

# Язык Си

## Параметры компилятора

### GCC

Позиционно независимый код ( **Position Independent Code** ).

Необходим для разделяемых библиотек

```
-fPIC
```

Параметры линковки, задает имя библиотеки **libraryname.so.1** и флаг разделяемой библиотеки

```
-shared -Wl,-soname,libraryname.so.1
```

Задает директорию, в нашем случае с переменной

**\$(DEFAULT\_LIB\_INSTALL\_PATH)** , где линковщик будет искать библиотеки для линковки

```
-Wl,-rpath,$(DEFAULT_LIB_INSTALL_PATH)
```

Регистрирует каталог с разделяемыми библиотеками в системе

```
ldconfig -n каталог
```

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:каталог
```

В свою очередь, ОС **Windows**, производит поиск зависимых разделяемых библиотек в следующей последовательности:

1. Производится поиск в текущей директории запуска исполняемого модуля
2. Производится поиск в директориях описанных в переменной окружения **PATH**
3. Производится поиск в системной директории **system32**

## Организация кода

1. Включение заголовочных файлов
2. Макроопределения, макроподстановки
3. Определение глобальных переменных
4. Описание прототипов функций
5. Определение функций.
6. Точка входа ( **main** ), если необходимо

## Типы данных

Язык Си является языком со статической типизацией, т.е. тип переменной известен на этапе компиляции и не меняется в процессе работы. Компилятор берет на себя проверку типов, снимая с программиста заботу о проверке типов(в отличии от динамической типизации где на этапе исполнения типы переменных могут не

совпадать и произойдет ошибка процесса исполнения). В современных языках используется прием вычисление типа на этапе компиляции. Если переменная обозначена специальным типом, то в процесс екомпиляции будет расичтан тип переменной и назначен ей. Такой прием применяется в новых стандартах языка `C++`, вычисляемым типом является `auto`.

Также, в языка Си применится слабая типизация, т.е. есть возможность привести один тип к другому, без проверки. Данный прием называется приведением типов. Эта операция считается опасной, и может привести к ошибкам на этапе исполнения программы, если типы не совпадают. Например при не совпадении типов указателей, при итерировании, можно выйти за границу массива и повредить память или при приведении типа `long long` к типу `int` можно потерять часть данных которые выходят за размер переменной. В современных компиляторах, преобразование типов, на стадии семантического анализа, производится попытка предположить возможность появления ошибки. Приведение типов записывается как `(int *)v` - что означает приведение переменной `v` к типу `int *` и называется `C-style` преобразование типов. Данный вид записи встречается во многих языках, нгапример в `Java`, `C#` - хотя в нем есть более удачные конструкции преобразования. Из-за приведения типов, язык не может считаться типобезопасным. В `C++`, такой стиль считается устаревшим и выдается предупреждение на этапе компиляции и предлагается использовать `static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast` - что является более безопасным и контролируемым компилятором.

Немного отступая, можно рассказать еще о так называемой “утиной типизации” - это такой вид типизации когда интерфейс явно не реализуется, а реализуются его методы. Данный подход применяется в языке `GoLang`.

Си также не грешит “каламбуром” типизации. Это когда обходится проверка системы типов компилятора, для выполнения определенных задач. Одним из примеров может быть интерфейс сокетов. Функция `bind` имеет следующее описание:

```
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
```

Вызов функции происходит примерно следующим образом

```
struct sockaddr_in sa = {0};  
int sockfd = ...;  
sa.sin_family = AF_INET;  
sa.sin_port = htons(port);  
bind(sockfd, (struct sockaddr *)&sa, sizeof sa);
```

Применяется за основу тот факт, что в языке указатель на `struct sockaddr_in` может беспрепятственно преобразовываться в указатель на `struct sockaddr`, а также что оба структурных типа частично совпадают по организации представления в памяти. Следовательно, указатель на поле `addr->sin_family` (где `addr` имеет тип `struct`

`sockaddr*` ) на самом деле будет указывать на поле `sa.sin_family` (где `sa` имеет тип `struct sockaddr_in`). Другими словами, библиотека использует каламбур типизации для реализации примитивной формы наследования.

Такой же подход можно встретить в `Windows API`.

Примеры типизации в других языках( `Asm` , `C++` , `Erlang` , `C#` , `Forth` )

Тип	Размер	Диапазон
char	1	
short	2	
int	4	
long	4	
float	4	
double	8	

## Квалификаторы

По стандарту `C11` , существует четыре квалификатора типа:

- `const (C89)` - означает что данный тип неизменяем после инициализации. (константа)
- `volatile (C89)` - означает что значение данной переменной часто подвержено изменениям.
- `restrict (C99)` - объявляемый указатель указывает на блок

памяти, на который не указывает никакой другой указатель

- `_Atomic (с C11)` . Также может именоваться `atomic` , если подключить `stdatomic.h` .

Также с 99го стандарта был добавлен квалификатор для функций `inline` , который является подсказкой компилятору, говорящей включить код из тела функции, вместо вызова самой функции.

Одной переменной могут принадлежать несколько квалификаторов.

```
const volatile int a = 5;
volatile int const * b = &a; //указатель на const volatile int
int * const c = 0;           // const указатель на int
```

## Классы хранения

Также в Си существует четыре класса хранения:

- `auto` - по-умолчанию для всех переменных.
- `register` - подсказка компилятору хранить переменные в регистрах процессора. Для таких переменных отсутствует операция взятия адреса
- `static` - статические переменные. Имеют область видимости файла.
- `extern` - переменные объявленные

## Макроподстановки.

# Макроопределения. Препроцессор.

Директивы:

- `define` - создание константы или макроса;
- `undef` - удаление константы или макроса;
- `include` - вставка содержимого указанного файла;
- `if` - проверка истинности выражения;
- `ifdef` - проверка существования константы или макроса;
- `ifndef` - проверка не существования константы или макроса;
- `else` - ветка условной компиляции при ложности выражения `if`;
- `elif` - проверка истинности другого выражения; краткая форма записи для комбинации `else` и `if`;
- `endif` - конец ветки условной компиляции;
- `line` - указание имени файла и номера текущей строки для компилятора;
- `error` - вывод сообщения и остановка компиляции;
- `warning` - вывод сообщения без остановки компиляции;
- `pragma` - указание действия, зависящего от реализации, для препроцессора или компилятора;
- если ключевое слово не указано, директива игнорируется;
- если указано несуществующее ключевое слово, выводится сообщение об ошибке и компиляция прерывается.

## Вставка файла

При обнаружении директив `#include "..."` и `#include <...>`, где `"..."` - имя файла, препроцессор читает содержимое указанного файла,

выполняет директивы и замены (подстановки), заменяет директиву `#include` на директиву `#line` и обработанное содержимое файла.

Для `#include "..."` поиск файла выполняется в текущей папке и папках, указанных в командной строке компилятора. Для `#include <...>` поиск файла выполняется в папках, содержащих файлы стандартной библиотеки (пути к этим папкам зависят от реализации компилятора).

При обнаружении директивы `#include` последовательность-лексем не совпадающей ни с одной из предыдущих форм, рассматривает последовательность лексем как текст, который в результате всех макроподстановок должен дать `#include <...>` или `#include "..."`. Сгенерированная таким образом директива далее будет интерпретироваться в соответствии с полученной формой.

Включаемые файлы обычно содержат:

- объявления функций;
- объявления глобальных переменных;
- определения интерфейсов;
- определения типов данных;

Директива `#include` обычно указывается в начале файла (в заголовке), поэтому включаемые файлы называются заголовочными.

Пример включения файлов из стандартной библиотеки языка C.



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

Использование препроцессора считается неэффективным по следующим причинам:

- каждый раз при включении файлов выполняются директивы и замены (подстановки); компилятор мог бы сохранять результаты препроцессирования для использования в будущем;
- множественные включения одного файла приходится предотвращать вручную с помощью директив условной компиляции; компилятор мог бы выполнять эту задачу самостоятельно.

## Константы и макросы

Константы и макросы препроцессора используются для определения небольших фрагментов кода.

```
// константа
#define BUFFER_SIZE ( 1024 )

// макрос
#define NUMBER_OF_ARRAY_ITEMS( array ) ( sizeof( array ) / si
sizeof( *(array) ) )
```

Каждая константа и каждый макрос заменяются соответствующим им

определением. Макросы имеют параметры, похожи на функции, используются для уменьшения накладных расходов при вызове функций в случаях, когда небольшого кода, вызываемого функцией, достаточно для ощутимого снижения производительности.

Пример. Определение макроса `max`, принимающего два аргумента: `a` и `b`.

```
#define max( a, b ) ( (a) > (b) ? (a) : (b) )
```

Макрос вызывается так же, как и любая функция.

```
z = max( x, y );
```

После замены макроса код будет выглядеть следующим образом:

```
z = ( (x) > (y) ? (x) : (y) );
```

Однако, наряду с преимуществами использования макросов в языке Си, например, для определения обобщённых типов данных или отладочных инструментов, они также несколько снижают эффективность их применения и даже могут привести к ошибкам.

Например, если `f` и `g` - две функции, вызов

```
z = max( f(), g() );
```

не вычислит один раз `f()` и один раз `g()`, и поместит наибольшее значение в `z`, как этого можно было ожидать. Вместо этого одна из функций будет вычислена дважды. Если функция имеет побочные эффекты, то вероятно, что её поведение будет отличаться от ожидаемого.

Макросы Си могут походить на функции, создавая новый синтаксис в некоторых пределах, а также могут быть дополнены произвольным текстом (хотя компилятор Си требует, чтобы текст был без ошибок написанным Си-кодом или оформлен как комментарий), но у них есть некоторые ограничения как у программных конструкций. Макросы, схожие с функциями, например, могут быть вызваны как «настоящие» функции, но макрос не может быть передан другой функции при помощи указателя, по той причине, что макрос сам по себе не имеет адреса.

Некоторые современные языки обычно не используют такой способ метапрограммирования с использованием макросов как дополнений строк символов, в расчете или на автоматическое или на ручное подключение функций и методов, а вместо этого другие способы абстракции, такие как шаблоны, обобщённые функции или параметрический полиморфизм. В частности, встраиваемые функции позволяют избежать одного из главных недостатков макросов в современных версиях Си и C++, так как встроенная функция обеспечивает преимущество макросов в снижении накладных расходов при вызове функции, но её адрес можно передавать в указателе для косвенных вызовов или использовать в качестве параметра.

Аналогично, проблема множественных вычислений, упомянутая выше в макросе `max`, для встроенных функций неактуальна.

Константы `#define` можно заменить на `enum`, а макросы - на функции `inline`.

## Условная компиляция

Препроцессор языка Си предоставляет возможность компиляции с условиями. Это допускает возможность существования различных версий одного кода. Обычно такой подход используется для настройки программы под платформу компилятора, состояние (отлаживаемый код может быть выделен в результирующем коде) или возможность проверки подключения файла строго один раз.

В общем случае, программисту необходимо использовать конструкцию типа:

```
#ifndef F00_H
#define F00_H

#endif
```

Такая “защита макросов” предотвращает двойное подключение заголовочного файла путём проверки существования этого макроса, который имеет то же самое имя, что и заголовочный файл.

Определение макроса `F00_H` происходит, когда заголовочный файл впервые обрабатывается препроцессором. Затем, если этот заголовочный файл вновь подключается, `F00_H` уже определен, в результате чего препроцессор пропускает полностью текст этого заголовочного файла.

То же самое можно сделать, включив в заголовочный файл директиву:

```
# pragma once
```

Условия препроцессора можно задавать несколькими способами, например:

```
# ifdef x
```

```
# else
```

```
# endif
```

```
# if x
```

```
# else
```

```
# endif
```

Этот способ часто используется в системных заголовочных файлах для

проверки различных возможностей, определение которых может меняться в зависимости от платформы; например, библиотека `Glibc` использует макросы с проверкой особенностей с целью проверить, что операционная система и оборудование их (макросы) корректно поддерживает при неизменности программного интерфейса.

Большинство современных языков программирования не используют такие возможности, больше полагаясь на традиционные операторы условия `if...then...else...`, оставляя компилятору задачу извлечения бесполезного кода из компилируемой программы.

## Предопределённые константы

Константы, создаваемые препроцессором автоматически:

- `__LINE__` заменяется на номер текущей строки; номер текущей строки может быть переопределен директивой `#line`; используется для отладки;
- `__FILE__` заменяется на имя файла; имя файла тоже может быть переопределено с помощью директивы `#line`;
- `__FUNCTION__` заменяется на имя текущей функции;
- `__DATE__` заменяется на текущую дату (на момент обработки кода препроцессором);
- `__TIME__` заменяется на текущее время (на момент обработки кода препроцессором);
- `__TIMESTAMP__` заменяется на текущие дату и время (на момент обработки кода препроцессором);
- `__COUNTER__` заменяется на уникальное число, начиная от 0;

после каждой замены число увеличивается на единицу;

- `__STDC__` заменяется на 1, если компиляция происходит в соответствии со стандартом языка C;
- `__STDC_HOSTED__` определена в C99 и выше; заменяется на 1, если выполнение происходит под управлением ОС;
- `__STDC_VERSION__` определена в C99 и выше; для C99 заменяется на число 199901, а для C11 - на число 201112;
- `__STDC_IEC_559__` определена в C99 и выше; константа существует, если компилятор поддерживает операции с числами с плавающей точкой по стандарту IEC 60559;
- `__STDC_IEC_559_COMPLEX__` определена в C99 и выше; константа существует, если компилятор поддерживает операции с комплексными числами по стандарту IEC 60559; стандарт C99 обязывает поддерживать операции с комплексными числами;
- `__STDC_NO_COMPLEX__` определена в C11; заменяется на 1, если не поддерживаются операции с комплексными числами;
- `__STDC_NO_VLA__` определена в C11; заменяется на 1, если не поддерживаются массивы переменной длины; в C99 массивы переменной длины обязательно должны поддерживаться;
- `__VA_ARGS__` определена в C99 и позволяет создавать макросы с переменным числом аргументов.

## Структуры. Перечисления.

## Объединения.

```
struct Person {
```

```
char *first_name;  
char *last_name;  
int age;  
};
```

```
enum Direct {  
    Left, Right, Unknown = -1  
};
```

```
union Match {  
    struct Binary {  
        unsigned char bin[4];  
    } binary;  
    unsigned long uni;  
};
```

## Строки

```
char *str = 0;
```

## Массивы

## Переменные

```
[модификаторы] тип имя [инициализация]
```



# Блок кода. Область видимости

```
{}
```

## Операции

Операция - это некоторая функция, которая выполняется над операндами и которая возвращает вычисленное значение - результат выполнения операции.

## Унарные операции

Унарные операции - это операции, содержащие единственный операнд.

К унарным операциям в Си относятся следующие операции:

1. **+** (унарный плюс),
2. **-** (унарный минус),
3. **~** (взятие обратного кода),
4. **!** (логическое отрицание),
5. **&** (взятие адреса),
6. **\*** (операция разыменовывания указателя),
7. **sizeof** (операция определения занимаемого объектом объёма памяти).

## Бинарные операции

Бинарные операции - это операции, содержащие два операнда, между которыми расположен знак операции.

К бинарным операциям в Си относятся следующие операции:

1. `+` (сложение),
2. `-` (вычитание),
3. `*` (умножение),
4. `/` (деление),
5. `%` (взятие остатка от деления),
6. `&` (поразрядное И),
7. `|` (поразрядное ИЛИ),
8. `^` (поразрядное исключающее ИЛИ),
9. `<<` (логический сдвиг влево),
10. `>>` (логический сдвиг вправо),
11. `&&` (логическое И),
12. `||` (логическое ИЛИ).

Также к бинарным операциям в Си относятся операции, по сути представляющие собою присваивание:

1. `+=` (добавление к левому операнду значения, представленного правым операндом);
2. `-=` (вычитание из левого операнда значения, представленного правым операндом);
3. `*=` (умножение левого операнда на значение, представленное правым операндом);
4. `/=` (деление левого операнда на значение, представленное правым операндом);
5. `&=` (поразрядное логическое И над левым и правым операндом);

6. `|=` (поразрядное логическое ИЛИ над левым и правым аргументом);
7. `^=` (поразрядное логическое исключающее ИЛИ над левым и правым аргументом);
8. `<<=` (поразрядный сдвиг влево левого аргумента на количество бит, заданное правым аргументом);
9. `>>=` (поразрядный сдвиг вправо левого аргумента на количество бит, заданное правым аргументом).

Данные операции предполагают, что левый операнд представляет собою лево-допустимое выражение.

## Тернарные операции

В Си имеется единственная тернарная операция - условная операция, которая имеет следующий вид:

`[условие]? [выражение1] : [выражение2] ;`

и которая имеет три операнда:

- `[условие]` - логическое условие, которое проверяется на истинность,
- `[выражение1]` - выражение, значение которого возвращается в качестве результата выполнения операции, если условие истинно;
- `[выражение2]` - выражение, значение которого возвращается в качестве результата выполнения операции, если условие ложно.

Знаком операции здесь служит целое сочетание `? :`.

# Функции и типы вызовов. Адрес возврата. Таблица функций.

Функция - это самостоятельный фрагмент программного кода, который может многократно использоваться в программе. Функции могут иметь аргументы и могут возвращать значения.

Для того, чтобы задать функцию в Си, необходимо её объявить:

- сообщить имя (идентификатор) функции,
- перечислить входные параметры (аргументы)
- указать тип возвращаемого значения,

Также необходимо привести определение функции, которое содержит блок операторов, реализующих поведение функции.

Отсутствие определения ранее определённой функции является ошибкой, что, в зависимости от реализации, приводит к выдаче сообщений или предупреждений.

Когда компилятор встречает в программном коде идентификатор функции, то он оформляет операцию вызова функции, в рамках которой, в частности, адрес точки вызова помещается в стек, создаются и инициализируются переменные, отвечающие за параметры функции, и передаётся управление коду, реализующему вызываемую функцию. После выполнения функции происходит освобождение памяти, выделенной при вызове функции, возврат в точку вызова и, если вызов функции является частью некоторого

выражения, передача в точку возврата вычисленного внутри функции значения.

Особый класс функций представляют встраиваемые (или подставляемые) функции - функции, объявленные с указанием ключевого слова `inline`. Определения таких функций непосредственно подставляются в точку вызова, что, с одной стороны, увеличивает объём исполняемого кода, но, с другой стороны, позволяет экономить время его выполнения, поскольку не используется дорогая по времени операция вызова функции.

## Объявление функции

Объявление функции имеет следующий формат:

```
[описатель] [имя] ( [список] );,
```

где

- `[описатель]` - описатель типа возвращаемого функцией значения;
- `[имя]` - имя функции (уникальный идентификатор функции);
- `[список]` - список (формальных) параметров функции.

Признаком объявления функции является символ `;`, таким образом, объявление функции - это инструкция.

В самом простом случае `[описатель]` содержит указание на конкретный тип возвращаемого значения. Функция, которая не

должна возвращать никакого значения, объявляется как имеющая тип `void`. При необходимости в описателе могут присутствовать дополнительные элементы:

- модификатор `extern` указывает на то, что определение функции находится в другом модуле;
- модификатор `static` задаёт статическую функцию;
- модификаторы `pascal` или `cdecl` влияют на обработку формальных параметров и связаны с подключением внешних модулей.

Список параметров функции задаёт сигнатуру функции.

Си не допускает объявление нескольких функций, имеющих одно и то же имя, перегрузка функций не поддерживается.

## Определение функции

Определение функции имеет следующий формат:

```
[описатель] [имя] ( [список] ) [тело]
```

Где `[описатель]`, `[имя]` и `[список]` - те же, что и в объявлении, а `[тело]` - это составной оператор, который представляет собою конкретную реализацию функции. Компилятор различает определения одноимённых функций по их сигнатуре, и таким образом (по сигнатуре) устанавливается связь между определением и соответствующим ему объявлением.

Тело функции имеет следующий вид:

```
{  
    [последовательность операторов]  
    return ([возвращаемое значение]) ;  
}
```

## Вызов функции

Вызов функции заключается в выполнении следующих действий:

- сохранение точки вызова в стеке;
- выделение памяти под переменные, соответствующие формальным параметрам функции;
- инициализация переменных значениями переменных (фактических параметров функции), переданных в функцию при её вызове, а также инициализация тех переменных, для которых в объявлении функции указаны значения по умолчанию, но для которых при вызове не были указаны соответствующие им фактические параметры;
- передача управления в тело функции.

В зависимости от реализации, компилятор либо строго следит за тем, чтобы тип фактического параметра совпадал с типом формального параметра, либо, если существует такая возможность, осуществляет неявное преобразование типа, что, очевидно, приводит к побочным эффектам.

Если в функцию передаётся переменная, то при вызове функции создаётся её копия (в стеке выделяется память и копируется значение). Например, передача структуры в функцию вызовет копирование всей структуры целиком. Если же передаётся указатель на структуру, то копируется только значение указателя. Передача в функцию массива также вызывает лишь копирование указателя на его первый элемент. При этом для явного обозначения того, что на вход функции принимается адрес начала массива, а не указатель на единичную переменную, вместо объявления указателя после названия переменной можно поставить квадратные скобки, например:

```
void example_func(int *array);
```

Си не допускает вложенные вызовы.

Частный случай вложенного вызова - это вызов функции внутри тела вызываемой функции. Такой вызов называется рекурсивным, и применяется для организации единообразных вычислений. Учитывая естественное ограничение на вложенные вызовы, рекурсивную реализацию заменяют на реализацию при помощи циклов.

## Возврат из функции

При возврате из функции освобождается память, выделенная под параметры функции и под переменные, объявленные внутри функции, и управление возвращается в точку вызова.



# Соглашения о вызовах

## `cdecl`

Используется компиляторами для языка Си.

Аргументы функций передаются через стек, справа налево.

Аргументы, размер которых меньше 4-х байт, расширяются до 4-х байт.

Очистку стека производит вызывающая программа. Это основной способ вызова функций с переменным числом аргументов (например, `printf()` ).

Перед вызовом функции вставляется код, называемый прологом и выполняющий следующие действия:

- сохранение значений регистров, используемых внутри функции;
- запись в стек аргументов функции.

После вызова функции вставляется код, называемый эпилогом и выполняющий следующие действия:

- восстановление значений регистров, сохранённых кодом пролога;
- очистка стека (от локальных переменных функции).

## `stdcall`

Используется в ОС `Windows` для вызова функций `WinAPI` .

Аргументы функций передаются через стек, справа налево. Очистку стека производит вызываемая подпрограмма.

## fastcall

Общее название соглашений, передающих параметры через регистры (обычно это самый быстрый способ, отсюда название). Если для сохранения всех параметров и промежуточных результатов регистров не достаточно, используется стек.

Соглашение о вызовах `fastcall` не стандартизировано, поэтому используется только для вызова процедур и функций, не экспортируемых из исполняемого модуля и не импортируемых извне.

## x64

Для архитектуры `x64`, в отличие от `x86`, параметры передаются так, первые четыре параметра через регистры `RCX`, `RDX`, `R8`, `R9`, остальные через стек.

С плавающей запятой и двойной точности аргументы передаются в `XMM0`, `XMM1`, `XMM2`, `XMM3`.

`_m128` строки, массивы и типы никогда не передаются по значению, вместо этого передается указатель на память.

Пример: переменная `a` в `RCX`, переменная `b` в `RDX`, переменная `c` в `R8`, переменная `d` в `R9`, переменная `e` помещается в стек

```
func1(int a, int b, int c, int d, int e);
```

Пример: переменная **a** в **XMM0**, переменная **b** в **XMM1**, переменная **c** в **XMM2**, переменная **d** в **XMM3**, переменная **e** помещается в стек

```
func2(float a, double b, float c, double d, float e);
```

Пример: переменная **a** в **RCX**, переменная **b** в **XMM1**, переменная **c** в **R8**, переменная **d** в **XMM3**

```
func3(int a, double b, int c, float d);
```

Пример: переменная **a** в **RCX**, указатель на **b** в **RDX**, указатель на **c** в **R8**, переменная **d** в **XMM3**

```
func4(__m64 a, _m128 b, struct c, float d);
```

## Возвращаемые значения

Пример: переменная **a** в **RCX**, переменная **b** в **RDX**, переменная **c** в **R8**, переменная **d** в **R9**, переменная **e** помещается в стек. Результат: **\_\_int64** помещен в **EAX**

```
__int64 func1(int a, float b, int c, int d, int e);
```

Пример: переменная **a** в **XMM0**, переменная **b** в **XMM1**, переменная **c**

в **R8** , переменная **d** в **R9** . Результат: **\_\_m128** помещен в **XMM0**

```
__m128 func2(float a, double b, int c, __m64 d);
```

Возвращение результат по значению.

Структура: размер превышает 64 бита

Пример: переменная **a** в **RDX** , переменная **b** в **XMM2** , переменная **c** в **R9** , переменная **d** помещена в стек. Результат: аллоцирован в **RCX** , указатель передан в **RAX**

```
struct Struct1 {  
    int j, k, l;  
};  
Struct1 func3(int a, double b, int c, float d);
```

Возвращение результат по значению.

Структура: размер не превышает 64 бита

Пример: переменная **a** в **RCX** , переменная **b** в **XMM1** , переменная **c** в **R8** , переменная **d** в **XMM3** . Результат: помещен в **RAX**

```
struct Struct2 {  
    int j, k;  
};  
Struct2 func4(int a, double b, int c, float d);
```

## Конструкции языка

# Получение текущего адреса вызова

Получить текущий адрес в исполняемом модуле можно выполнив

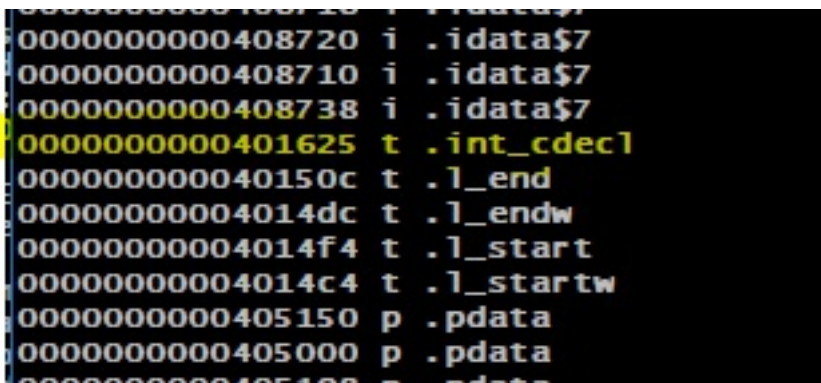
`call` по метке, в стеке будет адрес возврата

```
void *c_address = 0;
asm("call .int_cdecl;\n\t"
    ".int_cdecl: \n\t"
    "pop %%rax;\n\t"
    "movq %%rax, %0;" : "=r"(c_address));
fprintf(stdout, "Address: 0x%p\n", c_address);
```

Собрав пример, посмотрим адреса символов в исполняемом модуле вызвав команду

```
nm 04.Language_function_main.exe
```

Пример результата можно посмотреть на изображении



```
0000000000408720 i .idata$7
0000000000408710 i .idata$7
0000000000408738 i .idata$7
0000000000401625 t .int_cdecl
000000000040150c t .l_end
00000000004014dc t .l_endw
00000000004014f4 t .l_start
00000000004014c4 t .l_startw
0000000000405150 p .pdata
0000000000405000 p .pdata
0000000000405198 p .pdata
```

После вызова на исполнение, адрес будет выведен в терминал

Address : 0x0000000000401625

# Процесс компиляции

Создадим файл `m1.c` с кодом:

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

void m1() {
    fprintf(stdout, "M1\n");
}
```

Создадим файл `mm.c` с кодом:

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

void m1();

int main() {
    m1();
    return 0;
}
```

Скомпилируем файлы по отдельности:

```
gcc -c m1.c
```

```
gcc -c mm.c
```

В результате получим два файла с объектным кодом `m1.o` и `mm.o`.

Далее линкуем оба файла и получаем исполняемый модуль/

```
gcc m1.o mm.o -o mm
```