

Компиляция и интерпретация программ

Языки программирования, в общем случае, разработаны для того, чтобы человек мог в понятной ему форме написать программу.

Для того, чтобы программа могла быть исполнена компьютером, её необходимо привести к “понятному” для “машины” виду.

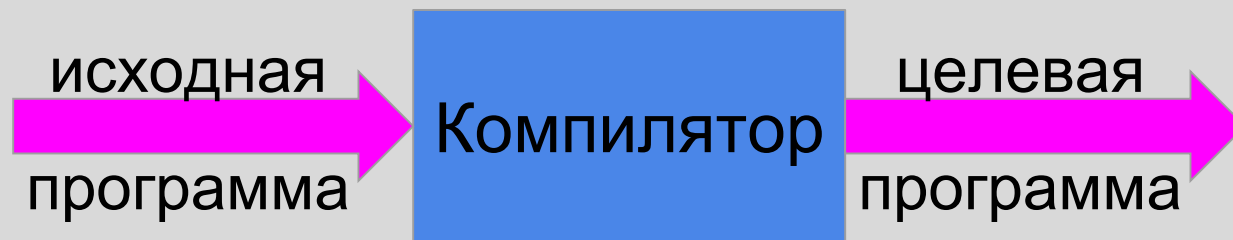
Существует два способа получения программы в пригодном для исполнения виде:

компиляция;

интерпретация.

Компиляция программы (1/2)

Компиляция - процесс преобразования программы (также говорят, трансляция), написанной на одном языке, в *полностью эквивалентную* программу на другом языке.



Компилятор - это компьютерная программа, выполняющая трансляцию других компьютерных программ.

Типичные *“исходные”* языки - C, C++, Java, многие другие.

“Целевые” языки - обычно, набор инструкций процессора или байт-код, исполняемый виртуальной машиной.

Компиляция программы (2/2)

Компиляторы также могут генерировать программу и на языке достаточно высокого уровня, например, на Си.

Компиляторами можно назвать много других систем.

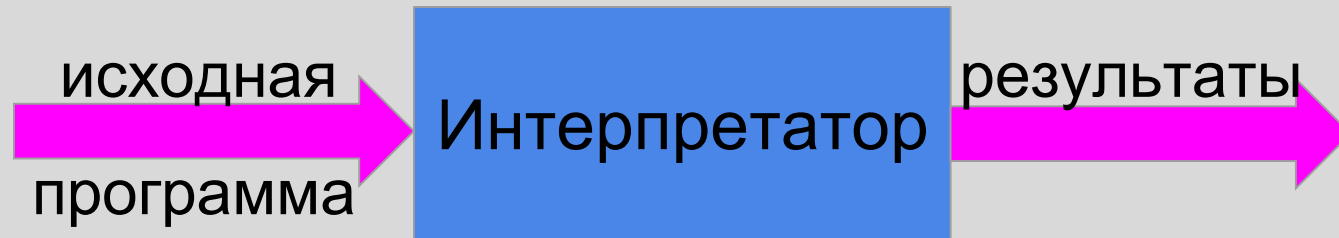
Например, PostScript - язык для описания страниц.

процесс получения PostScript-программы можно
назвать компиляцией;

в то же время, PostScript - интерпретируемый язык.

Интерпретация программы

Интерпретация программы - процесс её выполнения программой-интерпретатором последовательно, оператор за оператором.



Существуют гибридные схемы трансляции программ, основанные на совместном использовании компилятора и интерпретатора, например:



Интерпретация программы - примеры

Оболочки командной строки в различных операционных системах, такие как:

bash, ksh, cmd;

многие языки программирования - Python, php;

виртуализация - интерпретатор (по сути, и есть виртуальная машина) исполняет инструкции на промежуточном языке, полученные в результате компиляции. Это позволяет создавать переносимое ПО (например, языки, поддерживаемые Java Virtual Machine или Common Language Infrastructure).

Интерпретация или компиляция?

“За” интерпретацию:

интерпретатор легко реализовать (в сравнении с действительно хорошим компилятором);

код выполняется прямо “на лету” -

проще диагностировать ошибки;

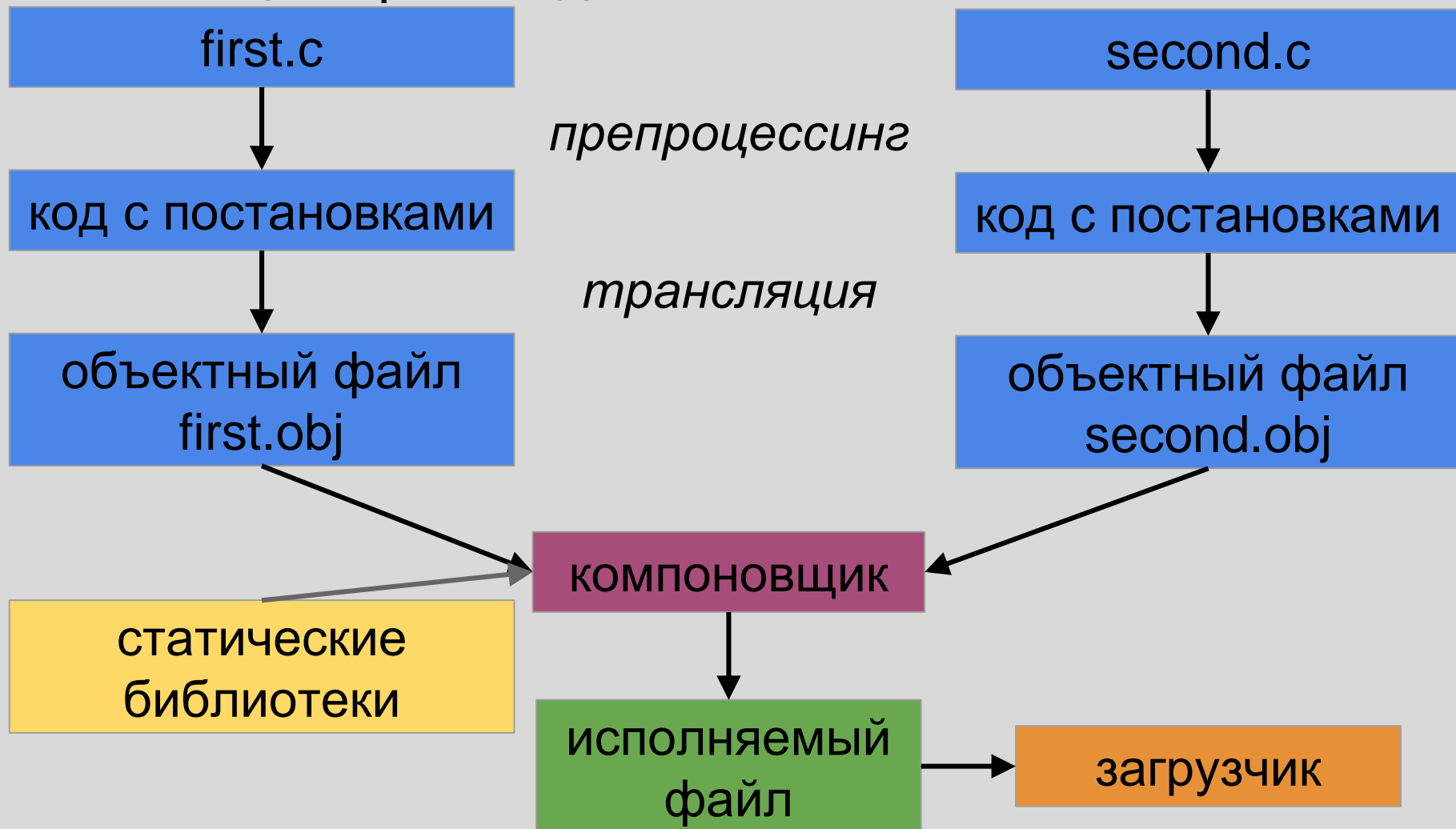
лучше подходит для языков с динамической типизацией;

“За” компиляцию:

компилятор может существенно оптимизировать программу;

Цикл компиляции

Компиляция происходит в несколько этапов.



Препроцессинг (1/4)

Текст программы - это набор символов. Первая стадия компиляции - препроцессинг исходного кода программы.

Эта стадия сводится к исполнению специальных команд для программы-препроцессора для обработки исходного текста программы.

Пример инструкции для препроцессора - директива **#define**, осуществляющая макроподстановку.

```
#define VECTOR_SIZE 42  
...  
int vec[VECTOR_SIZE];
```


Препроцессинг (2/4)

Процесс подстановки тела макроса вместо его имени называется инстанцированием.

После обработки исходного кода препроцессором, он примет такой вид:

```
#define VECTOR_SIZE 42  
...  
int vec[VECTOR_SIZE];
```



```
int vec[42];
```


Препроцессинг (3/4)

Препроцессор ничего не знает про синтаксис языка - про то, какие конструкции в нём разрешены.

Например, конструкция *if (condition) statement else statement* - цельная сущность с чётко определённой структурой.

Макросы позволяют оперировать её частями, но для успешной компиляции необходимо, чтобы эти части вместе составляли эту сущность.

```
#define VECTOR_SIZE 42+  
...  
int vec[VECTOR_SIZE 3]; //OK  
int a = VECTOR_SIZE; //Error
```



```
int vec[42+ 3]; //OK  
int a = 42+; //Error
```

Препроцессинг (4/4)

Препроцессор языка С - достаточно мощный инструмент. Он предоставляет следующие директивы, помимо уже упомянутой **#define**.

#undef

#include

#if, #ifdef, #ifndef, #elif, #else, #endif

#line

#error

#pragma

#

Трансляция программы (1/2)

Обычно компилятор принимает файл с исходным кодом и затем транслирует на другой язык.

Процесс трансляции состоит из множества внутренних стадий, на каждой из которых код может быть трансформирован из одного промежуточного представления (IR - intermediate representation) в другое.

Перед генерацией машинного кода IR близко к ассемблерному коду, и представление на языке ассемблера получить достаточно легко.

Точное же обратное представление (из ассемблерного кода в язык высокого уровня) получить невозможно.

Трансляция программы (2/2)

Модуль - единица трансляции исходного кода, обычно представленный как файл с исходным кодом, из которого генерируется **объектный файл**.

Объектный файл - файл, содержащий промежуточное представление программы (как правило, в виде машинного кода), из которого(-ых) может быть получен (скомпонован) исполняемый файл или библиотека.

Объектный файл не может исполняться - он может содержать обращения к коду, который содержится в других объектных файлах, но о них нет никакой информации - они компилируются ¹¹⁴раздельно.

Объектные файлы (1/2)

Релоцируемые объектные файлы. В комбинации с другими такими файлами компоновщик может сделать из него исполняемый объектный файл.

Релокация - это процесс привязывания адресов к различным частям программы и изменение программы таким образом, чтобы ссылки на эти части были правильными.

Компиляторы производят объектные файлы этого типа.

Объектные файлы (2/2)

Исполняемые объектные файлы. Содержат код и данные в таком виде, в котором они могут быть переданы загрузчику операционной системы для загрузки в память и выполнения.

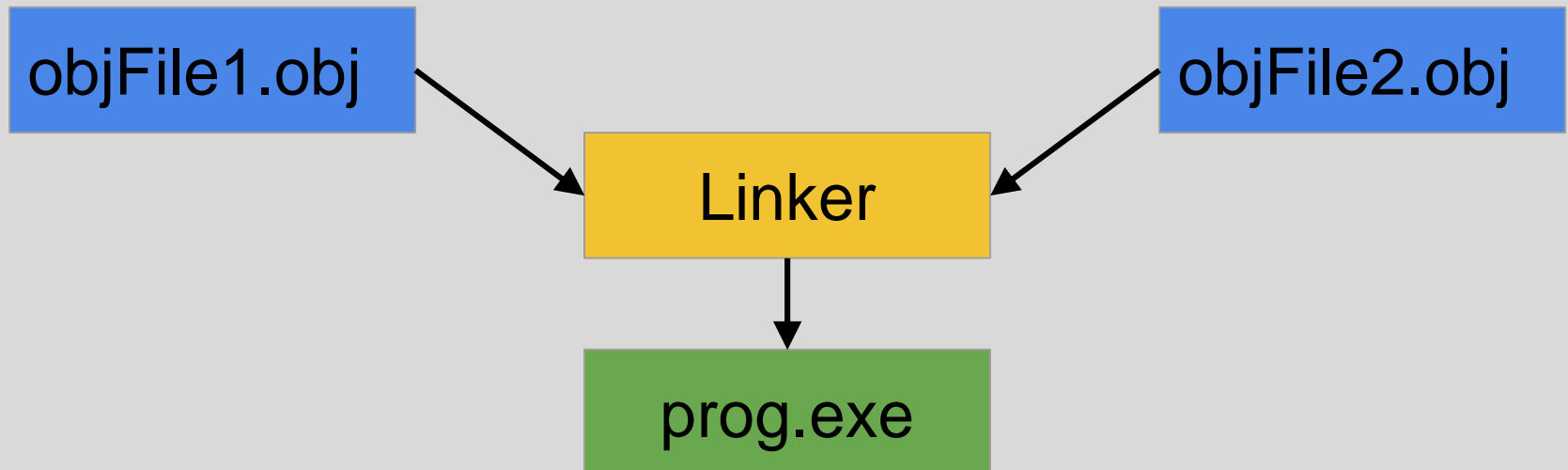
Файлы такого типа можно получить после компоновки.

Разделяемые объектные файлы. Специальный тип объектного файла, который можно загружать в память во время выполнения или загрузки основной программы и связывать с ней динамически (например, .dll в Windows, .so в *nix-системах).

Компоновка

Компоновка - процесс получения исполняемого файла из набора релоцируемых объектных файлов.

В процессе компоновки выполняется задача **разрешения символов (symbol resolution)** - в месте каждой ссылки на какой-либо символ устанавливается его правильный адрес.



Форматы объектных и исполняемых файлов

Существует больше количество форматов.

Наиболее распространённые - это

PE - Portable Executable, используемые в Windows;

ELF, COFF, используемые в *nix.

Форматы исполняемый файлов, тем не менее, имеют много общего - они содержат заголовки, инструкции, какие-либо другие данные.

Содержимое исполняемых и объектных файлов

Данные, необходимые для запуска процесса ОС

- а. описание окружения, в котором возможен запуск;
- б. данные, необходимые для запуска (например, размер кучи).

Отладочные данные.

Константы.

Список используемых функций, библиотек.

Различные ресурсы: тексты, изображения, звуки, видео, архивы, другие исполняемые данные;

Любые другие данные.

Загрузка исполняемых файлов

Загрузчик - служебная программа, часть операционной системы, размещающий в памяти образ исполняемого файла - секции (.text, .data, и др.), инициализирует .bss, выполняет отображение файлов с диска в память.

Статические библиотеки состоят из набора релоцируемых объектных файлов, которые связываются компоновщиком с основной программой и встраиваются в итоговый исполняемый файл.

Динамические библиотеки - те же самые разделяемые объектные файлы, но связываемые с программой во время её загрузки или выполнения.

Начало выполнения программы - точка входа

Точка входа - место в программе, с которого начинается её выполнение после загрузки в память (иначе говоря - происходит передача управления от операционной системы программе).

Понятие “точка входа” можно применить в двух случаях:

- 1.к исходному коду программы;
- 2.к исполняемому файлу.

Точка входа (взгляд на исходный код)

Выполнение программы может начинаться:

- с **функции `main`** (точное имя зависит от конкретного языка и настроек линковщика);

- с самого первого оператора, записанного в программе (это характерно для скриптовых языков, загрузчиков ОС, исполняемых файлов простейших форматов);

- с фиксированного адреса (абсолютного или относительного - смещения).

Обычно в программе может быть только одна точка входа, но существуют исключения: например, в ОС Android нет функции `main`, а выполнение начинается с загрузки и инициализации нескольких компонентов программы.

Точка входа (взгляд на исполняемый файл)

В распространённых форматах исполняемых файлов есть одна точка входа, которая указывается в определённом поле:

В ELF (*nix) это поле называется `e_entry`;

В формате PE (Windows) - это поле называется `AddressOfEntryPoint`.

В формате COM точка входа фиксирована - расположена по смещению `0100h`.

Точка входа в программах на С

Допустимы такие прототипы функции **main**, либо аналогичные:

```
int main(void);
```

```
int main(int argc, char *argv[]);
```

В качестве параметров в main могут передаваться аргументы командной строки и их количество:

- *argc* - *argument count* - количество аргументов;
- *argv* - *argument vector* - указатель на вектор строк-аргументов.

Именами могут быть любые идентификаторы, но принято использовать именно приведённые выше.

Параметры функции main

Значение `argc` всегда неотрицательное целое число.

`argv[argc]` - нулевой указатель.

если значение `argc > 0`, то:

`argv[0]` - имя процесса;

если имя процесса недоступно, то `argv[0][0]`
принимает значение нулевого символа.

`argv[1] ... argv[argc - 1]` - аргументы командной строки.

Код возврата (exit status) - 1/2

Если в качестве типа возвращаемого значения указан **int** (либо совместимый с ним), то число, которое вернёт функция **main** в вызвавшую её среду (например, родительскому процессу), говорит об *успешном или ошибочном завершении программы.*

В стандартной библиотеке C в виде макросов определены два таких значения:

EXIT_SUCCESS (*обычно определено как 0*) - означает успешное завершение программы.

EXIT_FAILURE - означает завершение программы в связи с ошибкой.

Код возврата (exit status) - 2/2

Возврат из функции **main** с определённым кодом аналогичен вызову библиотечной функции `exit` с этим же кодом.

В случае, если выполнение функции **main** завершилось, дойдя до конца блока функции (`}`), это равносильно возврату нуля (**return 0**), то есть успешному завершению программы.

Если в качестве типа возвращаемого значения функции **main** указан несовместимый с **int** тип, то значение, которое будет возвращено, не специфицируется.

Пример - вывод аргументов командной строки

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    for (i = 1; i < argc; i++)
        printf("Argument #%i is \"%s\".\n", i, argv[i]);

    return EXIT_SUCCESS;
}
```

Какие ошибочные ситуации не учтены в приведённой программе?

Использование макросов (1/4)

Для объявления констант

позволяет “спрятать” значение константы за именем, говорящем о назначении константы;

может привести к меньшему потреблению памяти, чем использование глобальных констант (**const**).

а также к тому, что программа будет работать немного быстрее за счёт отсутствия обращения к памяти

современные компиляторы могут выполнять такую: оптимизацию замену обращения к константе её значением (*constant folding*).

Использование макросов (2/4)

В С возможно объявление макросов с параметрами.

```
#define min(a, b) ((a > b) ? b : a)  
...  
int a = min(2, 5);
```

Также существует возможность приведения значения параметра к строковому представлению:

```
#define STR(x) #x  
...  
printf(STR(2)); //2  
printf(STR("2")); //"2"
```

Использование макросов (3/4)

Существует возможность подстановки и конкатенации токенов (оператор **##**):

```
#define TOKEN_PASTER_SAMPLE(num) \  
printf("var" #num " is %i", var##num);  
  
...  
int var3 = 2;  
TOKEN_PASTER_SAMPLE(3);  
//var3 is 2
```

Вместо **##<parameter_name>** будет подставлено имя параметра, переданное в макрос.

Обратите внимание на символ **'\'**, означающий, что макрос продолжается на следующей строке.

Использование макросов (4/4)

```
#define DEFINE_SCALAR(NAME, TYPE_NAME, VARIABLE, OP_ADD) \
typedef struct {
    TYPE_NAME VARIABLE;
} NAME;

NAME OP_ADD_##NAME(NAME arg1, NAME arg2) {
    NAME result;

    result.##VARIABLE = arg1.##VARIABLE + arg2.##VARIABLE;
    return result;
}

DEFINE_SCALAR(Temperature, double, value, arg1 + arg2)

void main() {
    Temperature t1 = { 10 }, t2 = { 20 };
    printf("%lf", OP_ADD_Temperature(t1, t2).value);
}
```

В результате
выполнения этой
программы будет
выведено:
30

Директивы препроцессора - #undef (1/2)

Директива **#undef** удаляет актуальное определение макроса.

```
#define VALUE 2

void printValue() {
    printf("%i", VALUE);
}

#undef VALUE

void main() {
    printValue();
    printf("%i", VALUE),
}
```

Ошибка компиляции. На момент прохода препроцессором этой строки, символ VALUE уже не определён.

Директивы препроцессора - #undef (2/2)

Директива **#undef** удаляет актуальное определение макроса.

```
#define VALUE 2

void printValue() {
    printf("%i", VALUE);
}

#undef VALUE
#define VALUE 3

void main() {
    printValue();
    printf("%i", VALUE);
}
```

Будет выведено:
23

Условная компиляция (1/2)

Существуют директивы, которые позволяют управлять компиляцией в зависимости от каких-либо

```
#ifdef NAME
```

```
/* Будут компилироваться эти строки, если объявлен  
символ NAME
```

```
*/
```

```
#endif
```

```
#ifndef NAME
```

```
/* Если символ NAME не объявлен, будут  
компилироваться эти строки */
```

```
#else
```

```
/* Будут компилироваться эти строки, если объявлен  
символ NAME */
```

```
#endif
```

Условная компиляция (2/2)

Директивы **#if**, **#elif** принимают в качестве аргумента константное выражение, результат вычисления которого - целое число.

```
#define FOO 0

void main() {
    #if defined(FOO)                //возможна также форма записи: defined
    FOO
        printf("FOO is defined. ");
    #if FOO < 0
        printf("FOO is negative.");
    #elif FOO == 0
        printf("FOO is zero.");
    #else
        printf("FOO is positive.");
    #endif                          //#if и #endif всегда идут в паре
}
```

Директивы препроцессора - #line,

- не оказывает никакого эффекта.

#line - устанавливает значение символа **__LINE__**, который предоставляет номер следующей за ним строки с исходным кодом*

```
#include <stdio.h>
#line 500
void main() {
    printf("%i", __LINE__); //Вывод: 501
}
```

изучите самостоятельно, какие макроимена, кроме **__LINE__, определены по стандарту.*

Директивы препроцессора - #error

#error - останавливает процесс компиляции с ошибкой в месте, где написана эта директива. После неё можно указать сообщение об ошибке.

```
#ifndef FOO
    #error Symbol 'FOO' is not defined
#endif
```

В этом примере компиляция будет прервана, в качестве сообщения об ошибке будет выведено “Symbol ‘FOO’ is not defined”.

Директивы препроцессора - **#include**

Вместо директивы **#include <filename>** вставляется содержимое этого файла.

Имя файла может быть заключено в скобки -

#include <stdio.h>

или в кавычки

#include “myHeaderFile.h”

Файл, заключённый в кавычки, ищется не только в директории компилятора, но и там же, где расположен файл с кодом, в котором написали эту директиву.

Заголовочные файлы (1/3)

Пусть в файле `functions.c` определены функции:

```
int sub();  
int add();  
double mul();
```

Допустим, необходимо сделать доступными эти функции в файле `main.c`, тогда в нём нужно написать:

```
int sub();  
int add();  
double mul();
```

Это приведёт к тому, что при необходимости использовать эти функции, нужно вручную писать их прототипы. Это не кажется хорошей идеей.

Заголовочные файлы (2/3)

Для решения такой проблемы придумали создавать заголовочные файлы (headers), содержащие объявления функций, типов, символов.

Создадим файл `functions.h`, где будут лишь объявления этих функций:

```
int sub();  
int add();  
double mul();
```

Для их использования останется лишь написать:

```
#include "functions.h"  
...
```


Заголовочные файлы (3/3)

При использовании заголовочных файлов, на место директивы **#include** препроцессор подставит содержимое этого файла, включая все необходимые объявления.

```
#include "functions.h"
```

```
int main() {  
    int a = add();  
}
```

Остаётся только пользоваться определёнными в другом модуле функциями: компоновщик определит их адреса и они будут вызваны. Так подключаются библиотеки. **Сама #include - не подключение библиотеки, а подстановка текста файла!**

Повторное включение заголовков (1/2)

Может возникнуть ситуация, когда одна и та же функция, макрос или тип данных определены в разных заголовочных файлах, и все они окажутся включены при помощи директивы **#include**.

Например, тип **size_t** определён в заголовочных файлах **stddef.h** и **stdlib.h**, и включение каждого из них могло бы привести к ошибке компиляции (но этого, конечно же, не происходит).

Повторное включение заголовков (2/2)

Другой пример: существуют заголовочные файлы -

```
//a.h  
#include <stdlib.h>  
  
size_t foo();
```

```
//b.h  
#include <stdlib.h>  
  
size_t bar();
```

И существует файл с кодом, code.c:

```
//code.c  
#include "a.h"  
#include "b.h"
```

Компилятор выдаст ошибку, сообщающую о повторном использовании типов данных и функций. Для решения этой проблемы существует Include guard.

Include guard

Каждый заголовочный файл оформляется так:

```
#ifndef _SOMETHING_H_  
    #define _SOMETHING_H_  
    //содержимое заголовочного файла  
#endif
```

При повторном включении заголовочного файла, оформленного таким образом, препроцессор обнаружит, что символ **_SOMETHING_H_** уже существует и повторно содержимое между директивами **#ifndef** и **#endif** включено не будет, а будет подставлена пустота.

Заголовочные файлы и библиотеки (1/2)

Как связаны между собой заголовочные файлы и библиотеки? - Никак!

Если подробнее: для того, чтобы использовать, например, функцию **printf**, необходимо подключить заголовочный файл **stdio.h**: **#include <stdio.h>**

Это не означает подключение библиотеки stdio.h!

Существует стандартная библиотека языка C (как и у многих других), в которой в виде машинного кода хранятся функции.

Заголовочные файлы и библиотеки (2/2)

При препроцессинге строки **#include <stdio.h>** подставляется содержимое заголовочного файла, помимо всего прочего, содержащего прототип функции **printf**.

В объектном файле, который создаёт компилятор, останутся неразрешёнными ссылки на функции, типы и переменные в других модулях (они записаны в **таблице импорта**).

Объявление функции без тела = запись в таблице импорта.

Объектные файлы также хранят **таблицы экспорта**, в которых указана информация о функциях, которые можно из этих файлов вызвать.

В процессе линковки будут установлены связи между объявлениями и определениями функций.

Ключевое слово `static`

У **`static`** есть несколько вариантов использования:

оно ограничивает область видимости глобальных переменных или функций модулем, в котором они определены (иными словами - такие функции и переменные не добавляются в таблицу экспорта).

локальные переменные, помеченные как **`static`**, создаются в области данных, а не в стеке, и значение такой переменной сохраняется при выходе из функции (а не удаляется, как в случае с обычной локальной переменной).

Ключевое слово static - пример

```
#include <stdio.h>

static void func() {
    static int a = 0;
    printf("%i", a++);
}

int main(void) {
    func();
    func();
    func();
}
```

Функция func не будет доступна из других модулей, а в результате будет выведено:
012

Строки в С (1/5)

Строка в С - просто массив символов (тип **char**), заканчивающийся символом с кодом 0 (**'\0'**).

Ноль выступает признаком конца строки, и такие строки называют нуль-терминированными.

Компилятор автоматически завершает строковые литералы символом **'\0'**.

Практически все строковые литералы в С помещаются в область, где хранятся константные глобальные данные.

Строки в C (2/5)

```
char *str = "this is a string.";
...
str[5] = 'q';
```

Попытка присвоить символ по индексу 5, скорее всего, завершится ошибкой в процессе выполнения (ошибка т.н. времени выполнения), потому что операционная система защищает область памяти, в которой хранятся константы.

Существует несколько способов создания изменяемых строк.

Строки в С (3/5)

1. Создать строку как глобальную переменную:

```
char str[] = "this is a mutable string.";
...
str[5] = 'q'; //всё будет ОК, строка будет изменена.
```

Это единственный случай, когда строковый литерал не будет сохранён в константной области данных.

Строки в C (4/5)

2. Создать строку как локальную переменную, в стеке:

```
void func() {  
    char str[] = "this is a mutable string too.";  
    ...  
    str[10] = '-'; //всё будет ОК, строка будет изменена.  
}
```

Сам строковый литерал будет сохранён в константной области данных, но при входе в функцию func будет выделено необходимое для строки количество байт, а содержимое строки - скопировано.

Строки в С (5/5)

3. Можно создать строку в динамической памяти:

```
#include <stdlib.h>
#include <string.h>
```

```
int main()
{
    char *str = (char*)malloc(sizeof(char) * 13);

    strcpy(str, "small string")

    free(str);
}
```

Обратите внимание, что строковый литерал состоит из 12 символов, а память выделена для 13-ти символов. Почему?

Интернирование строк

Интернирование строк означает, что константные строки создаются в области данных для хранения констант только один раз.

Даже если в программе строка встречается несколько раз, её копии создаваться не будут.

```
#include <stdio.h>

int main() {
    char *strA = "Simple string.";
    char *strB = "Simple string.";

    printf("%x\n%x", strA, strB);
}
```

В результате работы этой программы, будет дважды выведен один адрес, например:
0x816b30
0x816b30