Языки программирования

Лекция 2

ПМИ 2 курс

Демяненко Я.М.

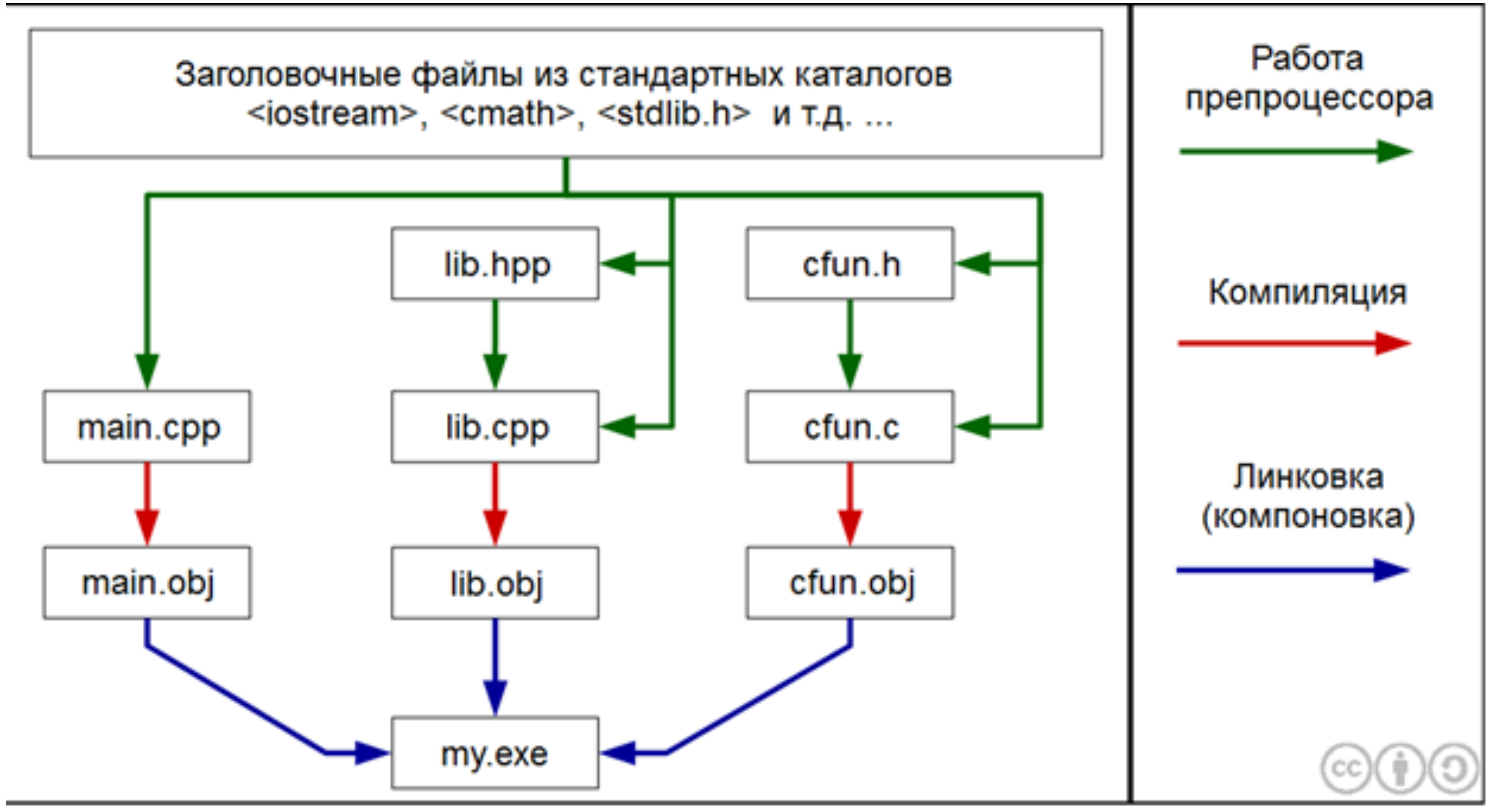
ЮФУ 2023

Многофайловая компоновка

В C++ имеет место независимая компиляция:
все файлы проекта компилируются независимо один от другого.

Компиляция состоит из

- этапа собственно компиляции
- этапа линковки



Заголовочные файлы

Содержат заголовки всех функций и объявления переменных, обычно имеют расширение *.h (header).
Теперь можно вынести объявления функций из всех файлов в один (заголовочный):

```
/* myheader.h */
extern int n;
void f(int);

/* myheader.cpp */
#include <iostream>
int n;

void f(int i) {
   std::cout << i;
}
```

```
/* a.cpp */
#include "myheader.h"

void main() {
   n = 5; f(n);
}

/* b.cpp */
#include "myheader.h"

void makeZero() {
   n = 0;
}
```

Имена пользовательских заголовочных файлов в директиве include заключаются в двойные кавычки, а имена стандартных заголовочных файлов — в угловые скобки.

Стандартные заголовочные файлы расположены в /INCLUDE.
Поиск пользовательских файлов производится в текущем каталоге.

Замечание: inline-функции не сохраняются в исходном коде, так как больше не используются (а сразу встраиваются на место вызова).

Чтобы воспользоваться такими функциями в другой единице компиляции,
их нужно поместить в заголовочный файл.

Содержимое заголовочных файлов

Что может содержать заголовочный файл:	Пример
Определения типов	struct point { int x, y; };
Шаблоны (типов и функций)	template<class T> class V { /* ... */ }
Объявления функций	int strlen(const char*);
Определения встраиваемых функций	inline char get() { return *p++; }
Объявления переменных	extern int a;
Определения констант	const float pi = 3.141593;
Перечисления	enum bool { false, true };
Объявления имен (типов)	class Matrix;
Команды включения файлов	#include <iostream>
Макроопределения	#define Case break;case
Комментарии	/* проверка на конец файла */

В заголовочном файле никогда не должно быть:	Пример
Определений обычных функций	char get() { return *p++; }
Определений данных	int a;
Определений составных констант	const tb[i] = { /* ... */ };

Глобальные описания в C++ и пространства имен

Для создания локального пространства имен используется ключевое слово namespace

```
namespace MyNamespace{
   ... // содержимое пространства имён: типы, функции, что-угодно…
}
```

Прототипы функций и глобальные переменные в заголовочных файлах, особенно в крупных проектах, необходимо заключать в пространства имен

```
/* header.h */
namespace MyNamespace{
   extern int n;
   void f();
}
```

Основные этапы сборки проекта

1. Препроцессирование
2. Компиляция каждого *.cpp-файла в объектный код (файлы *.obj или *.o)
3. Линковка — сборка всех объектных файлов в один исполняемый (*.exe)

Линковка

Ошибки во время линковки

- Одинаковые объявления в одном пространстве имен
- Ошибки при использовании #include "*.cpp" (грубая ошибка)
- Отсутствие определения функции
- Отсутствие main() во всех файлах проектах
- Несколько объявлений main()

Особенности линковки

- Константы имеют внутреннюю линковку
- inline-функции «погибают» при компиляции

Основные директивы препроцессора

#include — вставляет текст из указанного файла

#define — задает макроопределение (макрос) с параметрами или без (во втором случае это просто символическая константа)

#undef — отменяет предыдущее определение

#ifdef — осуществляет условную компиляцию при определённости макроса

#ifndef — осуществляет условную компиляцию при неопределённости макроса

#else — ветка условной компиляции при ложности выражения

#endif — окончание условной компиляции

Условная компиляция

```
#define FLAG1

#ifdef FLAG1
  // Код, помещённый здесь, откомпилируется только, если определена макроподстановка FLAG1

#else
  // Данный код откомпилируется, если макроподстановка FLAG1 не определена
/* FLAG1 */
```

Стражи включения (include guards)

include guards — шаблонная конструкция (клише), вставляемая в заголовочный файл.

```
#ifndef FILENAME_H
#define FILENAME_H

// Содержимое заголовочного файла

#endif /* FILENAME_H */
```

Такая конструкция защищает проект от повторного включения прототипов функции и зацикливания на этапе препроцессирования, что, в свою очередь, приводит к сокращению времени компиляции, а в случае зацикливания к корректной сборке проекта.

file «grandfather.h»

```
#pragma once

struct foo {
    int member;
};
```

ИЛИ

file «grandfather.h»

```
#ifndef GRAND_H
#define GRAND_H

struct foo {
    int member;
};

#endif GRAND_H
```

file «father.h»

```
#include "grandfather.h"
```

file «child.c»

```
#include "grandfather.h"
#include "father.h"
```

Перечисления в C++98

enum DayOfWeek {MON, TUE, WED, THU, FRI, SAT, SUN};

Каждый элемент перечисления — это целое число: MON = 0, TUE = 1, …
При этом существует возможность явно задавать значения элементов enum. Например:

enum DayOfWeek {MON, TUE = 3, WED, THU, FRI, SAT, SUN};
Тогда MON = 0, TUE = 3, WED = 4, …

Благодаря тому, что все имена перечисления являются целыми числами, возможно следующее присваивание:

```
int day = MON;     // Ok, day == 0;

DayOfWeek dow = 5;  // Ошибка компиляции

DayOfWeek wod1 = (DayOfWeek)5;

DayOfWeek wod2 = static_cast<DayOfWeek>(5);
```

Перечисления в C++11

При объявлении **enum** все имена перечисления экспортируются во внешнюю область видимости, это приводит к проблеме коллизии имен в крупных проектах.

Для решения данной проблемы в C++11 был введен **enum class** — строго типизированные перечисления с ограниченной областью видимости.

Объявление enum class происходит следующим образом:
enum class DayOfWeek {MON, TUE, WED, THU, FRI, SAT = 6, SUN};

Особенности работы с перечислениями в C++11

```
enum class DayOfWeek {MON, TUE, WED, THU, FRI, SAT = 6, SUN};

// Ошибка: WED нет в области видимости
int wday = WED;
// Ошибка при преобразовании DayOfWeek к int
int day = DayOfWeek::WED;

// Присваивание переменной значения из множества имен перечисления DayOfWeek
DayOfWeek dow = DayOfWeek::FRI;

// При необходимости присваивания переменной типа DayOfWeek целого числа
// можно воспользоваться оператором static_cast<type>(object);

DayOfWeek sat = static_cast<DayOfWeek>(6);
```

Использование перечислений

Часто имена перечислений используются в операторе switch.

```
dayOfWeek day;
…
switch (day) {
    case DayOfWeek::MON: cout << "Monday\n"; break;
    case DayOfWeek::TUE: cout << "Tuesday\n"; break;
    case DayOfWeek::WED: cout << "Wedesday\n"; break;
    case DayOfWeek::THU: cout << "Thusday\n"; break;
    case DayOfWeek::FRI: cout << "Friday\n"; break;
    case DayOfWeek::SAT: cout << "Satuday\n"; break;
    case DayOfWeek::SUN: cout << "Sunday\n"; break;
}
```

Массивы. Особенности массивов в C/C++:

- Элементы массивов индексируются с нуля
- **Нет контроля выхода за пределы массива**
- Высокая скорость работы с массивом (как следствие предыдущего)
- Массив не хранит свой размер

Работа с массивами в C/C++

<тип> <имя>[<размерность>];

```
// Создание массива из 5 элементов типа int
int b[5];

// Создание и инициализация массива из 4 элементов
int a[] = {2, 3, 5, 7};

// Запись в последний элемент массива
a[3] = 1;

// Так нельзя копировать, только в цикле
b = a;
```

Размеры массивов в C/C++

Так как массивы в C/C++ не помнят своего размера, то его необходимо определять вручную след. образом:

```
int size = sizeof(arr2) / sizeof(int);

// sizeof(arr2) возвращает размер массива в байтах
// sizeof(int) возвращает размер элемента массива
```

Цикл по массиву (for)

```
 int a[10], n,k;


for(int i=0; i<n; ++i) {
   cout<<i<<" array element ";
   cin>>a[i];
 }
```

Цикл по массиву (foreach)

```
int a[] = {3, 7, 9, 5, 7, 2, 7};

for (int x : a)     // Доступ к x только на чтение
   cout << x;

for (int &x : a)    // x передается по ссылке, поэтому
   x += 1;         // доступ есть на чтение и запись
```

Передача массива в функцию

В C/C++ массив всегда передается по ссылке, поэтому использование оператора sizeof() для определения его размера бесполезно.

При передаче массива в функцию следует явно передавать и его размер

```
const int n = 5;
int a[n] = {1, 3, 5, 6, 1};  // создали массив фиксированной длины 5
// …
void f(int a[], int len) {
 // создали функцию для работы с массивами любой заданной длины len
}

// …

f(a, n);                    // вызвали функцию для нашего массива длины 5
```

Функции могут изменять значения элементов массива.

Для предотвращения модификации исходного массива внутри тела функции можно в определении функции применять к параметру-массиву спецификатор типа **const**

Функции не должны модифицировать массивы без крайней необходимости (принцип наименьших привилегий):

int check_null(const int a[], int n)

Двумерные массивы

Двумерные массивы определяются как массив массивов.
Вот как будет выглядеть двумерный массив int a[3][4].

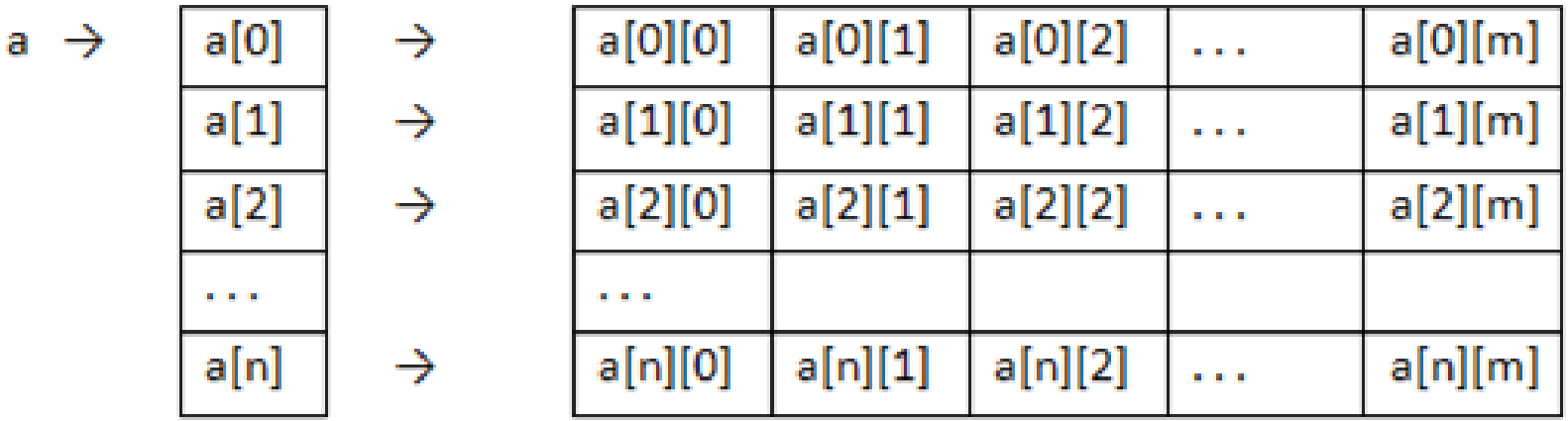
…	**a[ ][0]**	**a[ ][1]**	**a[ ][2]**	**a[ ][3]**
a[0][ ]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][ ]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][ ]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Передача в функции двумерных массивов

```
void print(int a[3][4], int m, int n) {
   for (int i = 0; i < n; i++)
       ...
}
```

При передаче массива в функцию сохранится размер массива **a[][4]**:
размер внешнего массива потеряется, а размер внутренних массивов сохранится.

Организация двумерного массива



Определение типов в C++

Синонимы (псевдонимы) типов определяются с помощью typedef

```
typedef unsigned char byte;

typedef int Arr[3];     // Arr имя типа
typedef int Matr[3][4]; // Matr имя типа

// Теперь можно использовать

void print(Matr a, int m, int n);
```

По сути объявление переменной Matr a будет заменено на int a[3][4]