# Основы алгоритмизации и программирования

Пашук Александр Владимирович

pashuk@bsuir.by

# Мем в начале

# Содержание лекции

1. Многофайловые программы
2. Причины использования многофайловых программ
3. Взаимодействие между файлами
4. Примеры
5. Вопросы из теста

# Причины использования

- Разделение кодовой базы
- Разделение работы над проектом на несколько разработчиков
- Удобство организации архитектуры программы
- Сокращение кодовой базы
- Упрощение внесения правок

# Взаимодействие между файлами

- Взаимодействие между скомпилированными по отдельности, но скомпонованными вместе исходными файлами (.cpp)
- Взаимодействие с заголовочными файлами (.h)

# Взаимодействие исходных файлов

Три основных элемента исходных файлов:

- Переменные
- Функции
- Классы

Для каждого типа элементов есть свои правила межфайлового взаимодействия.

# Межфайловые переменные

Объявление переменной – декларация её типа и имени.

Переменная определяется, когда происходит резервация места под не в памяти.

```
int some_var;
extern int some_var;
```

# Межфайловые переменные

```
// File A
int global_var;

// File B
global_var = 3; // Error!
```

# Межфайловые переменные

```
// File A
int global_var;

// File B
extern int global_var; // Declaration in File B
global_var = 3; // Good!

// But:
extern int global_var = 27; // extern will be ignored!
```

# Статические переменные

```
// File A
static int global_var; // Only visible in A

// File B
static int global_var; // Only visible in B
```

В этом случае статическая переменная имеет **внутреннее связывание**. Нестатические глобальные переменные имеют **внешнее связывание**.

Для сужения области видимости можно использовать также **пространства имен**.

Полезно: Внутренняя и внешняя линковка в C++

# Совет

- Не использовать глобальных переменных
- Если очень хочется, и по логике программы переменная используется только в текущем файле, то лучше сделать переменную статической.

В контексте глобальных переменных слово `static` просто **сужает область видимости до одного файла**.

# Константы

Переменная, определенная с помощью const, в общем случае не видна за пределами одного файла.

Но если очень нужно:

```
// File A
extern const int const_var = 99;

// File B
extern const int const_var;
```

# Межфайловые функции

**Объявление функции** задает ее имя, тип возвращаемых данных и типы всех аргументов.

**Определение функции** — это объявление плюс тело функции (тело функции — код, содержащийся внутри фигурных скобок).

Всё, что нужно знать при вызове – это имя, тип и типы аргументов. Все это есть в объявлении функции.

# Пример

```
// File A
int add(int a, int b) {
  return a + b;
}

// File B
int add(int, int);

int main() {
  cout << add(10, 10) << endl;
  return 0;
}
```

# Заголовочные файлы

Указанный после директивы `#include` файл просто вставляется в исходный (например, `iostream`).

Пример:

```
#include <iostream>
#include "add.h"
```

# Заголовочные файлы

```
// sum.h
extern int not_found = 404;
int sum(int, int);

// sum.cpp
int sum(int a, int b) { return a + b; }

// main.cpp
#include "sum.h"

int main() {
  cout << sum(10, 10) << endl;
  cout << not_found << endl;
  // 20
  // 404
}
```

# Заголовочные файлы

В заголовочном файле можно **хранить объявления, но не определения переменных или функций**.

В противном случае, использование этого файла в других (если только данные не `static` или `const`) приведет к ошибке компоновщика «повторные определения».

# Ошибка повторения включений

```
// example.h
#if !defined(EXAMPLE_H)
#define EXAMPLE_H

int global_var;
int sum(int a, int b) {
  return a + b;
}
#endif
```

# Пространства имен

Пространства имен предоставляют более гибкий подход к вопросу управления видимостью переменных и функций.

**Пространство имен** — это некая именованная область файла.

# Пример

```
namespace geo {
  const double PI = 3.14159;
  double L(double radius) {
    return 2 * PI * radius;
  }
}

int main() {
  cout << geo::PI << endl;
  cout << geo::L(10) << endl;
  // 3.14159
  // 62.8318
}
```

# Пример: using

```
namespace geo {
  const double PI = 3.14159;
  double L(double radius) {
    return 2 * PI * radius;
  }
}

L(100); // Not works!

using namespace geo;

L(100); // Works!
```

# Неоднократное определение пространств имен

```
namespace geo {
  const double PI = 3.14159;
}

// Other code ...

namespace geo {
  double L(double radius) {
    return 2 * PI * radius;
  }
}
```

# Пример

```
// geo.h
namespace geo {
  const double PI = 3.14159;
  double L(double radius);
}

// geo.cpp
#include "geo.h«
double geo::L(double radius) { return 2 * PI * radius; }

// main.cpp
#include "geo.h«
int main() {
  cout << geo::PI << endl;
  cout << geo::L(10) << endl;
}
```

# Статические и динамические библиотеки

**Библиотека** — это фрагмент кода, который можно многократно использовать (переиспользовать) в разных программах.

Библиотека в C++ состоит из:

- Заголовочный файл, который объявляет функционал библиотеки.
- Предварительно скомпилированный бинарный файл, содержащий реализацию функционала библиотеки.

Есть 2 типа библиотек: **статические и динамические**.

# Статическая библиотека

**Статическая библиотека** состоит из подпрограмм, которые непосредственно компилируются и линкуются с разрабатываемой программой.

При компиляции программы, которая использует статическую библиотеку, весь необходимый функционал статической библиотеки становится частью исполняемого файла.

В Windows статические библиотеки имеют расширение `.lib` (library), в Linux `.a` (archive).

# Статическая библиотека

**Преимущество:**

- Всего лишь один (исполняемый) файл, чтобы пользователи могли запустить и использовать программу. Статические библиотеки становятся частью программы

**Недостатки:**

- Копия библиотеки становится частью каждого исполняемого файла, что может привести к увеличению размера файла
- Для обновления статической библиотеки придется перекомпилировать каждый исполняемый файл, который её использует

# Динамическая библиотека

**Динамическая библиотека** состоит из подпрограмм, которые подгружаются в разрабатываемую программу во время её выполнения.

При компиляции программы, которая использует динамическую библиотеку, эта библиотека не становится частью исполняемого файла.

В Windows динамические библиотеки имеют расширение `.dll` (dynamic link library), в Linux `.so` (shared object).

# Динамическая библиотека

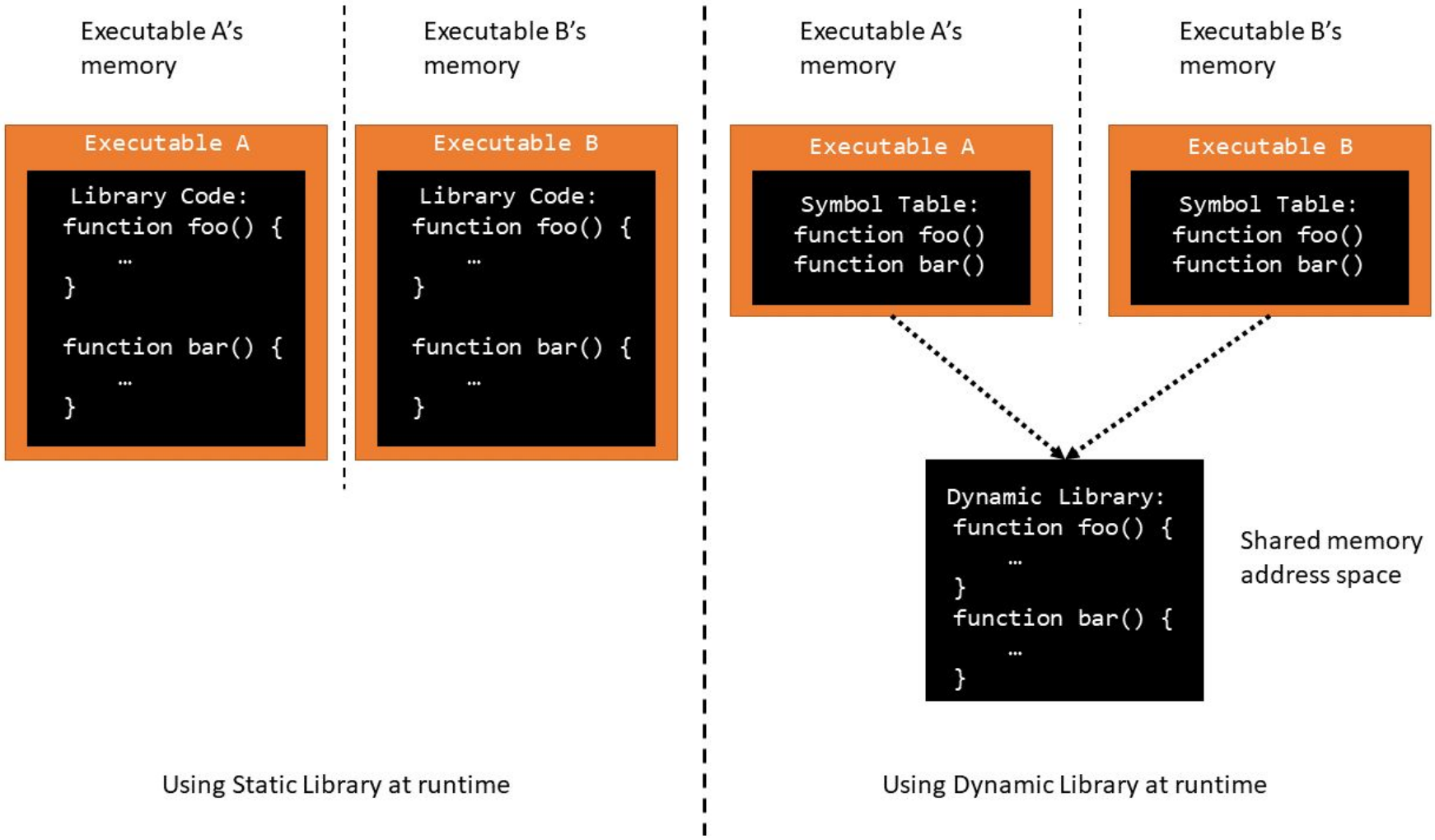
**Преимущества:**

- Разные программы могут совместно использовать одну копию динамической библиотеки, что значительно экономит используемое пространство
- Можно обновить до более новой версии без необходимости перекомпиляции всех исполняемых файлов, которые её используют.

**Недостаток:**

- Механизм взаимодействия может быть непонятен для новичков

# Использование библиотек



# Библиотеки импорта

**Библиотека импорта** (import library) – это библиотека, которая автоматизирует процесс подключения и использования динамической библиотеки.

- В Windows это обычно делается через небольшую статическую библиотеку (.lib) с тем же именем, что и динамическая библиотека (.dll).
- В Linux общий объектный файл (с расширением .so) дублируется сразу как динамическая библиотека и библиотека импорта.

# Пример

```
// math.h
#if !defined(MATH_H)
#define MATH_H
int round(double r);
#endif

// math.cpp
#include "math.h"
int round(double r) {
  return (r > 0.0) ? (r + 0.5) : (r - 0.5);
}
```

# Пример

```
#include "math.h"
#include <iostream>

using namespace std;

int main() {
  double number;
  cout << "Enter the number to round: ";
  cin >> number;
  cout << round(number) << endl;
}
```

# Пример: запускаем как обычно

```
g++ -c app.cpp -o app.o
g++ -c math.cpp -o math.o
g++ app.o math.o -o app.out
./app.out


// Enter the number to round: 11.11
// 11
```

# Пример: статическая библиотека

- Получаем файл библиотеки:

```
ar cr libmath.a math.o
```

- Используем полученный файл библиотеки:

```
g++ app.o libmath.a -o app.out
g++ app.o -L. -lmath -o app.out
```

- Проверяем:

```
./app.out
```

# Пример: динамическая библиотека

- Создаем файл динамической библиотеки:

```
g++ -shared -o libmath.so math.o
```

- Используем полученный файл библиотеки:

```
g++ app.o libmath.so -o app.out
g++ app.o -L. -lmath -o app.out
```

- Проверяем:

```
./app.out
./app.out: error while loading shared libraries: libmath.so: cannot open shared object file: No such file or directory
```

# В чем проблема?

Пользователь должен подсказать операционной системе, где искать необходимые библиотеки (`libmath.so`).

В случае с Linux:

1. Добавить путь к библиотеке к переменной окружения `LD_LIBRARY_PATH`
2. Использовать флаг `-rpath`, чтобы указать путь к динамической библиотеке при сборке исполняемого файла

# Исправленный пример

```
export LD_LIBRARY_PATH =
$LD_LIBRARY_PATH:/c/dev/OAIP/static_example/
./app.out
```

Или

```
g++ app.o -L. -lmath -o app.out
-Wl,-rpath,/c/dev/OAIP/static_example/
./app.out
```

# Пример вопроса на экзамене

**Глобальная переменная определена в файле A. Чтобы получить доступ к ней из файла B, необходимо:**

- Определить ее в файле B, используя `extern`;
- Определить ее в файле B, используя `static`;
- Расслабиться;
- Должно быть константой;
- Объявить ее в файле B, используя `extern`.

# Пример задачи на экзамене

**Создать библиотеку для нахождения параметров треугольников (тип треугольника, площадь, периметр и т.п.).**

Использовать заголовочные файлы и пространства имен.

C++ developer learning Python

Python developer learning C++