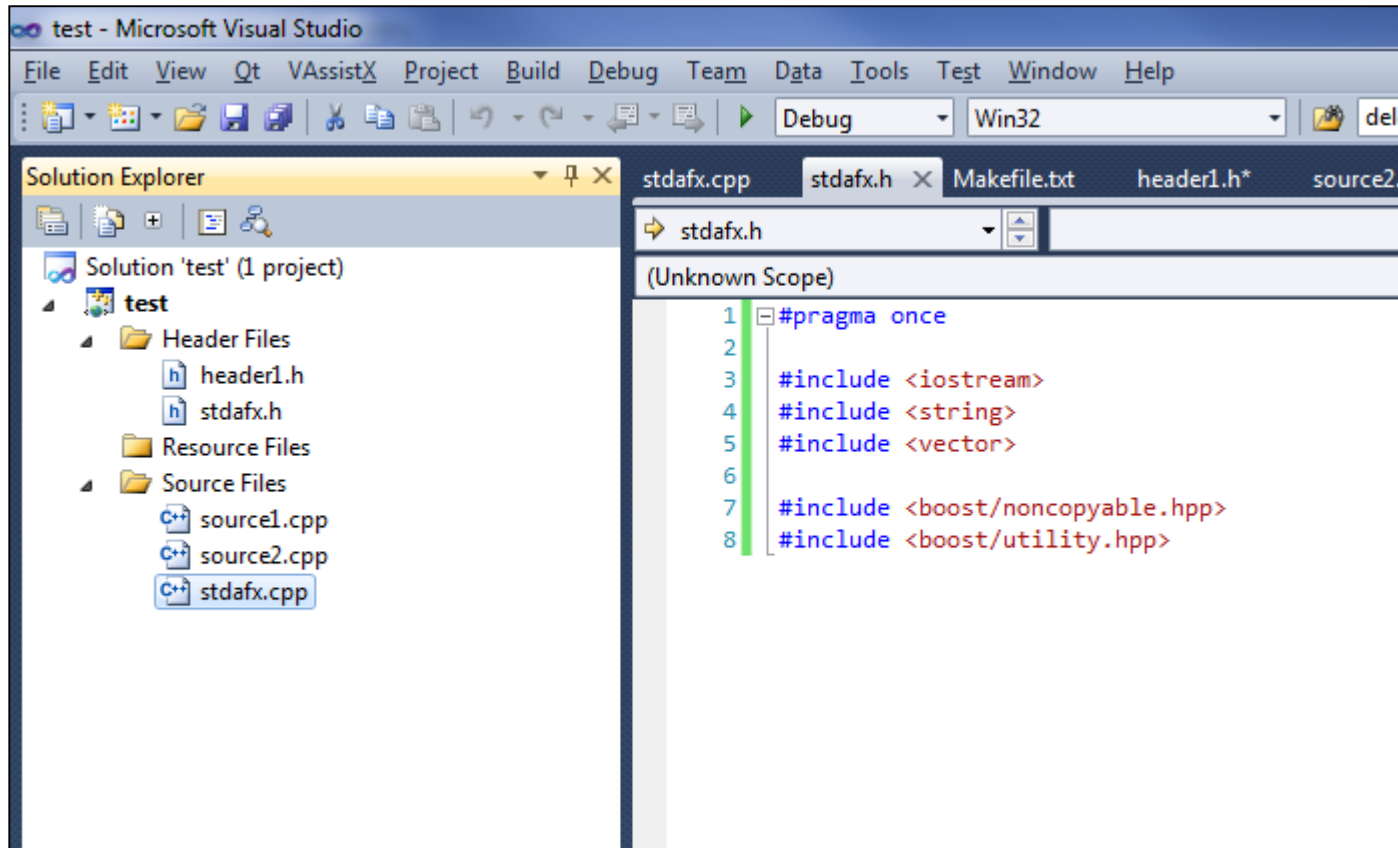
```
1. all: hello
2.
3. hello: main.o factorial.o hello.o
4.     g++ main.o factorial.o hello.o -o hello
5.
6. main.o: main.cpp
7.     g++ -c main.cpp
8.
9. factorial.o: factorial.cpp factorial.h
10.    g++ -c factorial.cpp
11.
12. hello.o: hello.cpp hello.h
13.    g++ -c hello.cpp
14.
15. clean:
16.    rm -f *.o hello
```


Более универсальный Makefile

```
1.  CXX=g++
2.  CXXFLAGS= $(CXXFLAGS) -Wall
3.  objects=main.o hello.o factorial.o
4.
5.  hello: $(objects)
6.      g++ -o hello $(objects)
7.
8.  hello.o: hello.cpp hello.h
9.      $(CXX) -c -o $@ $^ $(CXXFLAGS)
10.
11. factorial.o: factorial.cpp factorial.h
12.     $(CXX) -c -o $@ $^ $(CXXFLAGS)
13.
14. .PHONY: clean
15.
16. clean:
17.     rm -f $(objects)
```

- Указывая флаги компиляции, обязательно делайте их одинаковыми (за исключением спец. случаев)

Precompiled headers*



- Позволяют значительно сэкономить время компиляции для библиотечных хедеров

Вопросы?