Семинар #1: Компиляция и линковка

В данном семинаре рассматриваются процессы компиляции и линковки, а также создание и использование статических и динамических библиотек. Процессы компиляции и линковки могут существенно различаться в зависимости от операционной системы. Кроме того, компиляция и линковка на языке C++ имеют свои особенности по сравнению с языком C. Поэтому в этом семинаре сначала будут рассмотрены основные этапы компиляции и линковки в операционной системе Linux на языке C. Затем будут подробно разобраны особенности этих процессов в языке C++. В завершение будут освещены ключевые отличия и специфика компиляции и линковки в операционной системе Windows.

Часть 1: Этапы компиляции

Компиляция – то процесс преобразования исходного кода, в машинный код, который может быть выполнен компьютером. Компилятор – это программа, которая осуществляет компиляцию.
В этом курсе мы использовали компилятор `gcc` для кода на языке C и компилятор `g++` для кода на C++.

Компиляция состоит из четырех основных этапов: препроцессинг, компиляция, ассемблирование и линковка. Если использовать компилятор `gcc`, просто передав ему имя файла командой:

```
$ gcc main.c
```

то все этапы будут выполнены автоматически, и на выходе получится готовый исполняемый файл. Однако при необходимости можно выполнить только отдельные этапы компиляции, чтобы более детально контролировать процесс. Например, можно остановиться после препроцессинга или после генерации объектного файла.

1. **Препроцессинг**
   - На этом этапе выполняется обработка директив препроцессора (`#include`, `#define`, `#ifdef` и т.д.)
   - Происходит включение заголовочных файлов, раскрытие макросов и удаление комментариев.
   - Выполнить только этот этап компиляции с помощью `gcc` можно следующим образом:

     ```
     $ gcc -E main.c -o main.i
     ```

2. **Компиляция** (не путать с компиляцией, состоящей из четырёх этапов)
   - Преобразованный на этапе препроцессинга код компилируется в ассемблерный код.
   - Результатом этого этапа является является файл с расширением `.s`.
   - Выполнить процесс компиляции до этого этапа включительно можно следующей командой:

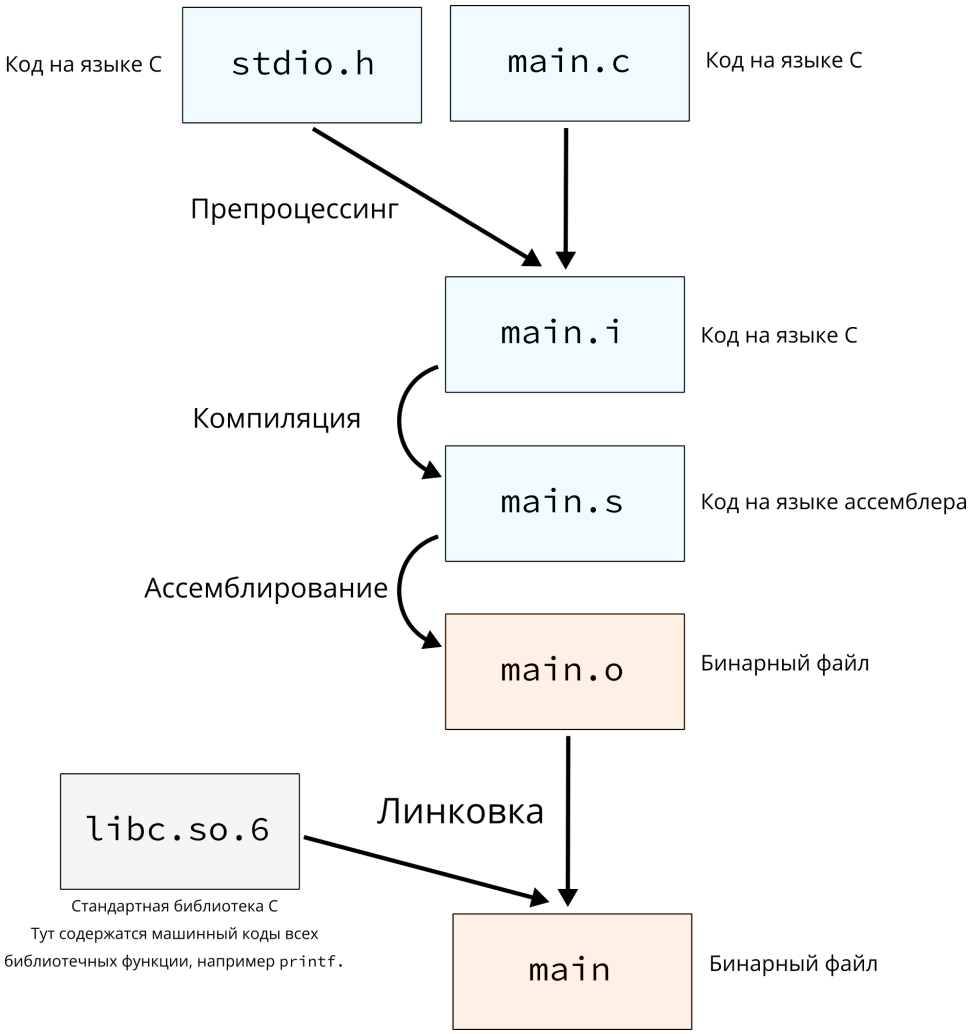
     ```
     $ gcc -S -masm=intel main.c
     ```

3. **Ассемблирования**
   - Ассемблерный код преобразуется в объектный код (машинный код), который представляет собой набор инструкций для процессора.
   - Результатом этого этапа является объектный файл с расширением `.o`.
   - Выполнить процесс компиляции до этого этапа включительно можно следующей командой:

     ```
     $ gcc -c main.c
     ```

4. **Линковка**
   - На этом этапе все объектные файлы и библиотеки связываются вместе.
   - Результатом этого этапа является исполняемый файл (на Linux исполняемые файлы обычно не имеют расширения, на Windows исполняемые файлы имеют расширение `.exe`) или библиотека.

Предположим, что наш проект состоит из одного файла `main.c` и из библиотек использует только стандартную библиотеку C и только функции из `stdio.h` (например, `printf`). Тогда процесс компиляции такого проекта можно показать на следующей схеме:



- На этапе препроцессинга компилятор видит директиву `#include <stdio.h>`. Поэтому он приступает к поиску заголовочного файла `stdio.h` в стандартных путях. Стандартные пути, в которых нужно искать заголовочные файлы хранятся внутри компилятора, их можно увидеть с помощью команды:

  ```
  $ echo | gcc -E -v -
  ```

- После того как он найдёт файл, он за место строки `#include <stdio.h>` вставит содержимое этого файла.
- Файл `stdio.h` содержит код на языке C и внутри этого файла также могут быть директивы `#include`, `#define` и другие. Компилятор будет выполнять все директивы пока их не останется.
- Внутри файла `stdio.h` НЕ содержатся коды стандартных функций. Там содержатся объявления (прототипы) функций. Это нужно, чтобы наша программа знала, что существуют те или иные функции.
- Код стандартных функций в уже скомпилированном виде (так называемый машинный код) содержится в библиотеке `libc.so.6` и подключается на этапе линковки.
- Различают статическую и динамическую линковку. При статической линковке машинный код всех используемых функций вставляется в окончательный исполняемый файл. При динамической линковке необходимые функции находятся в библиотеках во время выполнения.

Часть 2: Разделение кода программы на части с помощью `#include`

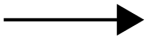
Простейшей способ разделение кода программы на несколько файлов – это использование директивы `#include`.

main.c

```c
#include <stdio.h>

int cube(int x)
{
    return x * x * x;
}

int main()
{
    printf("%i\n", cube(5));
}
```



main.c

```c
#include <stdio.h>
#include "cube.h"

int main()
{
    printf("%i\n", cube(5));
}
```

cube.h

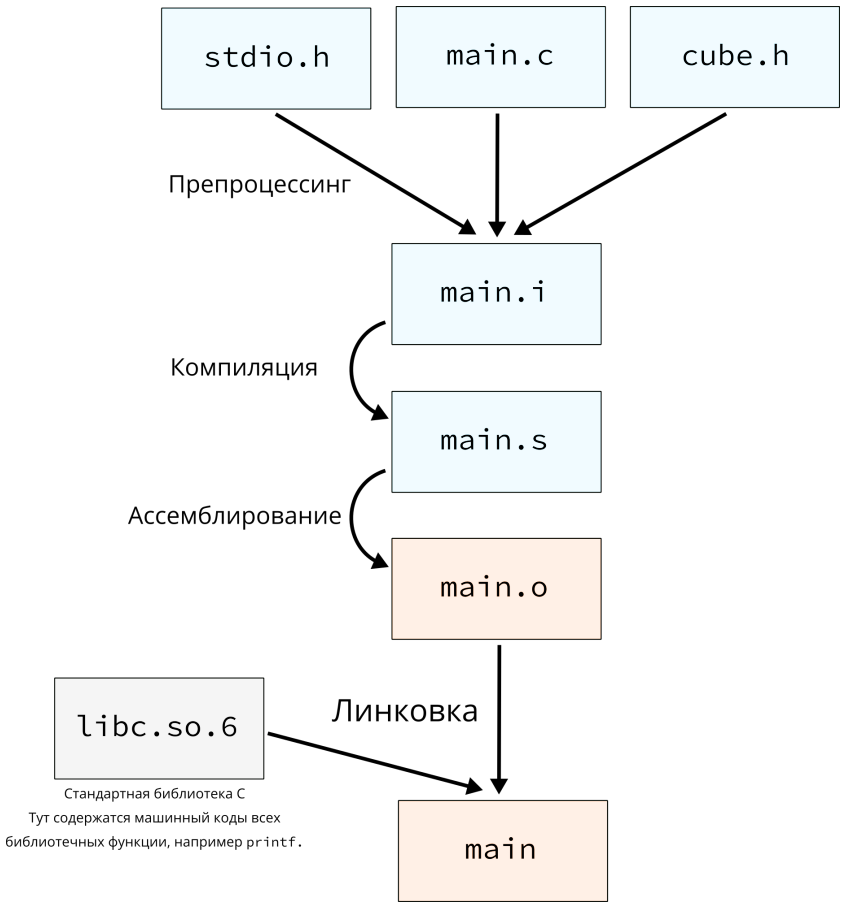
```c
#pragma once

int cube(int x)
{
    return x * x * x;
}
```

Файл `cube.h` просто вставится в файл `main.c` за место строки `#include "cube.h"` на этапе препроцессинга. При использовании угловых `<>` скобок в `#include` компилятор будет искать файл в стандартных путях, а при использовании кавычек `""` сначала поищет в текущей директории. Скомпилировать эту программу можно так:

```
$ gcc main.c
```

Передавать название файла `cube.h` компилятору не нужно, так как он найдёт этот файл через `#include`. Процесс компиляции этой программы почти не будет отличатся от предыдущей, что можно увидеть на схеме:

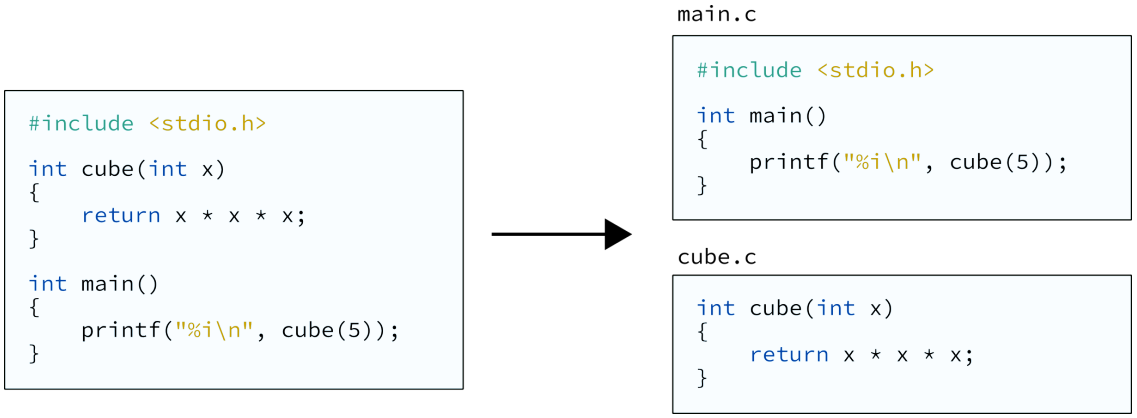


Часть 3: Раздельная компиляция

Раздельная компиляция – это подход, при котором программа разбивается на несколько отдельных файлов исходного кода (`.c`), которые компилируются независимо друг от друга. В отличии от использования `#include` при раздельной компиляции части программы будут компилироваться отдельно и объединяться вместе только на этапе линковки.

Ошибочный подход

Попробуем произвести раздельную компиляцию нашей программы, просто разделив один файл исходного кода на два: `main.c` и `cube.c` и передав на вход компилятору эти два файла.



Попробуем скомпилировать программу следующей командой:

```
$ gcc main.c cube.c
```

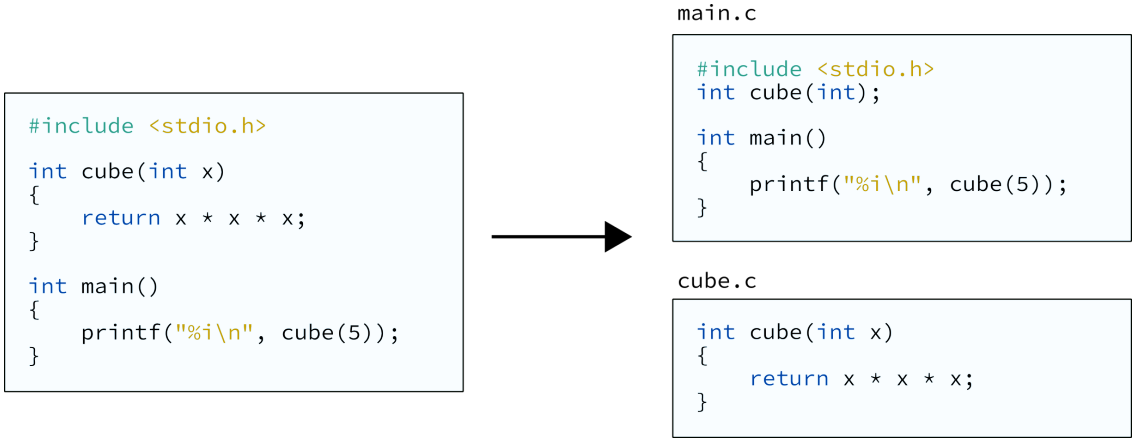
Произойдёт следующая ошибка:

```
main.c:5:20: error: implicit declaration of function 'cube'
```

Ошибка происходит из-за того, что компилятор пытается скомпилировать файлы `main.c` и `cube.c` раздельно, но не может скомпилировать файл `main.c` из-за того, что встечает неизвестную ему функцию `cube`. Компилятор не знает, что такая функция существует в другом файле, так как он должен скомпилировать два файла независимо друг от друга. Узнать, что в другом файле есть функция `cube` компилятор сможет только на этапе линковки.

Более правильный подход

Мы можем сказать компилятору, что функция `cube` существует в другом файле программы, просто объявив (написав прототип) эту функцию в файле `main.c`.

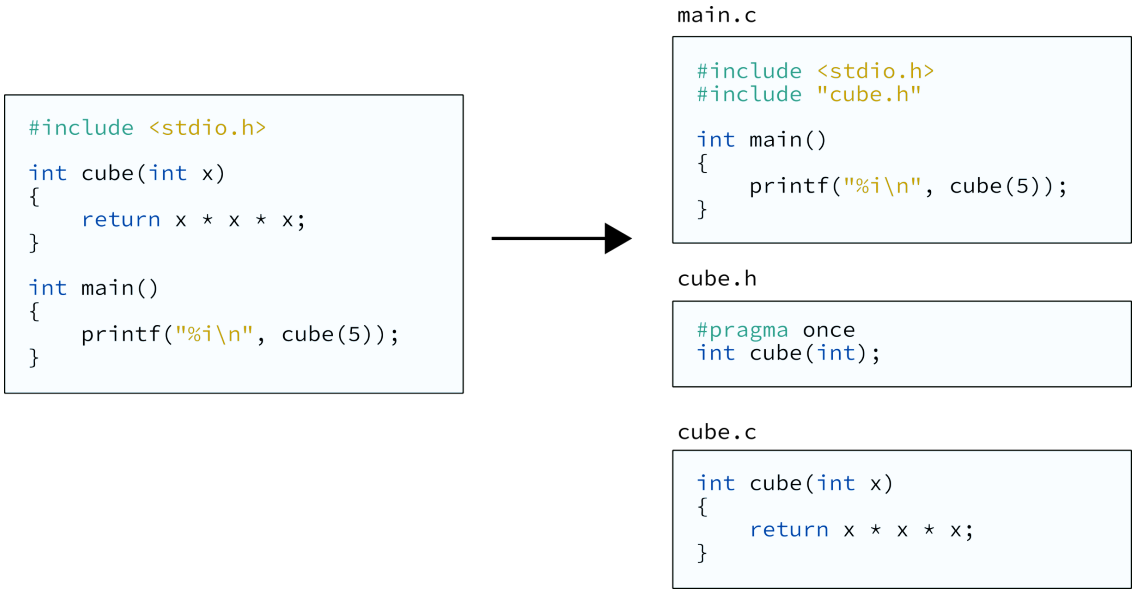


В этом случае, при компилировании файла `main.c` компилятор будет знать, что функция `cube` существует и компиляция пройдёт успешно.

Использование заголовочных файлов

Обычно, код делится на части таким образом, что в каждом файле исходного кода содержится множество функций, а не одна, как в нашем примере. Представьте, что за место файла `cube.c` с одной функцией, у нас был бы файл `mathfuncs.c`, который бы содержал 100 разных математических функций. Чтобы подключить такой файл к файлу `main.c` нам нужно было бы написать 100 прототипов функций в файле `main.c`. Если бы нам потребовалось подключить файл `mathfuncs.c` к ещё одному файлу, то нам бы потребовалось написать все 100 прототипов ещё раз. Чтобы решить эту проблему используются заголовочные файлы.

Заголовочный файл (англ. *header file*) – это текстовый файл с расширением `.h`, который содержит объявления функций, глобальных переменных, макросы, объявления структур и других типов данных, предназначенных для использования в нескольких файлах исходного кода (`.c`). Такие файлы вставляются в файлы исходного кода с помощью `#include`.



Теперь, если даже у нас будет файл, содержащий не одну функцию, а 100 функций, то мы просто напишем соответствующий заголовочный файл, содержащий 100 прототипов функций, и будем использовать его для подключения к другим файлам.
Скомпилировать данную программу можно командой:

```
$ gcc main.c cube.c
```

Указывать файл `cube.h` не нужно, так как компилятор найдёт его и вставит, используя директиву `#include`. При использовании этой команды компилятор выполнит все этапы компилирования и получит окончательный исполняемый файл. При этом компилятор будет компилировать файлы `main.c` и `cube.c` раздельно и объединять их только на этапе линковки.

Но можно произвести компиляцию поэтапно, воспользовавшись соответствующими опциями компилятора. Например, мы можем сами скомпилировать отдельно файл `main.c` в объектный файл `main.o`, а файл `cube.c` в объектный файл `cube.o`, а после этого слинковать их вместе в итоговый исполняемый файл.

```
$ gcc -c main.c      # компилируем файл main.c отдельно и получаем файл main.o
$ gcc -c cube.c      # компилируем файл cube.c отдельно и получаем файл cube.o
$ gcc main.o cube.o  # линкуем файлы main.o и cube.o и получаем исполняемый файл
```

Замечание об использовании заголовочных файлов

Как правило, файлы с расширением `.h` предназначения для того, чтобы содержать в себе только объявления (функций, структур, глобальных переменных) и макросы. Но несмотря на это, иногда в таких файлах содержится сам код функций (как мы это делали в Части 2). Это допустимо, но нужно понимать, что храня код функций в заголовочных файлах, мы лишаемся преимуществ, которые даёт раздельная компиляция.

Замечание о подключении заголовочного файла в соответствующий исходный файл

Заголовочный файл следует подключать во все файлы, в которых используются его объявления. Более того, важно подключать заголовочный файл его в соответствующий `.c`-файл. Например, `cube.h` должен быть включён в `cube.c`. Хотя компиляция может пройти и без этого, такое подключение обеспечивает контроль соответствия объявлений и определений. Если изменить объявление в `cube.h`, но забыть обновить `cube.c`, компилятор сразу обнаружит ошибку. Правильное разделение кода в нашем примере выглядит так:

```c
#include <stdio.h>

int cube(int x)
{
    return x * x * x;
}

int main()
{
    printf("%i\n", cube(5));
}
```

main.c

```c
#include <stdio.h>
#include "cube.h"

int main()
{
    printf("%i\n", cube(5));
}
```

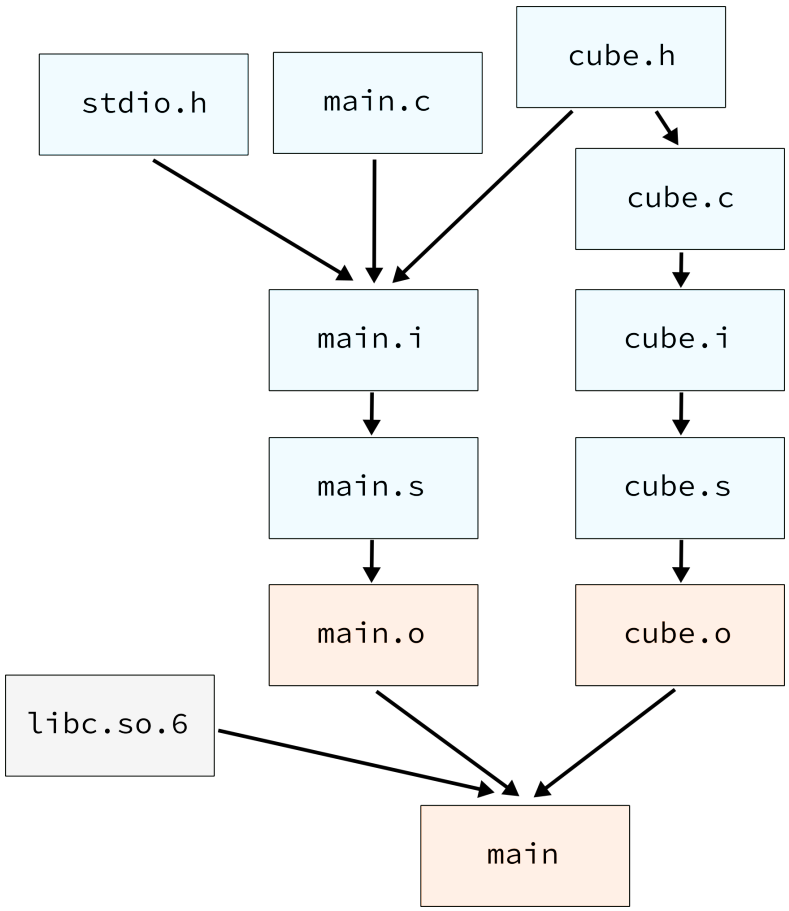
cube.h

```c
#pragma once
int cube(int);
```

cube.c

```c
#include "cube.h"
int cube(int x)
{
    return x * x * x;
}
```

Схема процесса раздельной компиляции нашей программы:



Преимущества раздельной компиляции

Само по себе разбиение программы на несколько файлов умеет преемущества, такие как:

- **Лучшая структура программы**
  Программу, разделённую на несколько файлов, можно сделать более структурированной. Код разбивается на части, каждая из которых отвечает за определённую функциональность.
- **Повторное использование кода**
  Код из одного проекта легче использовать в других проектах.

Однако, разбить код на части можно, используя только директиву `#include`, не прибегая к раздельной компиляции. Использование раздельной компиляции по сравнению с `#include` имеет следующие преимущества:

- **Ускорение перекомпиляции после изменений**
  Это самое главное преимущество раздельной компиляции. Чтобы его понять, представьте, что у вас есть большая программа, состоящая из сотен файлов. Предположим, что мы сделали маленькое изменение в одном из файлов и хотим заново скомпилировать программу. Если бы мы не использовали раздельную компиляцию (а просто бы подключали через `#include`), то нам бы пришлось перекомпилировать всю программу. Но, если мы используем раздельную компиляцию и сохранили объектные файлы со времени предыдущей компиляции, то нам будет достаточно перекомпилировать только файл, в котором были сделаны изменения. Этап линковки, всё же, придётся повторить для всей программы. Тем не менее, это всё равно будет намного быстрее компиляции всей программы.
- **Более точная настройка компиляции**
  Дальше в этом семинаре мы разберём, что компилировать программу можно разными способами, передавая компилятору разные опции. Использование раздельной компиляции позволяет использовать разные опции для разных файлов программы. Иногда это бывает полезно.

Ещё один пример программы, использующей раздельную компиляцию

Рассмотрим немного более сложную программу:

```c
#include <stdio.h>
#include <math.h>

struct point
{
    float x;
    float y;
};
typedef struct point Point;

float distance(Point a, Point b)
{
    Point c = {a.x - b.x, a.y - b.y};
    return sqrt(c.x * c.x + c.y * c.y);
}

float triangle_area(Point a, Point b, Point c)
{
    float sa = distance(b, c);
    float sb = distance(c, a);
    float sc = distance(a, b);
    float p = (sa + sb + sc) / 2.0f;
    return sqrt(p * (p - sa) * (p - sb) * (p - sc));
}

int main()
{
    Point a = {0, 0};
    Point b = {10, 20};
    Point c = {20, 5};
    printf("%f\n", triangle_area(a, b, c));
}
```

Разбить такую программу на несколько файлов, для последующего применения раздельной компиляции, можно следующим образом. Обратите внимание, что объявление структуры `Point` тоже помещается в заголовочный файл. Далее, мы включаем файл `point.h` во все файлы, где используется структура `Point`.

main.c

```c
#include <stdio.h>
#include "point.h"
#include "area.h"

int main()
{
    Point a = {0, 0};
    Point b = {10, 20};
    Point c = {20, 5};
    printf("%f\n", triangle_area(a, b, c));
}
```

area.h

```c
#pragma once
#include "point.h"
float triangle_area(Point, Point, Point);
```

area.c

```c
#include <math.h>
#include "point.h"

float triangle_area(Point a, Point b, Point c)
{
    float sa = distance(b, c);
    float sb = distance(c, a);
    float sc = distance(a, b);
    float p = (sa + sb + sc) / 2.0f;
    return sqrt(p * (p - sa) * (p - sb) * (p - sc));
}
```

point.h

```c
#pragma once
struct point
{
    float x;
    float y;
};
typedef struct point Point;

float distance(Point, Point);
```

point.c

```c
#include <math.h>
#include "point.h"

float distance(Point a, Point b)
{
    Point c = {a.x - b.x, a.y - b.y};
    return sqrt(c.x * c.x + c.y * c.y);
}
```

Часть 4: Опции компиляции

Опции компиляции – это параметры, которые передаются компилятору через командную строку для управления процессом компиляции. Они позволяют настраивать поведение компилятора, указывать пути к файлам, задавать уровни оптимизации, стандарты языка и многое другое.

Флаги компиляции – формально это булевые опции компиляции. Но часто понятия флаги компиляции и опции компиляции используются как синонимы.

Опция `-o`, задающая название выходного файла

Если скомпилировать программу, следующим образом:

```
$ gcc main.c
```

то навыходе получится исполняемый файл под названием `a.out`. Запустить этот файл можно будет с помощью:

```
$ ./a.out
```

Название итогового файла можно задать с помощью опции `-o`:

```
$ gcc main.c -o dog
```

В этом случае создастся файл с именем `dog`. Запустить этот файл можно будет с помощью:

```
$ ./dog
```

Опции для выполнения только первых шагов компиляции

- Опция `-E` – компилятор исполнит только этап препроцессинга.
- Опция `-S` – компилятор исполнит этапы препроцессинга и компиляции.
- Опция `-c` – компилятор исполнит этапы препроцессинга, компиляции и ассемблирования.

Опции для выбора стандарта языка

Язык C появился на свет в 1972 году и быстро стал популярным. В те времена было написано несколько компиляторов языка различными компаниями. Однако ранние компиляторы могли сильно различаться между собой. Можно сказать, что каждый компилятор воспринимал только свою версию языка, отличающуюся от тех с какими работали другие компиляторы. Это являлось большой проблемой, так как код, написанный для одного компилятора, мог не работать на другом. Для решения этой проблемы, в 1989 был представлен стандарт языка. Стандарт языка – это документ, который полностью описывает язык и его стандартную библиотеку. После этого разработчики компиляторов стали ориентироваться на стандарт и различий между разными версиями языка для различных компиляторов стало меньше.

Первая версия языка C (вышедшая в 1972) и даже первая стандартизированная версия языка (вышедшая в 1989) сильно отличались от современного языка C. Со временем в язык добавлялись новые возможности. Таким образом, язык C (как и большинство других языков программирования) не является постоянным, а меняется во времени. Поэтому переодически выходят новые стандарты языка. Так, новые стандарты вышли в 1999-м, 2011-м и 2023-м годах. Грубо говоря, стандарты являются версиями языка программирования. Код, написанный в соответствии с одним стандартом, может не компилироваться под другой версией стандарта.

Выбор стандарта языка C в компиляторе `gcc` осуществляется с помощью опции `-std=версия_стандарта`. Например, `-std=c89` означает, что нужно использовать стандарт 1989 года. Есть аналогичные опции `-std=c99`, `-std=c11` и `-std=c23` для стандартов 1999-го, 2011-го и 2023-го годов соответственно.

Аналогично, есть стандарты языка C++. Стандарты языка C++ выходили в 1998-м, 2003-м, 2011-м, 2014-м, 2017-и, 2020-м, 2023-м и 2026-м годах. Для языка C++ планируется выход нового стандарта каждые 3 года. Выбор стандарта языка C++ в компиляторе `g++` осуществляется с помощью следующих опций: `-std=c++98`, `-std=c++03`, `-std=c++11`, `-std=c++14`, `-std=c++17`, `-std=c++20`, `-std=c++23`, `-std=c++26`, ...

Опции для работы с предупреждениями (warnings)

Предупреждение (англ. *warning*) – это сообщение компилятора, которое указывает на потенциально проблемный код, который не нарушает синтаксических правил языка, но может привести к различным ошибкам во времени выполнения.

Формально, предупреждения не являются ошибками, поэтому программу, содержащую предупреждения, можно скомпилировать. Но к предупреждениям следует относится серьёзно, так как они могут указывать на ошибки в программе. Рассмотрим следующую программу, содержащую ошибку:

```
#include <stdio.h>
int main()
{
    for (int i = 0; i < 10; ++i);
        printf("Hello\n");
}
```

По умолчанию, данная программа скомпилируется без ошибок. Если её запустить, то на экране напечатается слово `Hello` один раз, что, скорей всего, не то, что мы хотели. Если же данную программу скомпилировать с опцией `-Wall`, то компилятор выдаст предупреждение – лишняя точка с запятой после цикла `for`. Проверки на большинство предупреждений не включены по умолчанию. Чтобы их включить используются опции:

- `-Wall` – включает некоторые предупреждения.
- `-Wextra` – включает дополнительные предупреждения.
- `-Wconversion` – предупреждает о неявных преобразованиях, которые могут привести к потере данных.
- `-Werror` – превращает все найденные предупреждения в ошибки.

Опции для оптимизации кода

Оптимизация кода - это изменение программы для увеличения её производительности без изменения её наблюдаемого поведения. Компилятор может оптимизировать ваш код, если ему передать необходимые опции. Рассмотрим, например, следующую программу:

```
#include <stdio.h>
int main()
{
    int a = 0;
    for (int i = 0; i < 2000000000; ++i)
        a += 1;
    printf("%i\n", a);
}
```

Эта программа прибавляет к числу, изначально равному нулю, единицу 2 миллиарда раз, а затем печатает это число. Очевидно, что эту программу можно очень сильно оптимизировать, если просто убрать цикл из программы и просто напечатать на экран число `2000000000`.
Если скомпилировать эту программу без опций для оптимизации, то получится программа, которая будет честно выполнять весь цикл и, соответственно, очень долго работать. Если же использовать опцию оптимизации:

```
$ gcc -O2 main.c
```

То комплятор оптимизирует программу. В результате программа не будет выполнять цикл, а просто напечатает на экран число `2000000000`. Такая программа будет работать очень быстро.

Компилятор может производить множество оптимизаций, например, вычисление констант на этапе компиляции, разворачивание циклов, для уменьшения накладных расходов на проверку условия, замена вызова функции на её тело (инлайнинг) и многие другие. Для включения оптимизаций используются следующие опции:

- `-O0` – отключает почти все оптимизации, используется по умолчанию.
- `-O1` – включает базовые оптимизации.

- `-O2` – включает большинство оптимизаций.
- `-O3` – включает все оптимизации, иногда может сильно увеличить размер исполняемого файла.
- `-Os` – пытается минимизировать размер исполняемого файла.

Опция -g для включения информации для дебаггера

Опции, для определения макросов (определений компиляции)

Рассмотрим следующую программу:

```
#include <stdio.h>

int main()
{
    #if defined(CAT)
        printf("Yes\n");
    #else
        printf("No\n");
    #endif
}
```

В данной программе директивы `#if`, `#else` и `#endif` используются для условной компиляции. В зависимости от того, определён ли макрос `CAT`, будет компилироваться та или иная строка.

- Если эту программу скомпилировать без опций:

  ```
  $ gcc main.c
  ```

  то при запуске напечатается строка `No`.

- Если же в начале программы определить макрос `CAT`:

  ```
  #include <stdio.h>
  #define CAT

  int main()
  {
      #if defined(CAT)
          printf("Yes\n");
      #else
          printf("No\n");
      #endif
  }
  ```

  скомпилировать и запустить программу, то напечатается `Yes`.

- Определить макрос в программе можно не используя директиву `#define` в коде программы, а используя опцию компиляции `-D` вот так:

  ```
  $ gcc -DCAT main.c
  ```

  В этом случае файл `main.c` будет компилироваться так, как будто в начале файла было написано `#define CAT`. Если после компиляции с опцией `-DCAT` запустить программу, то напечается `Yes`.

Опции, начинающиеся с `-D` также называются определениями компиляции (англ. *compile definitions*).

Опции, для определения макросов со значениями

Макросам можно задавать значения в коде программы:

```
#define CAT 10
```

В этом случае макрос будет считаться определённым (можно будет проверить это с помощью `defined`). Но, помимо этого, если макрос представляет собой целое число, то с ним в директивах `#if` и `#elif` можно производить операции сравнения, арифметические, логические и побитовые операции. Например, можно написать такой код:

```
#include <stdio.h>
int main()
{
    #if defined(CAT)
        #if CAT == 0
            printf("One\n");
        #elif CAT > 10 && CAT + 5 < 30
            printf("Two\n");
        #else
            printf("Three\n");
        #endif
    #endif
}
```

Макросы со значениями также можно задать через опцию компилятора `-D`. Посмотрим, что будет, если скомпилировать нашу программу со следующими опциями:

- Если скомпилировать без опций, то программа ничего на напечатает.
- Если скомпилировать с опцией `-DCAT=0`, то программа напечатает `One`.

  ```
  $ gcc -DCAT=0 main.c
  ```

- Если скомпилировать с опцией `-DCAT=20`, то программа напечатает `Two`.

  ```
  $ gcc -DCAT=20 main.c
  ```

Опция `-DNDEBUG`

Опции для работы с библиотеками

Есть несколько важных опций для подключения сторонних библиотек:

- `-I путь_к_папке` – добавление путей для поиска заголовочных файлов, подключаемых директивой `#include`.
- `-L путь_к_папке` – добавление путей для поиска библиотек (как статических, так и динамических).
- `-l имя_библиотеки` – указание файла библиотеки.

Далее эти опции будут разобраны более подробно.

Определения для нахождения ошибок

```
_FORTIFY_SOURCE
MALLOC_CHECK_
_GLIBCXX_ASSERTIONS
D_GLIBCXX_DEBUG
```

Часть 5: Статические библиотеки. Утилита `ar`.

Часть 6: Динамические библиотеки

Часть 7: Библиотека `dlfcn.h`

Часть 8: Статические и внешние символы

Единица трансляции

Объявление и определение

ODR

Ключевые слова `static` и `extern`

Просмотр символов объектных и исполняемых файлов

```
nm <имя файла>
nm -C <имя файла>
readelf -s <имя файла>
objdump -t <имя файла>
```

Часть 9: Ключевое слово `inline`

Часть 10: Особенности компиляции и линковки на языке C++

Манглирование имён

Утилита `c++filt`

`extern "C"`

Шаблоны

Статическая инициализация

`inline` и `constexpr`