КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 12 Подпрограммы

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2022

2022.04.15

Оглавление

Подпрограммы (subroutines)

Предназначены для неоднократного выполнения какой-нибудь определенной последовательности действий:

```
printf("%06i:\t%s\n", key, value_string);
fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC);
```

Подпрограммы бывают двух видов — процедуры и функции.

Процедуры (чистые) изменяют состояние [информационных объектов] программы (методы).

Функции (чистые) возвращают информационные объекты, оставляя состояние программы без изменения.

```
exit(exit_code);          // чистая процедура
ldiv_t res = ldiv(p, q);  // чистая функция
```

Поведение подпрограмм может контролироваться со стороны вызывающего кода с помощью передачи в подпрограмму параметров в момент их вызова.

Область видимости

Область видимости (область действия) — часть программы, в пределах которой *идентификатор*, объявленный как имя некоторой программной сущности (переменной, типа данных, функции ...), остаётся связанным с этой сущностью, то есть позволяет посредством себя обратиться к ней.

Говорят, что идентификатор объекта «виден» в определённом месте программы, если в данном месте, используя идентификатор, можно обратиться к данному объекту.

За пределами области видимости тот же самый идентификатор может быть связан с другой переменной или функцией, либо быть свободным (не связанным ни с какой из них).

Область видимости может, но не обязана совпадать с областью существования объекта, с которым связано имя.

Глобальная область видимости

В этом случае идентификатор доступен во всем тексте программы.

Для этой цели используется пара директив **GLOBAL** и **EXTERN** — экспорт и импорт символов, соответственно, из других модулей.

EXTERN аналогична ключевому слову C **extern** — она используется для объявления символа, который не определен где-либо в текущем компилируемом модуле, но предполагается, что он определен в каком-то другом модуле, и на него необходимо ссылаться.

Директива **EXTERN** принимает произвольное количество аргументов. Каждый аргумент — это имя символа:

```
extern _printf
extern _sscanf, _fscanf
```

Некоторые форматы объектных файлов предоставляют дополнительные функции для директивы **EXTERN**.

Во всех случаях дополнительные функции используются путем добавления двоеточия к имени символа, за которым следует текст, специфичный для объектного формата (данные или метки).

В любом случае, дополнительный синтаксис отделяется от имени символа двоеточием. Например, формат **obj** при помощи следующей директивы позволяет объявить, что базой сегмента внешних символов по умолчанию должна быть группа **dgroup**:

```
  extern _variable:wrt dgroup
```

GLOBAL — это другая сторона **EXTERN** — если один модуль объявляет символ как **EXTERN** и ссылается на него, то какой-то другой модуль должен фактически определить этот символ и объявить его как **GLOBAL**. В противном случае получим ошибку компоновки.

GLOBAL использует тот же синтаксис, что и **EXTERN**, за исключением того, что он должен ссылаться на символы, которые определены в том же модуле, что и директива **GLOBAL**. Например:

```
global _main
_main:
        ; some code
```

И **GLOBAL** и **EXTERN** позволяют определять специфические для конкретного формата объектного модуля расширения (спецификаторы) с помощью двоеточия.

Например, формат объекта elf позволяет указать, являются ли глобальные элементы данных функциями или данными, например:

```
global  hashlookup:function, hashtable:data
```

Если переменная объявлена как **GLOBAL** и **EXTERN**, или если она объявлена как **EXTERN**, а затем определена, она будет обрабатываться как **GLOBAL**.

После ввода спецификатора можно также указать в виде числового выражения (которое может включать метки и даже опережающие ссылки) размер ассоциированных с символом данных:

```
global hashtable:data (hashtable.end - hashtable)
hashtable:
          db this,that,theother  ; здесь некоторые данные
.end:
```

Это заставит NASM автоматически подсчитать длину таблицы и поместить эту информацию в символьную таблицу ELF.

Область видимости в пределах модуля

В этом случае идентификатор доступен только в определенном модуле программы.

`STATIC` — Определение локальных символов в модулях

В отличие от **`EXTERN`** и **`GLOBAL`**, директива **`STATIC`** определяет локальный символ, который именуется в соответствии с глобальными правилами преобразования имен (mangling). Она аналогична ключевому слову языка C **`static`** применительно к функциям или глобальным переменным).

```
static foo
foo:
        ; codes
```

Локальная область видимости

Имя доступно только внутри определенной функции/процедуры, например, как это реализуется в пределах локального блока в языке C.

Пакет/Пространство имен

В этом случае в глобальной области видимости выделяется поименованная подобласть, имя «привязывается» к этой части программы и существует только внутри нее.

Вне данной области имя недоступно.

Передача параметров в процедуру

Передача параметров по ссылке и по значению

Параметры бывают формальные — при объявлении/определении подпрограммы, и фактические — при ее вызове.

Количество параметров может быть фиксировано, либо может быть переменным.

```
int printf(const char *format, ...);
```

Существует два способа передачи параметров — про ссылке и по значению.

Передача по ссылке подразумевает передачу в подпрограмму ссылки на объект (адреса, указателя).

Подпрограмма в этом случае может изменить состояние объекта в процессе своей работы.

Передача по значению — в подпрограмму передается копия объекта, изменение которой никак не сказывается на работе программы, поскольку сам объект недоступен внутри подпрограммы.

В различных языках используются различные способы передачи параметров.

Фортран — по ссылке, C — по значению, C++ — оба.

Передача параметров через регистры

Перед вызовом подпрограммы необходимые параметры помещаются в регистры.

```
lea     esi, [array]    ; загружаем адрес массива в регистр esi
mov     ecx, array_sz   ; загружаем размер массива в регистр ecx
call    array_proc      ; вызываем процедуру
```

Данный метод является самым быстрым, но есть существенный недостаток — количество регистров ограничено.

В x86_64 количество РОН увеличено вдвое (добавлены R08 – R15), что позволяет чаще пользоваться данным методом.

a32	Может использоваться в вызывающей подпрограмме для	+x86_64*
EAX	возврата значения из функции, + может быть испорчен неявно	R08
EBX	может быть испорчен неявно инструкцией XLAT	R09
ECX	в качестве счетчика loop, rep, …, + может быть испорчен неявно	R10
EDX	возврата значения из функции, + может быть испорчен неявно	R11
ESI	индексного регистра источника	R12
EDI	индексного регистра приемника	R13
ESP	Указатель стека	R14
EBP	Указатель фрейма	R15

Передача параметров через общую область памяти

Вызывающая и вызываемая процедура «договариваются» о совместном использовании некоторой области памяти, которая должна быть доступна как для вызывающей, так и для вызываемой процедур.

COMMON — Определение общих областей данных

Директива **COMMON** используется для объявления общих переменных.

Общая переменная очень похожа на глобальную переменную, объявленную в разделе неинициализированных данных (BSS), так что

```
common      intvar  4
```

функционально похож на

```
global      intvar
section    .bss
intvar      resd    1
```

Разница в том, что если несколько модулей определяют одну и ту же общую переменную, то во время компоновки эти переменные будут объединены, и ссылки на **intvar** во всех модулях будут указывать на один и тот же фрагмент памяти.

В языке C нет директивы/спецификатора **COMMON** — эту роль выполняет **extern** без инициализации. Неинициализированные **extern**-объекты, объявленные в разных модулях, после сборки совместно используют одну и ту же память, выделенную глобальной переменной с таким же именем (под неинициализированные **extern**-объекты память не выделяется).

Как и **GLOBAL** и **EXTERN**, директива **COMMON** поддерживает специфичные для объектного формата расширения. Например, формат elf позволяет указать требования к выравниванию общей переменной:

```
common  intarray 100:4   ; works in ELF: 4 byte aligned
```

Если переменная объявлена как **COMMON** и **EXTERN**, она будет рассматриваться как **COMMON**.

Через таблицу параметров

В сегменте данных выделяется дополнительная память — таблица параметров. В эту таблицу перед вызовом процедуры заносятся значения параметров, а в подпрограмму передается только адрес таблицы (например через регистр).

Метод часто используется в архитектурах, в которых отсутствует аппаратная поддержка стека (IBM System 360/370/390).

Просто ради информирования, что собой представляет IBM System

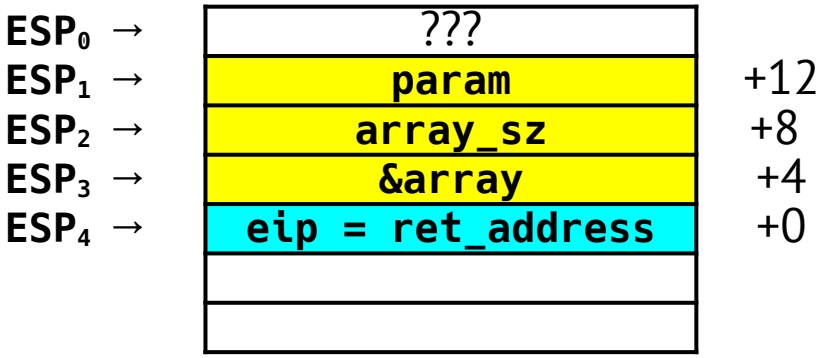
Info: IBM System Z10

RAM — 1.5TB
nProc — 77
IO — 288 GB/сек.

Передача параметров через стек

Перед вызовом процедуры все параметры заносятся в стек. Следующая за этим инструкция вызова процедуры **CALL** поместит в стек значение **eip** следующей команды в потоке команд (**ret_address**).

```
foo(&array, array_sz, param);
```



Размещенные в стеке параметры удаляются из стека по завершении работы подпрограммы.

Очистка стека выполняется:

1) внутри вызываемой подпрограммы;

2) вызывающей процедурой после возврата ей управления.

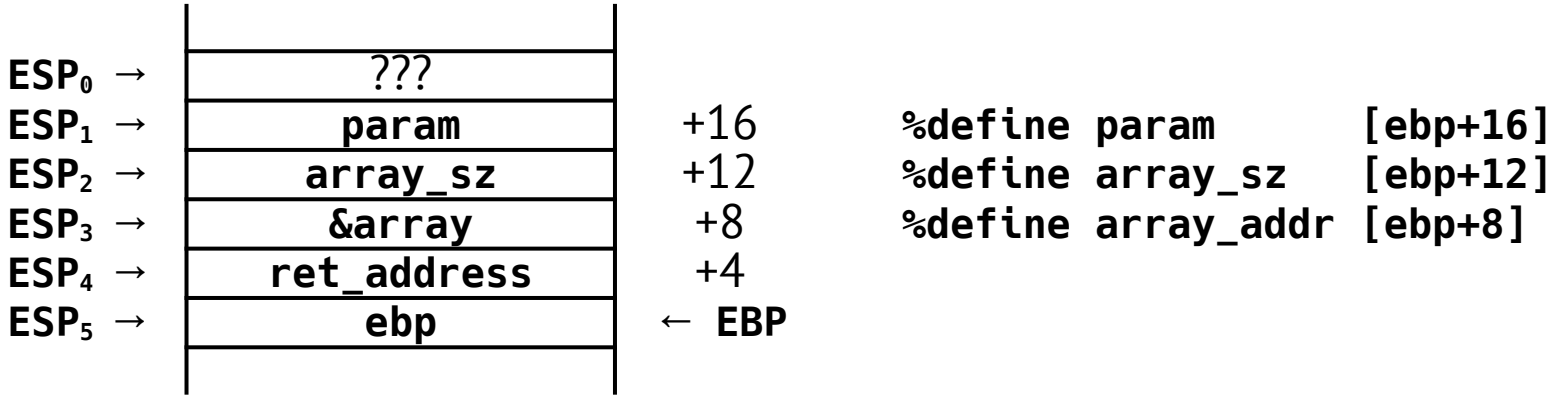
ret n ; возврат с очисткой стека — значение n добавляется к ESP с целью удаления параметров, которые были помещены в стек вызывающей процедурой.

Обычно подпрограммы очень активно используют стек.

Для того что бы иметь доступ к сохраненным в стеке аргументам, необходимо в начале вызываемой процедуры загрузить в регистр **ebp** адрес вершины стека, для чего пишут пролог:

```
push ebp        ; сохранить базу стекового кадра вызывающей процедуры
mov  ebp, esp   ; загрузить базу текущего стекового кадра
```

После этого можно обращаться к аргументам процедуры, размещенным в стеке, относительно адреса хранящегося в **ebp**.



Для восстановления состояния стека до его использования подпрограммой, необходимо загрузить в регистр **esp** адрес хранящийся в **ebp** и восстановить **ebp** (написать эпилог):

```
mov esp, ebp ; если баланс стека соблюден, инструкция необязательна
pop ebp
```

и вытолкнуть параметры из стека

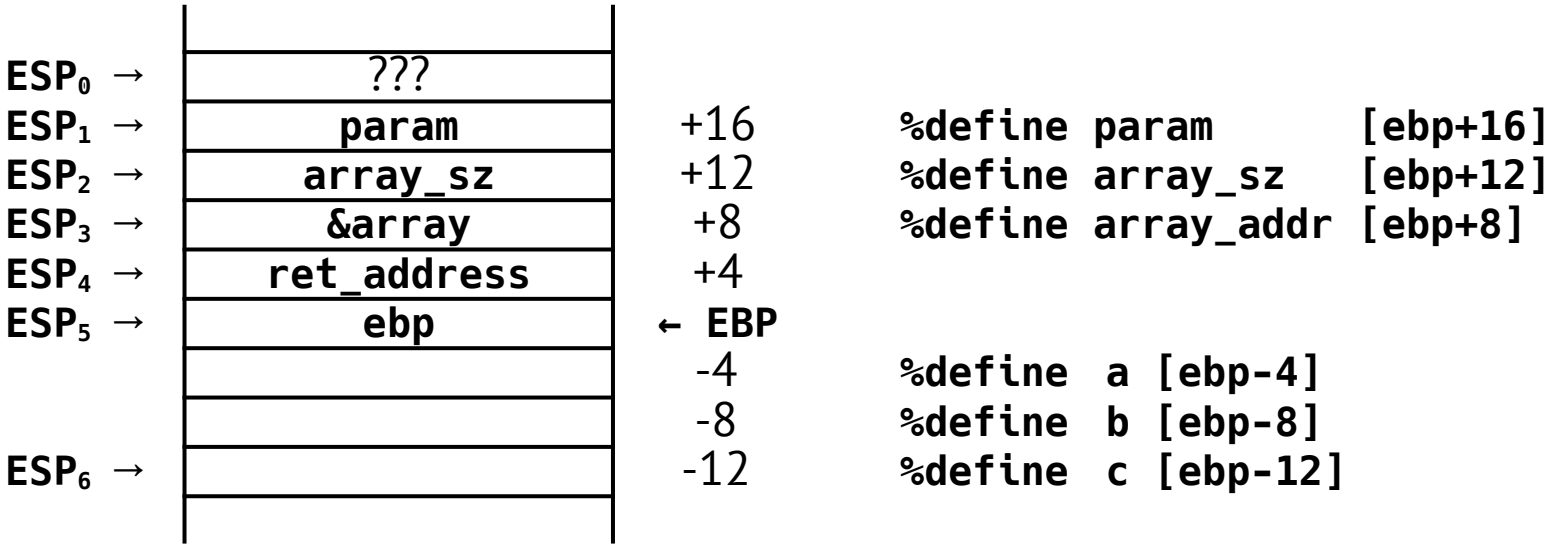
```
ret 12
```

Если число параметров переменное, они помещаются в стек как обычно, однако перед вызовом процедуры в стек помещается число — количество переданных параметров. Таким образом, вызываемая процедура может «узнать», сколько параметров ей реально передано и может соответственно скорректировать свое поведение.

Порядок загрузки стека параметрами для функции в языке C — справа налево.

Локальная память процедуры

```
push ebp      ; сохраняем ebp
mov  ebp, esp ; сохраняем вершину стека на входе в процедуру
sub  esp, 3*4 ; фрейм под локальные переменные a, b и c
```




```
mov esp, ebp ; восстанавливаем вершину стека на выходе из процедуры
pop ebp      ; восстанавливаем ebp
ret 12
```

Работа с регистрами

Возврат значения из процедуры или функции — обычно **eax/ax/al** или **edx:eax/dx:ax**.

Соглашения об использовании регистров часто определяются компиляторами с ЯВУ.

При этом есть возможности эти соглашения в определенных границах скорректировать.

В ассемблере с этим полная свобода и есть два крайних случая:

1) все регистры разрешается портить;

2) нельзя портить никакие регистры.

CDECL — компромиссное решение, используемое большинством компиляторов на платформе x86.

Подпрограмма имеет право портить **eax**, **edx**, **ecx**.

Все остальные РОН либо она не трогает, либо если она собирается какие-то использовать, должна сохранять.

Q: Почему именно **eax, edx, ecx?**

A: Эти регистры используются инструкциями на безальтернативной основе.

eax, **edx:eax** — в арифметических инструкциях, **ecx** — в качестве счетчика.

Так же есть инструкция XLAT, которая использует регистр **EBX**.

Системные вызовы

Системные вызовы (*syscalls*) — это вызовы функций, сделанные из пользовательского пространства в ядро для запроса какой-либо службы или ресурса из операционной системы.

Системные вызовы варьируются от привычных, таких как **`read()`** и **`write()`**, до экзотических, таких как **`get_thread_area()`** и **`set_tid_address()`**.

Linux реализует гораздо меньше системных вызовов, чем большинство других ядер операционных систем. Например, количество системных вызовов архитектуры ia32/x86_64 составляет более 400/350 по сравнению с якобы тысячами системных вызовов в Microsoft Windows.

```
$ cat /usr/include/asm/unistd_32.h | grep '__NR_' | wc
    439    1317   11822
$ cat /usr/include/asm/unistd_64.h | grep '__NR_' | wc
    361    1083    9633
$ rpm -qf /usr/include/asm/unistd_32.h
kernel-headers-5.16.5-100.fc34.x86_64
```

В ядре Linux каждая машинная архитектура (например, Alpha, i386 или PowerPC) реализует свой собственный список доступных системных вызовов. Следовательно, системные вызовы, доступные в одной архитектуре, могут отличаться от доступных в другой.

Тем не менее, очень большое подмножество системных вызовов — более 90 процентов — реализовано всеми архитектурами.

Порядок вызова

Напрямую связать приложения пользовательского пространства с пространством ядра невозможно — из соображений безопасности и надежности приложениям пользовательского пространства нельзя разрешать непосредственно выполнять код ядра или манипулировать данными ядра.

Вместо этого ядро предоставляет механизм, с помощью которого приложение пользовательского пространства может «сигнализировать» ядру о том, что оно желает вызвать системный вызов.

Приложение может обратиться к ядру с помощью этого четко определенного механизма и выполнять только тот код в пространстве ядра, который ядро позволяет ему выполнить.

Конкретный механизм варьируется от архитектуры к архитектуре.

Например, на ia32 приложение из пользовательского пространства выполняет команду программного прерывания **`INT`** со значением **`0x80`**.

На x86_64 для этой цели используется инструкция **`SYSCALL`**, которая выполняет необходимые действия вместо **`INT 80h`**.

Инструкция **`INT`** вызывает переключение в пространство ядра — в защищенную область, где ядро выполняет программный обработчик прерываний. В случае ia32 обработчик программного прерывания **`INT 0x80`** — обработчик системных вызовов!

Приложение сообщает ядру, какой системный вызов выполнять и с какими параметрами, используя регистры процессора.

Системные вызовы обозначаются числом, начиная с 0.

В архитектуре ia32, чтобы запросить сервис системного вызова с номером 5 (**`open()`**), прикладные программы пользовательского пространства перед выполнением инструкции **`int`** заносят 5 в регистр **`eax`**.

Передача параметров осуществляется аналогичным образом. Если число параметров не превышает пяти, используются регистры (именно в таком порядке)

```
ebx, ecx, edx, esi, edi
```

которые содержат первые пять параметров.

В случаях системного вызова с более чем пятью параметрами один регистр используется для указания буфера в пользовательском пространстве, где хранятся все параметры.

Большинство системных вызовов имеют всего пару параметров.

Другие архитектуры могут обрабатывать системные вызовы по другому, хотя идея аналогичная.

Библиотека C

Библиотека C (**`libc`**) лежит в основе всех приложений Unix (Linux, IOS, MacOS, CISCO, ...).

Даже в случае программирования на другом языке с вероятностью чуть менее, чем 100%, используется библиотека C.

Обернутая библиотеками более высокого уровня, она предоставляет базовые сервисы и упрощает вызов системных функций.

В современных системах Linux библиотека C предоставляется GNU libc, сокращенно **`glibc`** и произносится как *gee-lib-see* или, реже, *glib-see*.

API и ABI

Программисты, заинтересованы в том, чтобы их программы работали на всех системах сейчас и в будущем. Это называется — ***переносимость.***

На системном уровне существует два отдельных набора определений и описаний, которые влияют на переносимость.

Одним из них является **интерфейс прикладного программирования (API)**, а другим — **бинарный (двоичный) интерфейс приложения (ABI)**. Оба определяют и описывают интерфейсы между различными частями компьютерного программного обеспечения.

API — интерфейс прикладного программирования

API определяет интерфейсы, с помощью которых одна часть программного обеспечения взаимодействует с другой ***на уровне исходных кодов***.

Он обеспечивает абстракцию, предоставляя стандартный набор интерфейсов, обычно функций, которые одна часть программного обеспечения может вызывать из другой части программного обеспечения. Например, API может абстрагировать концепцию рисования текста на экране с помощью семейства функций, которые обеспечивают все необходимое для рисования текста.

API только лишь определяет интерфейс — часть программного обеспечения, которая реально обеспечивает API, называется **реализация API**.

Обычно пользователь API (как правило, программное обеспечение более высокого уровня) имеет нулевой вклад как в API, так и в его реализацию. Он может использовать API как есть или не использовать его вообще.

API всего лишь гарантирует, что если обе части программного обеспечения следуют данному API, они будут совместимы с исходным кодом и пользователь API успешно скомпилирует (compile) и скомпонует (link) свои программы.

Реальным примером является API, определенный стандартом ISO/IEC-9899 языка C и реализованный стандартной библиотекой C.

Этот API-интерфейс определяет семейство основных и необходимых функций, таких как процедуры манипуляции со строками или стандартная библиотека ввода/вывода.

ABI — бинарный (двоичный) интерфейс приложения

ABI определяет **низкоуровневый двоичный интерфейс** между двумя или более частями программного обеспечения **в конкретной архитектуре и операционной системе**. Он определяет, как приложение взаимодействует с самим собой, как оно взаимодействует с ядром и как оно взаимодействует с библиотеками. ABI можно рассматривать как набор правил, позволяющих компоновщику объединять откомпилированные модули без перекомпиляции всего кода, в то же время определяя двоичный интерфейс, чтобы обеспечить двоичную совместимость.

ABI связаны с такими проблемами, как:

- соглашения о вызовах подпрограмм;
- порядок следования байт;
- использование регистров процессора;
- порядок использования системных вызовов (состав и формат системных вызовов);
- связывание (компоновка);
- поведение библиотек;
- форматы двоичных объектов (в операционной системе).

Например, соглашение о вызовах определяет, как функции вызываются, как аргументы передаются в функции, какие регистры сохраняются, а какие портятся, и как вызывающая сторона получает возвращаемое значение.

Операционные системы, включая Linux, определяют свои собственные ABI, как они считают нужным. ABI тесно связан с архитектурой — подавляющее большинство ABI говорит о конкретных для машины понятиях, таких как конкретные регистры или инструкции ассемблера. Таким образом, каждая машинная архитектура имеет свой собственный ABI.

Чем ABI отличается от API

API описывает функциональность, предоставляемую библиотеками — список функций, аргументы функций, возвращаемые значения, возможные исключения, поведение функций.

ABI описывает функциональность, предоставляемую ядром ОС и архитектурой набора команд (без привилегированных команд).

Если API разных платформ совпадают, код для этих платформ можно компилировать и без изменений.

Если для разных платформ совпадают и API, и ABI, исполняемые файлы можно переносить на эти платформы без изменений (если они опираются на один и тот же ISA).

Если API или ABI платформ отличаются, код требует изменений и повторной компиляции.

ABI обеспечивает совместимость среды выполнения программы — API нет.

Соглашения о вызовах, используемые разными компиляторами с языка C++ в разных ОС

<https://agner.org/optimize/calling_conventions.pdf>

Agner Fog. Calling conventions for different C++ compilers and operating systems. Technical University of Denmark. Copyright © 2004 - 2021. Last updated 2021-01-31.

EABI — Бинарный интерфейс встраиваемых приложений

Бинарный интерфейс встраиваемых приложений (англ. *embedded application binary interface*, EABI) — набор соглашений для использования во встраиваемом программном обеспечении, описывающий:

- форматы файлов;
- типы данных;
- способы использования регистров;
- организацию стека;
- способы передачи параметров функциям.

Если объектный файл был создан компилятором, поддерживающим EABI, становится возможной компоновка этого объектного файла любым компоновщиком, поддерживающим тот же EABI.

Основное отличие EABI от ABI в ОС общего назначения заключается в том, что в коде встраиваемых приложений допускаются привилегированные команды, а динамическое связывание (компоновка) не требуется (а иногда и полностью запрещена), а также, в целях экономии памяти, используется более компактная организация стека.

Соглашения о вызове

Соглашение о вызове (англ. calling convention) — часть двоичного интерфейса приложений (англ. application binary interface, ABI), которая регламентирует технические особенности вызова подпрограммы, передачи параметров, возврата из подпрограммы и передачи результата вычислений в точку вызова.

Состав

Соглашение о вызове описывает следующее:

- способ передачи аргументов в функцию;
- порядок размещения аргументов в регистрах и/или стеке;
- код, ответственный за очистку стека;
- конкретные инструкции, используемые для вызова и возврата;
- способ передачи в функцию указателя на текущий объект (**`this`** или **`self`**) в объектно-ориентированных языках;
- код, ответственный за сохранение и восстановление содержимого регистров до и после вызова функции;
- список регистров, подлежащих сохранению/восстановлению до/после вызова функции.

Способ передачи аргументов в функцию

– аргументы передаются через регистры процессора;

– аргументы передаются через стек;

– смешанные (соответственно, стандартизируется алгоритм, определяющий, что передаётся через регистры, а что — через стек или другую память).

1) первые несколько аргументов передаются через регистры, остальные через стек (не-большие аргументы[1]) или другую память (большие аргументы[2]);

2) аргументы небольшого размера передаются через стек, большие аргументы через другую память.

Порядок размещения аргументов в регистрах и/или стеке

- слева направо или прямой порядок. Аргументы размещаются в том же порядке, в котором они перечислены при вызове функции. Достоинство — машинный код соответствует коду на языке высокого уровня;

- справа налево или обратный порядок. Аргументы передаются в порядке от конца к началу. Достоинство — упрощается реализация функций, принимающих произвольное число аргументов (например, **`printf()`**) (так как на вершине стека оказывается всегда первый аргумент).

---

1 Под небольшими аргументами понимаются значения, размер которых меньше или равен размеру регистра процессора. Например, 1, 2 и 4 байта для процессора ia32, работающего в 32-битном режиме.

2 Под большими аргументами понимаются значения, размер которых больше размера регистра процессора. Например, 8 и более байт для процессора ia32, работающего в 32-битном режиме.

Код, ответственный за очистку стека

- код, вызывающий процедуру/функцию, или ***вызывающая процедура/функция***. Достоинство — возможность передачи в функцию произвольного числа аргументов;

- код самой процедуры/функции или ***вызываемая процедура/функция***. Достоинство — уменьшение количества инструкций, необходимых для вызова функции (инструкция для очистки стека записывается в конце кода функции и только один раз).

Конкретные инструкции, используемые для вызова и возврата

Для процессора ia32, работающего в защищённом режиме, используются исключительно инструкции **`call`** и **`ret`**.

При работе в реальном режиме используются инструкции **`call near`**, **`call far`** и **`pushf/call far`** (для возврата соответственно **`retn`**, **`retf`** и **`iret`**);

Способ передачи в функцию указателя на текущий объект (this или self)

Передача в функцию указателя на текущий объект (this или self) используется в объектно-ориентированных языках. Варианты (для процессора x86, работающего в защищённом режиме):

- как первый аргумент;
- через регистр **`ecx`** или **`rcx`**.

Код, ответственный за сохранение содержимого регистров

- вызывающая функция;
- вызываемая функция.

Соглашение о вызове может быть описано в документации к ABI архитектуры, в документации к ОС или в документации к компилятору.

Соглашения о вызовах более подробно

Соглашения о вызовах, используемые разными компиляторами с языка C++ в разных ОС

Agner Fog. Calling conventions for different C++ compilers and operating systems. Technical University of Denmark. Copyright © 2004 - 2021. Last updated 2021-01-31.
<https://www.agner.org/optimize/calling_conventions.pdf>

Рассматриваются для 16, 32, 64-разрядных компиляторов:

- размеры данных;
- выравнивание статических данных;
- выравнивание членов структур;
- выравнивание стека;
- использование регистров в коде пользовательского уровня (свободные, сохраняемые, для передачи параметров, для возврата);
- использование регистров в коде ядра;
- использование управляющих флагов — направления, прерывания;
- кто очищает стек.

Соглашения о вызовах, используемые на x86 при 32-битной адресации

Основные из применяемых на сегодня соглашений:

```
__cdecl;
__pascal;
__fortran;
__thiscall;
__stdcall;
__fastcall;
__msfastcall;
__regcall;
__vectorcall;
interrupt.
```

Способ передачи параметров в функцию по умолчанию не всегда так стандартизирован, как хотелось бы. Однако, во многих случаях можно указать конкретное соглашение о вызовах в объявлениях C++, например:

```
int __stdcall foo(int n, double x, double y, double z);
```

В любом случае необходимо смотреть в документацию конкретного компилятора, чтобы избежать проблем.

Соглашения о вызове **`__cdecl`**, **`__stdcall`**, **`__pascal`**, **`__fastcall`** и прочие можно явно указать в объявлениях C++ функций для компиляторов, которые поддерживают эти соглашения.

__cdecl используется по умолчанию для приложений и статических библиотек.

__stdcall является значением по умолчанию для системных вызовов (включая вызовы API Windows) и рекомендуется для библиотек DLL в 32-разрядной версии Windows.

__thiscall по умолчанию используется в компиляторах Microsoft для функций-членов в 16- и 32-битном режиме.

Для перечисленных соглашений (кроме **__cdecl**) перед возвратом значений из функции подпрограмма обязана восстановить значения сегментных регистров, регистров **esp** и **ebp**. Значения остальных регистров могут не восстанавливаться.

Если размер возвращаемого значения функции не больше размера регистра **eax**, возвращаемое значение сохраняется в регистре **eax**. Иначе, возвращаемое значение сохраняется на вершине стека, а указатель на вершину стека сохраняется в регистре **eax**.

Совместимость

Компиляторы Intel для Windows совместимы с Microsoft.

Компиляторы Intel для Linux совместимы с Gnu.

Компиляторы Symantec, Digital Mars и Codeplay совместимы с Microsoft.

В 64-битном режиме для каждой операционной системы существует единственное соглашение о вызовах по умолчанию — другие соглашения о вызовах в 64-битном режиме встречаются редко.

`__cdecl`

`__cdecl` (сокращение от англ. c-declaration) — соглашение о вызовах, используемое компиляторами для языка Си (отсюда название).

1) Аргументы функций передаются через стек, справа налево.

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

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

4) Способы получения возвращаемого значения функции приведены в таблице.

Тип	Размер возвращаемого значения, байт	Способ передачи возвращаемого значения	Примечание
Целое, указатель	1, 2, 4	`eax`	Значения, размер которых меньше 4-х байт, расширяются до 4-х байт
Целое	8	`edx:eax`	
FP-число	4, 8	`st0` (из стека FPU)	
Другие	> 8	`[eax]`	Указатель на структуру данных сохраняется в `eax`

5) Перед вызовом функции вставляется код, называемый ***прологом*** (англ. prolog)) и выполняющий следующие действия:

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

6) После вызова функции вставляется код, называемый ***эпилогом*** (англ. epilog)) и выполняющий следующие действия:

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

__pascal

pascal — соглашение о вызовах, используемое компиляторами для языка Паскаль. Также применялось в ОС Windows 3.x.

1) Аргументы процедур и функций передаются через стек, слева направо.

2) Указатель на вершину стека (значение регистра **esp**) на исходную позицию возвращает подпрограмма (**ret n**).

3) Изменяемые параметры передаются только по ссылке.

4) Возвращаемое значение передаётся через изменяемый параметр **Result**.

5) Параметр **Result** создаётся неявно и является первым аргументом функции.

`__stdcall (winapi)`

`stdcall` или **`winapi`** — соглашение о вызовах, применяемое в ОС Windows для вызова функций WinAPI.

1) Аргументы функций передаются через стек, справа налево.

2) Очистку стека производит вызываемая подпрограмма (**`ret n`**), поэтому для функций с переменным количеством аргументов компилятор микрософт использует **`__cdecl`**.

3) Аргументы пеердаются по значению.

4) Перед именем ставится префикс подчеркивания (_). За именем следует знак **`@`**, за которым следует количество байтов (в десятичном формате) в списке аргументов. Поэтому функция, объявленная как **`int func( int a, double b )`**, будет оформлена следующим образом: **`_func@12`**.

__fastcall

fastcall — соглашение о вызовах, применяемое компиляторами фирмы Borland для языка Delphi.

1) Передача параметров подпрограмм и функций осуществляется через регистры (обычно, это самый быстрый способ — отсюда название).

2) Если для сохранения всех параметров и промежуточных результатов регистров не достаточно, используется стек.

3) Соглашение о вызовах **fastcall** не стандартизировано, поэтому используется только для вызова процедур и функций. Для экспортируемых из даного модуля и не импортируемых извне не используется.

4) В компиляторе фирмы Borland для соглашения **__fastcall**, называемого также **register**, параметры передаются слева направо в регистрах **eax**, **edx**, **ecx**, а если параметров больше трёх — в стеке, также слева направо.

5) Исходное значение указателя на вершину стека (значение регистра **esp**) возвращает вызываемая подпрограмма.

6) В 32-разрядной версии компилятора фирмы Microsoft, а также в компиляторе GCC, соглашение **__fastcall**, также называемое **__msfastcall**, определяет передачу первых двух параметров слева направо в регистрах **ecx** и **edx**, а остальные параметры передаются справа налево в стеке.

7) Очистку стека производит вызываемая подпрограмма.

__safecall

safecall — соглашение о вызовах, используемое для вызова методов интерфейсов COM.

Методы интерфейсов COM представляют собой функции, возвращаемое значение которых имеет тип **HRESULT**.

1) После вызова функции добавляется код, который анализирует возвращаемое значение.

2) При наличии ошибки код записывает код ошибки, сообщение о ошибке и генерирует исключение.

3) В случае успешного завершения настоящее возвращаемое значение скрывается, а вместо него используется параметр, передаваемый в функцию последним по ссылке.

Например, можно считать два следующих объявления функции эквивалентными.

```
// safecall
function DoSomething(a:DWORD):DWORD; safecall;

// симуляция safecall
function DoSomething(a:DWORD; out Result:DWORD):HResult; stdcall;
```

thiscall

thiscall — соглашение о вызовах, используемое компиляторами для языка C++ при вызове функций-членов классов.

1) Аргументы функции передаются через стек, справа налево.
2) Очистку стека производит вызываемая функция.
3) Указатель на объект, для которого вызывается функция-член (указатель **`this`**), записывается в регистр **`ecx`**.

Представление данных

Segmetnt word size	32 bit						64 bit			
compiler	MS	Intel Win	Borla	Watc	GCC,Clang	Int Linux	MS	Intel Win	GCC,Clang	Int Linux
bool	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1
wchar_t	2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2
int	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	8	8
int64_t	8	8			8	8	8	8	8	8
enum (typical)	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8
long double	8	16	10	8	12	12	8	16	16	16
__m64	8	8			8	8		8	8	8
__m128	16	16			16	16	16	16	16	16
__m256	32	32			32	32	32	32	32	32
__m512	64	64			64	64	64	64	64	64
pointer	4	4	4	4	4	4	8	8	8	8
function pointer	4	4	4	4	4	4	8	8	8	8
data member ptr (min)	4	4	8	4	4	4	4	4	8	8
data member ptr (max)	12	12	8	12	4	4	12	12	8	8
member func ptr (min)	4	4	12	4	8	8	8	8	16	16
member func ptr (max)	16	16	12	16	8	8	24	24	16	16