Программирование на C++

Лекция 2

ПМИ 2 курс

Демяненко Я.М.

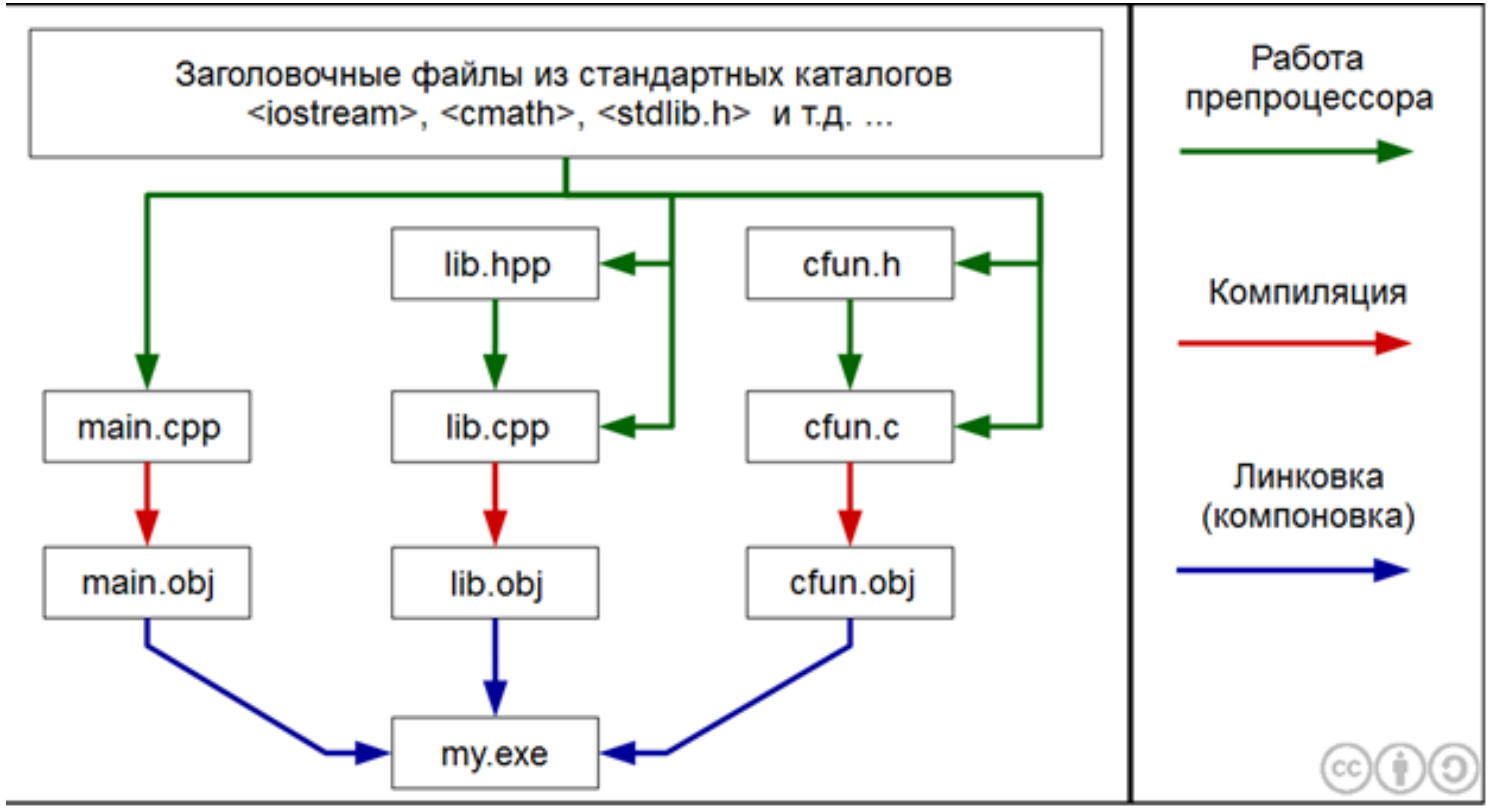
ЮФУ 2025

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] = { /* ... */ };

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"
```

Ошибки и их обработка

Причины ошибок времени выполнения

- некорректные данные
- некорректная работа с памятью
- некорректная работа с файлами

Они могут возникать не при каждом запуске программы, что затрудняет их поиск.

Если ошибка может произойти внутри функции

Если ошибка может произойти внутри функции, то необходимо сформулировать охраняющее условие.

При хороших данных функция должна вернуть результат или просто выполнить необходимые действия,

а в случае плохих — можно воспользоваться одним из приемов:

- аварийное завершение выполнения; exit(code) и abort
- выдача диагностики и завершение выполнения; assert()
- возврат признака ошибки (например, EOF).

Аварийное завершение выполнения

- exit(code) и abort(), выполняют выход из всей программы.
- Функция exit() позволяет вернуть код возврата как признак ошибки.
- **Различие между функциями exit и abort** состоит в том, что при использовании функции **exit** происходит обработка завершения среды выполнения C++ (вызываются глобальные деструкторы объектов), а при использовании функции **abort** программа завершается сразу.

Выдача диагностики и завершение выполнения

- assert()

```
int calc(int a, int e){
    assert(e,”division by zero”);
    return a/e;
}
```

- Функцию assert() удобно использовать для отладки программы.

- В окончательной версии отладочный код обычно отключается директивой #define NDEBUG

Функция с побочным эффектом

Так, например, функцию, вычисляющей целое частное двух целых чисел,

```
int calc (int a, int e) {
    return res=a/e;
}
```

заменяем следующей функцией:

```
bool calc (int a, int e, int &res) {
  if (e) {
    res=a/e;
    return true;
  }
  return false;
}
```

Использование функции с побочным эффектом

Один из вариантов вызова может выглядеть следующим образом:

```
if (calc(a,e,r))
   cout<<r;
else
   cout<<”ERROR”;
```

Синтаксис C++ позволяет использовать вызов функции, игнорируя возвращаемое значение.

```
calc(a,e,r);
cout<<r;
```

В этом случае ошибка, обнаруженная в функции, но не обработанная при вызове, может привести к непредсказуемым последствиям

Механизм обработки исключений

Благодаря механизму обработки исключений при возникновении ошибки, можно прервать выполнение и передать управление в другую часть программы вместе с информацией об ошибке.

- объект исключения (разных типов)
- генератор исключения
- обработчик исключений

Оператор генерации исключения

throw <исключение>;

```
int calc (int a, int e) {
  if (e)   {
    return a/e;
  }
  else
    throw ”Ошибка вычисления”;
}
```

Оператор **throw создает объект исключения** и может завершить выполнение функции, в которой возникла ошибка.
При этом **объект исключения** возвращается как **результат функции**, даже если тип этого объекта не соответствует типу функции.

Блок try

Чтобы предусмотреть обработку исключений, выполняемый код, в котором они могут возникнуть, помещается в блок try.

```
try {
// Программный код, который может генерировать исключения
}
```

Обработка исключений производится после блока try.
Это позволяет основному коду не смешиваться с кодом обработки ошибок.

Блок catch

Блок, где программа должна среагировать на сгенерированное исключение, называется **обработчиком исключения**.

Для каждого типа перехватываемого исключения должен быть свой обработчик.

Обработчики исключений следуют сразу же за блоком **try** и обозначаются ключевым словом **catch**

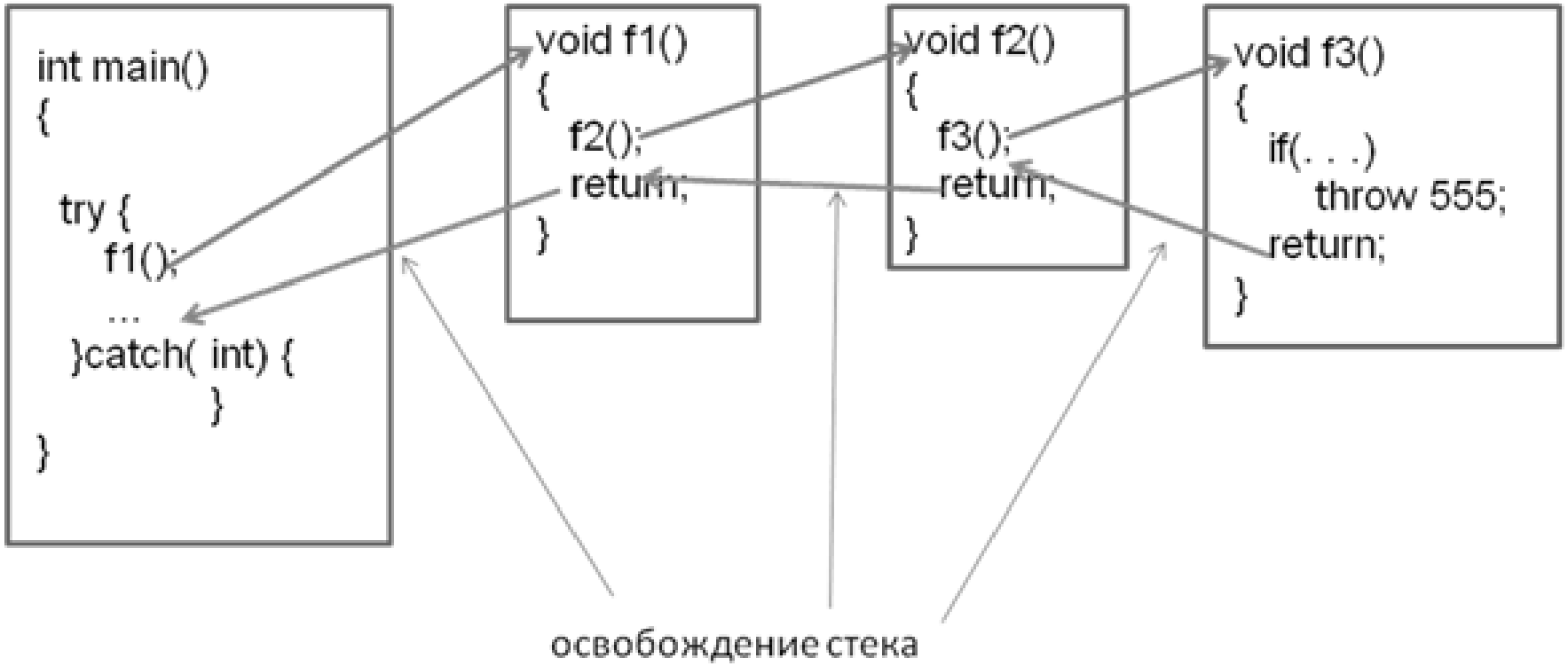
Блоки catch

```
catch (type1 id1){
// Обработка исключений типа type1
}
catch (type2 id2){
// Обработка исключений типа type2
}
…
catch (typeN idN){
// Обработка исключений типа typeN
}
//Здесь продолжится нормальное выполнение программы...
```

```
int calc (int a, int e) {
  if (e)   {
    return a/e;
  }
  else
    throw ”Ошибка вычисления”;
}
```

```
int main (){
  int a,e;
  cin>>a>>e;
  try {
    cout << calc(a,e);
  }
  catch (char *) {
    // что делать в случае ошибки?
  }
  return 0;
}
```

Нормальное завершение вызова функции f1



Завершение вызова функции f1 при возникновении исключительной ситуации