ODR, компиляция и линковка

Гусев Илья

Московский физико-технический институт

Москва, 2017

Содержание

Declaration vs definition

Declaration

1. A declaration may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If so, the declaration specifies the interpretation and attributes of these names.
2. Проще говоря, объявление(declaration) - всего лишь имя. Влиять она может лишь на компиляцию, шаблоны и, например, аргументы функций.
3. Примеры объявлений:

```
extern int a; // declares a
extern const int c; // declares c
int f(int); // declares f
struct S; // declares S
typedef int Int; // declares Int
extern X anotherX; // declares anotherX
using N::d; // declares d
```

Declaration vs definition

Declaration

A declaration is a definition unless:

1. contains the extern specifier or a linkage-specification
2. doesn't contain an initializer,
3. doesn't contain function body,
4. a function without specifying the function's body,
5. a static data member in a class definition,
6. a class name declaration,
7. an opaque-enum-declaration,
8. a template-parameter,
9. a parameter-declaration in a function declarator that is not the declarator of a function-definition
10. a typedef declaration,
11. an alias-declaration,
12. a using-declaration,
13. a static_assert-declaration,
14. an attribute declaration,
15. an empty-declaration,
16. a using-directive.

Declaration vs definition

Declaration

Примеры определений:

```
int a; // defines a
extern const int c = 1; // defines c
int f(int x) { return x+a; } // defines f and defines x
struct S { int a; int b; }; // defines S, S::a, and S::b
struct X { // defines X
    int x; // defines non-static data member x
    static int y; // declares static data member y
    X(): x(0) { } // defines a constructor of X
};
int X::y = 1; // defines X::y
enum { up, down }; // defines up and down
namespace N { int d; } // defines N and N::d
namespace N1 = N; // defines N1
X anX; // defines anX
```

Declaration vs definition

Больше примеров

```
// This is the definition of a uninitialized global variable
int x_global_uninit;
// This is the definition of a initialized global variable
int x_global_init = 1;
// This is the definition of a uninitialized global variable,
// albeit one that can only be accessed by name in this C file
static int y_global_uninit;
// This is the definition of a initialized global variable,
// albeit one that can only be accessed by name in this C file
static int y_global_init = 2;
// This is a declaration of a global variable that exists somewhere
// else in the program
extern int z_global;
// This is a declaration of a function that exists somewhere else
// in the program (you can add "extern" beforehand if you like,
// but it's not needed)
int fn_a(int x, int y);
```

Declaration vs definition

Больше примеров

```
// This is a definition of a function, but because it is marked as
// static, it can only be referred to by name in this C file alone
static int fn_b(int x)
{
  return x+1;
}
// This is a definition of a function.
// The function parameter counts as a local variable
int fn_c(int x_local)
{
  // This is the definition of an uninitialized local variable
  int y_local_uninit;
  // This is the definition of an initialized local variable
  int y_local_init = 3;
  // Code that refers to local and global variables and other
  // functions by name
  x_global_uninit = fn_a(x_local, x_global_init);
  y_local_uninit = fn_a(x_local, y_local_init);
  y_local_uninit += fn_b(z_global);
  return (y_global_uninit + y_local_uninit);
}
```

One definition rule

1. A source file together with all the headers and source files included via the preprocessing directive include, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a **translation unit**.
2. **No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.**
3. Every program shall contain **exactly one definition** of every non-inline function or variable that is odr-used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined. An inline function shall be defined in every translation unit in which it is odr-used.
4. Переменная считается odr-used, если она находится в потенциально вычисляемом выражении.

One definition rule - complete class types

Exactly one definition of a class is required in a translation unit if the class is used in a way that requires the class type to be complete.
The following complete translation unit is well-formed, even though it never defines X:

```
struct X; // declare X as a struct type
struct X* x1; // use X in pointer formation
X* x2; // use X in pointer formation
```

One definition rule - исключения

There can be more than one definition of:

1. a class type
2. enumeration type
3. inline function with external linkage
4. class template
5. non-static function template static data member of a class template
6. member function of a class template
7. template specialization for which some template parameters are not specified

in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements:

One definition rule - исключения

1. каждое определение должно состоять из одной и той же последовательности токенов;
2. имена, использованные в определении (в т.ч. использованные неявно), должны относиться к одним и тем же сущностям, или быть определены в самом определении;
3. в каждом определении сущности должны иметь одно и то же связывание;
4. если в определении есть вызов функции со значением аргумента по умолчанию, считается что токены значения вставляются в определение, и таким образом подпадают под правила, написанные выше;
5. если определение - это класс с неявно-определенным конструктором, он должен вызывать одни и те же конструкторы своих членов и базовых классов.

One definition rule - примеры

```
// Единица трансляции 1:          |     // Единица трансляции 2:
int x = 1;                        |     int x = 1;
int y = 1;                        |     int y = 1;
decltype(x) xx; // OK, "x" использована в не-вычисляемом контексте.
// нарушение ODR, "y" определена в двух единицах
// трансляции и ODR-использована.
int z = y;
```

One definition rule - примеры

```
// Единица трансляции 1:          |     // Единица трансляции 2:
int x = 1;                        |     int x = 1;
int y = 1;                        |     int y = 1;
decltype(x) xx; // OK, "x" использована в не-вычисляемом контексте.
// нарушение ODR, "y" определена в двух единицах
// трансляции и ODR-использована.
int z = y;
```

One definition rule - примеры

Разные конструкторы базовых классов при вызове неявно определенного конструктора D::D().

```
// Единица трансляции 1:
struct X {
  X(int);
  X(int, int);
};

X::X(int = 0) { }

class D: public X { };
```

```
// Единица трансляции 2:
struct X {
  X(int);
  X(int, int);
};

X::X(int = 0, int = 0) { }

class D: public X { };
```

One definition rule - примеры

"Имена, использованные в определении должны относиться к одним и тем же сущностям"

```
// g.h
void f(float t);
inline void g() { f(1); }
```

```
// f_int.h
void f(int);
```

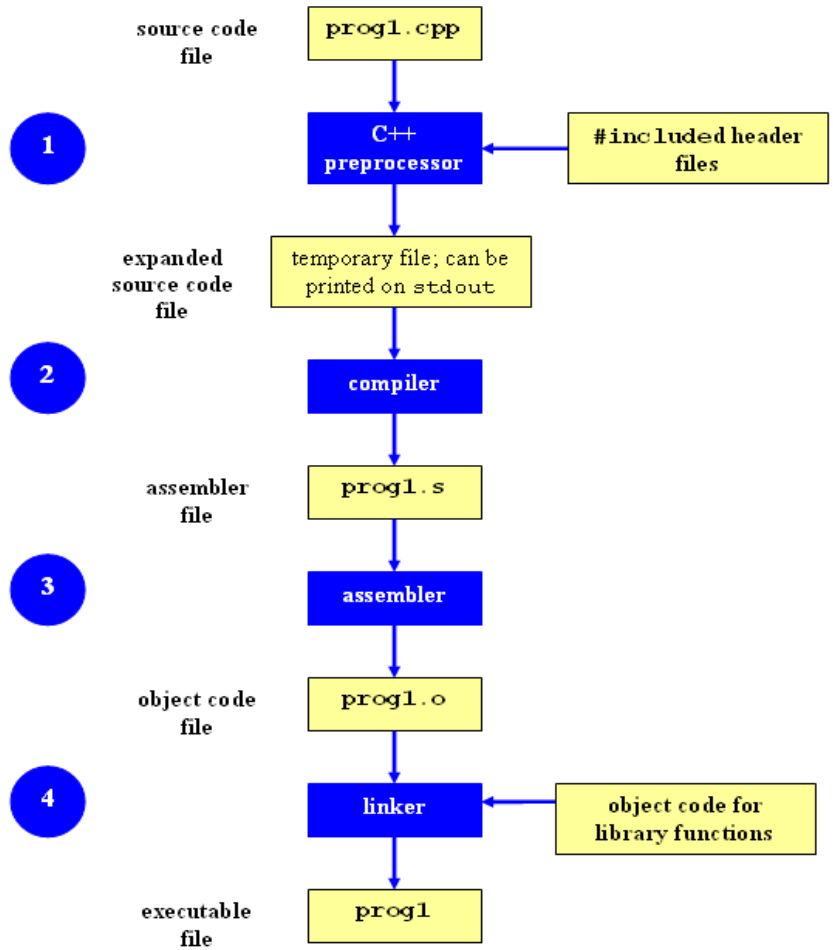
```
// main.cpp
#include "g.h"
int main() {
  g();
}
```

```
// f.cpp
#include "g.h"
#include "f_int.h"
void f(int) {}
void f(float t) {}
```

Общая схема

g++ prog1.cpp

1. Препроцессор копирует содержимое включённых файлов в исходный код, разворачивает макросы и подставляет константы, переопределённые с помощью #define.
2. Развёрнутый исходный код компилируется в платформозависимый ассемблер (.s). Включение вывода: -S
3. Ассемблерный код собирается в объектный код платформы (.o). Смотреть: nm, objdump.
4. Объектный код связывается с другими объектными файлами и библиотеками.

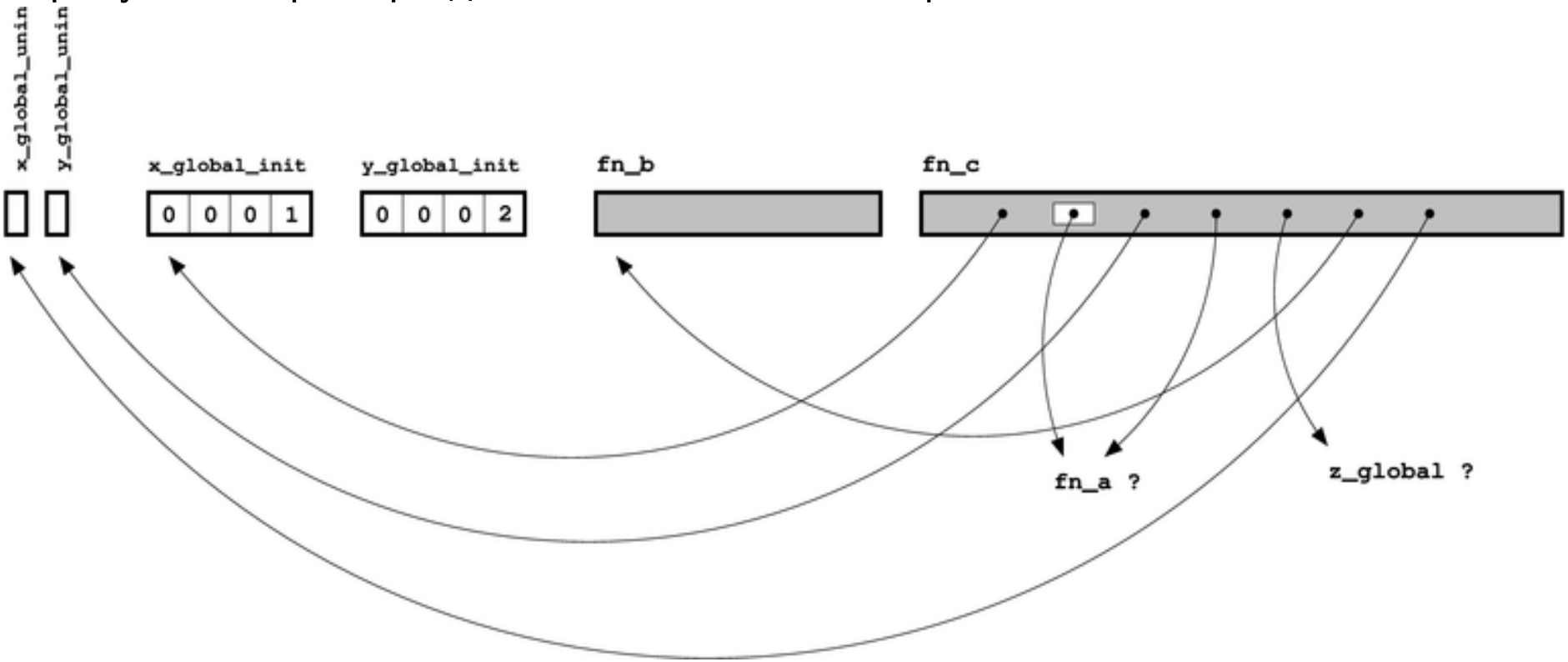


Компиляция

1. Токенизация
2. Построение промежуточного дерева по грамматике
3. Построение abstract-syntax tree (AST)
4. Построение платформо-зависимого дерева
5. Создание ассемблерного кода.

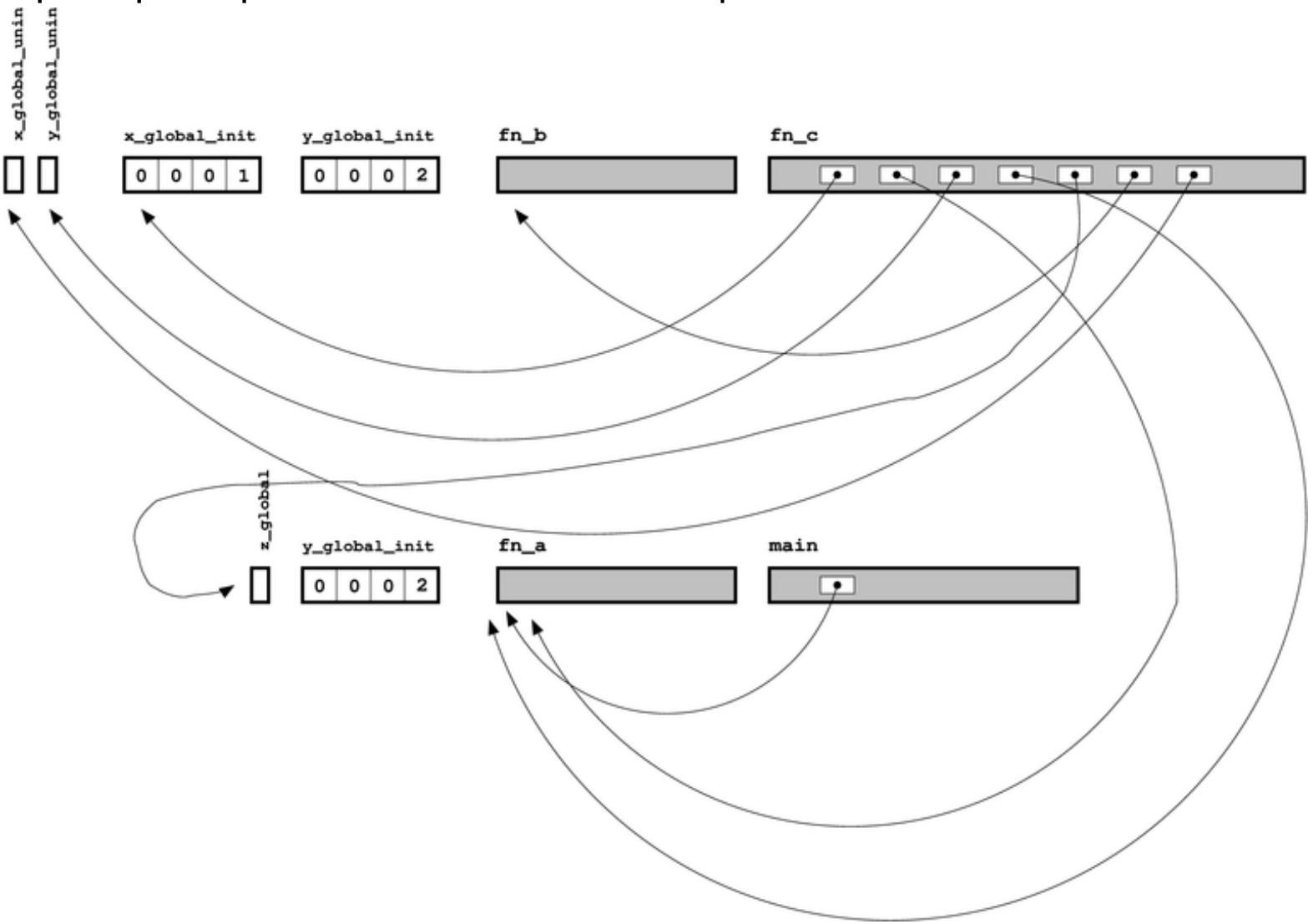
Линковка

На выходе с компилятора приходят объектные файлы. Они ничего не знают о существовании друг друга и их надо объединить в один, заполнив их "пропуски". Пример одиночного объектного файла:



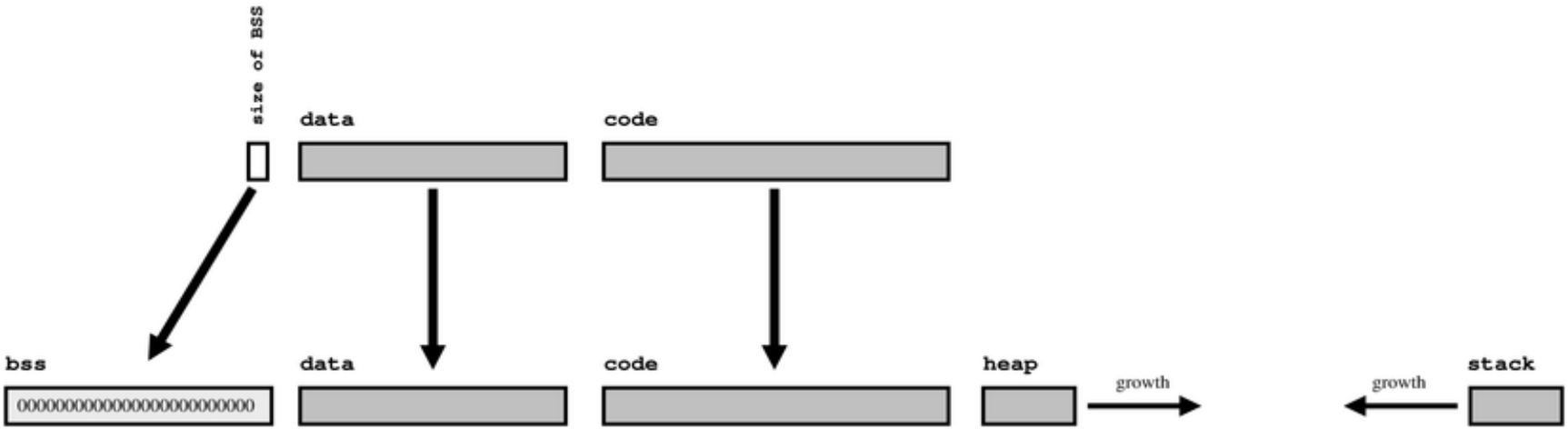
Линковка - собранный файл

Пример собранного исполняемого файла:



Загрузка исполняемого файла

1. Глобальные переменные кладутся в память процесса as is.
2. Локальные переменные кладутся на стек, который растёт при вызовах функций и уменьшается при их завершении.
3. Динамические переменные расходуют память кучи, системные malloc и free управляют этой частью процесса.
4. bss - фрагмент неинициализированных пременных, заполнен нулями
5. Кроме того, на затронут mapping файлов в память и пространство ядра (kernel space).



Статические библиотеки

1. Самая простая версия библиотек
2. Linux: lib*.a, Windows: *.lib
3. Библиотека может состоять из нескольких .o файлов.
4. Библиотека участвует в линковке объектными файлами, то есть тянется не вся библиотека, а только объектники, которые заполняют какой-либо 'пропуск'. При этом, тянется всё, что есть в этом объектном файле, в том числе новые 'пропуски'.
5. Линковка библиотек только после линковки частей программы, и только если есть 'пропуски'.
6. Библиотеки линкуются строго по порядку. Если вдруг библиотека-2 (идущая после библиотеки-1) требует что-то, что есть в незагруженном объектнике библиотеки-1, линкер это что-то не найдёт!

Проблемы статических библиотек

1. Каждый исполняемый файл содержит копию библиотеки $\rightarrow$ исполняемые файлы занимают кучу места.
2. Если мы хотим поменять код библиотеки нужно перелинковывать ВСЕ исполняемые файлы.

Shared библиотеки

1. *.so, *.dll, *.dylib.
2. Линкер просто оставляет некоторые 'пропуски' открытыми, записывая, что они должны заполняться из такой-то shared библиотеки.
3. Код библиотеки не включается в исполняемый файл.
4. Во время запуска программы (до исполнения main) операционная система натравливает маленькую версию линкера (ld.so) на оставшиеся 'пропуски'.
5. Если мы захотим поменять код, скажем, printf - нам не нужно перелинковывать все исполняемые файлы.
6. Загрузка shared библиотек происходит целиком, а не по объектным файлам!
7. *ldd* <имя исполняемого файла> чтобы посмотреть, какие библиотеки он захватывает.

Windows DLL

1. Нужно явно указывать символы для экспорта одним из 3 вариантов:
   1. __declspec(dllexport) int my_exported_function(int x, double y);
   2. LINK.EXE /dll /export:my_exported_function
   3. LINK.EXE /dll /DEF:def_file
2. Инормация об экспортируемых символах и их местонахождении хранится в .lib файле (не статическая библиотека, другое!). Этот файл нужен исполняемому, чтобы корректно слинковать DLL.
3. Можно ещё задавать импорты для лучшей оптимизации:
   1. __declspec(dllimport) int function_from_some_dll(int x, double y);
   2. __declspec(dllimport) extern int global_var_from_some_dll;

Windows DLL

Что может получиться после линковки:

1. library.DLL: код и данные библиотеки, непосредственно используется исполняемым файлом.
2. library.LIB: файл, в котором описываются экспортируемые DLL символы.
3. library.EXP: ещё один файл с описаниями симоловов, нужен для разрешения циклических зависимостей.
4. library.ILK: при опции линкера /INCREMENTAL, хранит состояние инкрементальной компоновки, для ускорения перекомпиляции.
5. library.PDB: при опции линкера /DEBUG, содержит отладочные данные и сведения о состоянии проекта, например информацию о типах.
6. library.MAP: при опции линкера /MAP, хранит расположение функций и данных в DLL.

Windows DLL

Что может быть до линковки:

1. library.LIB: файлы других библиотек, в которых описываются экспортируемые ими символы.
2. library.LIB: статические библиотеки - набор объектных файлов. Неоднозначное расширение :)
3. library.DEF: определение библиотеки, контроль некоторых настроек библиотеки, в частности - в нём можно задавать экспортируемые символы.
4. library.EXP: файл с описаниями симоловов линкуемых библиотек, нужен для разрешения циклических зависимостей.
5. library.ILK: см. выше.
6. library.RES: ресурсный файл с разными GUI штуками, локализациями и всем таким, это всё нужно исполняемому файлу

Динамически загружаемые библиотеки (настоящие DLL)

1. Загрузка бибилиотеки в любом месте программы, не только до main.
2. dlopen и dlsym (или LoadLibrary и GetProcAddress).
3. dlopen - берёт библиотеку и загружает её в память процесса.
4. dlsym - берёт имя символа и возвращает указатель на его местонахождение в в загруженной области памяти.

Перегрузка функций

1. В Си нельзя перегружать функции, в C++ можно.
2. Пример: int max(int x, int y); float max(float x, float y);
3. А что делать линкеру, он же по имени символы ищет?

Mangling

1. Решение:
   1. int max(int x, int y) -> имя - _Z3maxii
   2. float max(float x, float y) -> имя - _Z3maxff
2. Всё сложнее для классов.
3. И ещё сложнее для шаблонов.
4. extern “C” у объявления и определения - убирает mangling для совместимости кода на Си и C++.
5. extern “C” игнорируется методами класса.

Конструкторы статических объектов

1. В Си инициализация статических объектов очень простая - просто копируем данные из data секции в память.
2. В C++ появились конструкторы.
3. Компилятор обязан составить список конструкторов, которые нужно вызвать перед запуском программы и включить в программу код, который все эти конструкторы вызовет.
4. В рамках одной единицы трансляции порядок конструирования закреплён.
5. Но в каком порядке конструкторы будут вызываться из разных единиц трансляции - неизвестно, будьте осторожны, особенно если они друг от друга зависят!

Полезные ссылки I

C++11 n3690
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf

Статья про линковку
https://www.lurklurk.org/linkers/linkers.htmlcfile

Habr: ODR
https://habrahabr.ru/company/abbyy/blog/108166/

SO: ODR
https://ru.stackoverflow.com/questions/418755/Что-такое-Правило-одного-определения-one-definition-rule

The C++ compilation process
http://faculty.cs.niu.edu/ mcmahon/CS241/Notes/compile.html

Хабр: Организация памяти
https://habrahabr.ru/company/smart$_s$*oft*/*blog*/185226/