6

Структурное программирование

[Использование процедур при разработке программ](#_Toc322757336)

[Операции со стеком](#_Toc322757337)

[Стековая организация памяти](#_Toc322757338)

[Использование стека](#_Toc322757339)

[Команда PUSH](#_Toc322757340)

[Команда POP](#_Toc322757341)

[Команды PUSHFD и POPFD](#_Toc322757342)

[Команды PUSHAD, PUSHA, POPAD и РОРА](#_Toc322757343)

[Определение и использование процедур](#_Toc322757344)

[Определение процедуры](#_Toc322757345)

[Команды CALL и RET](#_Toc322757346)

[Локальные и глобальные метки](#_Toc322757347)

[Передача параметров процедурам через регистры](#_Toc322757348)

[Сохранение и восстановление регистров](#_Toc322757349)

[Оператор USES](#_Toc322757350)

[Локальные переменные](#_Toc322757351)

[Директива LOCAL](#_Toc322757352)

[Стековые параметры](#_Toc322757353)

[Директива INVOKE](#_Toc322757354)

[Директива PROC](#_Toc322757355)

[Процедура сравнения строк](#_Toc322757356)

[Структуры](#_Toc322757357)

[Определение структуры](#_Toc322757358)

[Объявление структурных переменных](#_Toc322757359)

[Обращение к структурным переменным](#_Toc322757360)

[Вычисление площади прямоугольника](#_Toc322757361)

[Задания к работе](#_Toc322757362)

# Использование процедур при разработке программ

Любая программа, кроме самой простой, состоит из некоторого количества команд, выполняющих различные задачи, обусловленные алгоритмом. Можно, конечно, написать одну большую программу, весь код которой состоит из одной процедуры. Однако через некоторое время вы поймете, что разобраться в ней, а тем более отладить код, становится все труднее и труднее. Интуиция нам подсказывает, что одну большую программу нужно разделить на части, каждая из которых будет решать какую-то отдельную задачу, и оформить эти фрагменты кода в виде отдельных процедур. Причем эти процедуры могут находиться как в одном, так и в нескольких исходных файлах, содержащих программный код.

Перед тем как приступить к написанию программы, желательно сначала составить спецификацию этой программы, то есть перечень требований, которым должна отвечать программа, и список выполняемых ею действий. Обычно спецификация составляется в результате тщательного анализа поставленной перед вами задачи. Готовую спецификацию можно взять за основу при разработке программы.

Как было уже сказано выше, при использовании стандартного подхода к проектированию программ, сложная задача разбивается на ряд более простых, решение каждой из которых оформляется в виде отдельной процедуры. Процесс разбиения сложной задачи на ряд простых задач называют функциональной декомпозицией, а такой подход к проектированию – нисходящим. Ниже приведены несколько предпосылок, положенных в основу нисходящего подхода к проектированию.

* Одну сложную задачу можно легко разделить на ряд простых подзадач.
* Программу гораздо легче написать, отладить и сопровождать, если каждую процедуру можно протестировать независимо от других.
* Использование нисходящего подхода к проектированию позволяет увидеть существующие взаимосвязи между процедурами.
* Когда создана общая структура проекта, вы легко можете сосредоточиться на решении конкретных задач, а также написании кода, реализующего каждую процедуру.

# Операции со стеком

Стек – это структура данных типа LIFO (last-in, first-out, или последним пришел, первым обслужили), широко применяемая в вычислительной технике для хранения данных и получения к ним доступа. Стек поддерживает операции добавления и удаления данных, при этом новые элементы всегда добавляются в вершину стека, а существующие элементы всегда удаляются, начиная с самого верхнего. Стековая организация памяти широко используется в большом классе приложений.

Процессоры семейства x86 поддерживают на аппаратном уровне операции со специальной областью памяти, доступ к которой организован по принципу стека. Эта область памяти используется для реализации механизма вызова процедур, передачи им параметров, возврата управления из процедуры в точку, следующую за ее вызовом, а также для временного хранения данных. Для упрощения мы будем называть такую организацию памяти просто стеком.

## Стековая организация памяти

Стековая организация памяти представляет собой непрерывный блок оперативной памяти, доступ к которому осуществляется непосредственно центральным процессором с помощью двух регистров: SS и ESP. При работе в защищенном режиме в регистре SS хранится указатель на дескриптор сегмента, причем содержимое регистра SS не изменяется пользовательской программой. В регистре ESP хранится 32-разрядное смещение вершины стека, на которой располагается последний добавленный в стек элемент. Его содержимое изменяется автоматически такими командами, как CALL, RET, PUSH и POP. Кроме того, в специальных случаях в программах возникает необходимость изменять регистр ESP напрямую.

## Использование стека

Ниже перечислены основные причины использования стека в программах.

* Стек представляет собой очень удобное место для сохранения значения регистров, в случае если они используются для нескольких целей. После изменения содержимого регистра, его первоначальное значение можно легко восстановить.
* При выполнении команды CALL процессор сохраняет в стеке адрес следующей за ней команды. Тем самым обеспечивается возврат из процедуры и передача управления следующей за CALL команде.
* При вызове процедуры ей обычно передается ряд входных параметров, которые могут быть помещены в стек.
* Сразу после вызова внутри процедуры создается ряд локальных переменных, которые тоже располагаются в стеке. Значение этих переменных теряется при возврате управления в вызвавшую программу.

## Команда PUSH

Команда PUSH помещает в стек значение 16- или 32-разрядного операнда, уменьшая перед этим значение регистра ESP, соответственно, на 2 или 4. Существуют следующие форматы команды PUSH:

push *reg16*

push *mem16*

push *imm16*

push *reg32*

push *mem32*

push *imm32*

В защищенном режиме размер непосредственно заданного в команде PUSH операнда составляет 32 бита. В реальном режиме, если не задана директива определения процессора .386 (или выше), по умолчанию используется 16-разрядное значение.

При выполнении 32-разрядной операции помещения в стек сначала из регистра указателя стека ESP вычитается число 4, а затем по хранящемуся в нем адресу записывается выталкиваемое в стек число.

Обратите внимание, что при помещении элемента в стек значение в регистре ESP уменьшается, что фактически приводит к тому, что стек растет в сторону нижних адресов памяти («вниз»). Однако, несмотря на направление роста, стековая модель памяти всегда подчиняется принципу LIFO (последним пришел, первым обслужили).

Предположим, что текущая вершина стека равна ESP = 00001000h, и выполняется команда

push 20h

Изменение состояния стека при этом показано на рисунке 1.

10h

00001000h

00000FFCh

00000FF8h

00000FF4h

ESP

10h

00001000h

00000FFCh

00000FF8h

00000FF4h

ESP

20h

*a)*

*b)*

Рисунок - Пример стека до (a) и после (b) выполнения команды push 20h

## Команда POP

Команда POP копирует содержимое вершины стека, на которую указывает регистр ESP, в 16- или 32-разрядный операнд, указанный в команде, а затем прибавляет к регистру ESP, соответственно, число 2 или 4. Существует следующие форматы команды POP:

pop *reg16*

pop *mem16*

pop *reg32*

pop *mem32*

После того как число извлечено из стека, регистр ESP будет указывать на следующий по порядку элемент, расположенный на вершине стека.

Предположим, что для стека из предыдущего примера выполняется команда

pop eax

Изменение состояния стека при этом показано на рисунке 2.

10h

00001000h

00000FFCh

00000FF8h

00000FF4h

ESP

10h

00001000h

00000FFCh

00000FF8h

00000FF4h

ESP

20h

*a)*

*b)*

20h

Рисунок - Пример стека до (a) и после (b) выполнения команды pop eax

## Команды PUSHFD и POPFD

Команда PUSHFD помещает в стек значение 32-разрядного регистра флагов процессора ЕFLAGS, а команда POPFD выполняет обратную операцию, то есть восстанавливает значение регистра ЕFLAGS из стека. У этих команд операндов нет:

pushfd

popfd

В реальном режиме для сохранения в стеке значения 16-разрядного регистра флагов FLAGS используется команда PUSHF, а для выполнения обратной операции – команда POPF.

Команды сохранения флагов иногда используются для помещения значения регистра флагов в переменную или регистр общего назначения, что может быть удобно для последующего анализа состояния флагов при помощи команд побитовой логики. Следующий фрагмент программы помещается значение регистра флагов в регистр EAX:

pushfd ; Помещаем значение регистра флагов в стек

pop eax ; Выталкиваем регистр флагов из стека в EAX

## Команды PUSHAD, PUSHA, POPAD и РОРА

Команда PUSHAD сохраняет в стеке значение всех 32-разрядных регистров общего назначения в следующем порядке: ЕАХ, ЕСХ, EDX, ЕВХ, ESP (то значение, которое было до выполнения команды), EBP, ESI и EDI. Команда POPAD выполняет обратную операцию, то есть восстанавливает из стека значения указанных регистров в обратном порядке. По аналогии, команда PUSHA, появившаяся в процессоре 80286, сохраняет в стеке значение всех 16-разрядных регистров общего назначения в следующем порядке: АХ, СХ, DX, ВХ, SP (то значение, которое было до выполнения команды), BP, SI и DI. Команда РОРА выполняет обратную операцию, восстанавливая из стека значения указанных регистров в обратном порядке.

Команда PUSHAD обычно используется в начале процедуры или фрагмента кода, в котором модифицируется много 32-разрядных регистров общего назначения. Для восстановления первоначального значения этих регистров в конце процедуры или фрагмента кода используется команда POPAD. Ниже приведен фрагмент кода:

MySub PROC

pushad ; Сохраняем регистры

mov eax, 1

mov ebx, 2

mov ecx, 3

; и так далее, различные операции с регистрами

...

popad ; Восстанавливаем регистры

ret

MySub ENDP

Такой подход может оказаться удобным, но не оптимальным с точки зрения производительности. Поэтому чаще используется индивидуальное сохранение регистров, когда процедуры явно сохраняют только те регистры, которые действительно используются при помощи команд PUSH и POP.

# Определение и использование процедур

При изучении языков высокого уровня основной акцент делается на декомпозицию программы на законченные логические модули, которые называются функциями. Эго позволяет разбить решение любой сложной проблемы на ряд простых задач, которые легко можно описать, реализовать и отладить. В языке ассемблера для обозначения логических модулей программы принято использовать другой, более общий термин – процедура.

## Определение процедуры

Нестрого процедуру можно определить как именованный блок команд, оканчивающийся оператором возврата. Для объявления процедуры используются директивы PROC и ENDP. При объявлении процедуре должно быть назначено имя, которое является одним из разрешенных идентификаторов. В каждой из программ, приведенных ранее, была только одна процедура под названием main, например:

main PROC

main ENDP

При создании большинства процедур, в их конце необходимо поместить команду RET. В результате процессор вернет управления команде, следующей за той, которая вызвала эту процедуру:

simple PROC

; тело процедуры

ret ; возврат из процедуры

simple ENDP

Давайте создадим процедуру Min, выполняющую сравнение чисел со знаком в регистрах EAX и EBX, и возвращающую в регистре EAX минимальное из них.

Min PROC

cmp eax, ebx

jl minExit ; в EAX уже находится минимально число

mov eax, ebx

minExit:

ret ; возврат из процедуры

Min ENDP

Хорошим стилем программирования считается использование в программе коротких и понятных комментариев, которые позволяют программисту быстро разобраться в ее сути. Ниже приведены несколько рекомендаций по поводу информации, которая должна быть размещена в начале каждой процедуры.

* Описание всех функций, выполняемых процедурой.
* Список входных параметров и описание их значений. Если какой-либо из параметров имеет особый тип, его нужно также указать. Обычно описание параметра начинается со слова Param.
* Список возвращаемых процедурой значений, указанных после слова Return.
* Перечень особых требований (если таковые имеются), которые должны быть удовлетворены перед вызовом процедуры. Они называются предусловиями. Например, для процедуры, которая чертит на экране прямую линию, одним из входных условий является работа видеоадаптера в графическом режиме.

## Команды CALL и RET

Команда CALL предназначена для передачи управления процедуре, адрес которой указывается в качестве параметра. При этом процессор начинает выполнять команду, расположенную по указанному адресу. Чтобы вернуть управление команде, расположенной сразу за CALL, в процедуре используется команда RET. Строго говоря, команда CALL помещает в стек текущее значение счетчика команд, который на фазе выполнения команды CALL содержит адрес следующей команды, а затем загружает в счетчик команд указанный адрес процедуры. При возврате из процедуры – при выполнении в ней команды RET, – адрес возврата загружается из стека в счетчик команд. Напомним, что процессор всегда выполняет команду, адрес которой указывается в регистре EIP.

Предположим, что в процедуре main по смещению 00000020h расположена команда CALL. В 32-разрядном режиме длина этой команды составляет 5 байтов. Поэтому следующая команда (в нашем случае M0V) будет расположена со смещением 00000025h:

main PROC

00000020 call MySub

00000025 mov eax, ebx

Далее, предположим, что первая команда процедуры MySub расположена со смещением 00000040h:

MySub PROC

00000040 mov eax, edx

ret

MySub ENDP

При выполнении команды CALL в стек помещается адрес следующей за ней команды (в данном случае 00000025h), после чего в регистр EIP загружается адрес процедуры MySub.

После этого процессор начинает выполнять последовательность команд процедуры MySub, пока в ней не встретится команда RET. При выполнении команды RET содержимое стека, на которое указывает регистр ESP, копируется в регистр EIP. В результате процессор после команды RET будет выполнять не следующую за ней команду, а команду, находящуюся по адресу 00000025h. А это как раз команда, расположенная следом за командой CALL, которая вызвала данную процедуру.

Вложенный вызов процедуры происходит, когда вызванная процедура вызывает еще одну процедуру до того, как управление будет передано вызывающей процедуре. Предположим, что из процедуры main вызывается процедура Sub1. При выполнении процедуры Sub1 вызывается процедура Sub2, которая в свою очередь вызывает процедуру Sub3.

При выполнении команды RET в конце процедуры Sub3, в счетчик команд будет занесено текущее содержимое вершины стека, на которое указывает регистр ESP. В результате процессор продолжит выполнение команд процедуры Sub2 и передаст управление команде, следующей за командой вызова процедуры Sub3 (call Sub3). После возврата в процедуру Sub2 регистр ESP будет указывать на следующий элемент в стеке. И наконец, при возврате из процедуры Sub1 из стека в указатель команд загрузится адрес команды, следующей за call Sub1. В результате процессор возобновит выполнение последовательности команд процедуры main.

Как видно из описанного примера, стек может использоваться в качестве некоего устройства для запоминания информации. С его помощью очень легко реализуется режим вложенного вызова процедур. А вообще говоря, стековые структуры часто используются в случаях, когда программа должна выполнить определенные команды в заданном порядке.

## Локальные и глобальные метки

В MASM по умолчанию метки кода (идентификаторы, после которых указано одно двоеточие) имеют локальную область видимости. Это означает, что ими можно пользоваться только внутри текущей процедуры. Тем самым в программе предотвращается случайный переход с помощью команды JMP или LOOP на метку, расположенную за пределами текущей процедуры. Однако хоть и не часто, но встречаются случаи, когда в программе требуется перейти на метку, расположенную вне текущей процедуры. Для этого нужно объявить глобальную метку, поставив после идентификатора два двоеточия, как показано ниже:

GlobalLabel::

В приведенном ниже фрагменте программы, команда перехода из процедуры main на метку L2 вызовет появление сообщения об ошибке, поскольку метка L2 является локальной для процедуры sub2. Переход же из процедуры sub2 на метку L1 корректен, поскольку L1 определена как глобальная метка:

main PROC

jmp L2 ; Ошибка, переход по локальной метке другой процедуры

L1:: ; Глобальная метка

ret

main ENDP

sub1 PROC

L2:

jmp L1 ; Переход по глобальной метке

ret

sub1 ENDP

## Передача параметров процедурам через регистры

При написании процедуры, которая выполняет одно из стандартных действий, например, такое как суммирование элементов целочисленного массива, не имеет смысла использовать в ней конкретные имена переменных. Дело в том, что тогда данная процедура сможет обрабатывать только элементы одного массива с указанным именем. Гораздо лучше при вызове процедуры передать ей в качестве параметров адрес массива и количество его элементов. Будем называть эти параметры аргументами или входными параметрами. В языке ассемблера для передачи параметров процедурам часто используются регистры общего назначения.

Напомним, что в предыдущем разделе мы создали простую процедуру Min, которая определяет минимальное из двух чисел, находящихся в регистрах ЕАХ и ЕВХ. Перед вызовом этой процедуры нам нужно загрузить соответствующие значения в регистры ЕАХ и ЕВХ:

mov eax, 20 ; Загрузим значения аргументов

mov ebx, 10

call Min ; Выполняет поиск минимального числа

mov [res], eax ; После вызова Min в EAX минимальное число

; сохраняем его

После вызова процедуры Min с помощью команды CALL в регистре ЕАХ будет находиться искомое минимальное число, которое мы можем сохранить в переменной для дальнейшего использования.

## Сохранение и восстановление регистров

Процедуры практически всегда используют дополнительные регистры в процессе выполнения. Такие процедуры должны в начале своей работы сохранять все затрагиваемые регистры в стеке, и перед выходом восстанавливать их значения. Это один из примеров хорошего стиля программирования. Если при написании процедур вы будете сохранять значения модифицируемых в них регистров, то тем самым облегчите последующую отладку программ. При этом в вызывающей программе не нужно будет отслеживать, какие из регистров «испортила» вызываемая программа.

## Оператор USES

Оператор USES, указанный сразу после директивы PROC, позволяет перечислить имена всех регистров, значение которых изменяется в процедуре. При его обработке компилятор ассемблера выполняет две вещи. Во-первых, в начале процедуры автоматически генерируется последовательность команд PUSH, с помощью которых в стеке сохраняются значения регистров, указанных в операторе USES. Во-вторых, при выходе из процедуры (точнее, перед каждой командой RET, если в процедуре их несколько) автоматически восстанавливаются значения этих регистров. Оператор USES указывается сразу за ключевым словом PROC. Список регистров следует сразу за ключевым словом USES, при этом имена регистров разделяются пробелом или символом табуляции (не запятой!).

Рассмотрим процедуру ArraySum, выполняющую вычисление суммы элементов массива, адрес которого передается в регистре ESI, а количество элементов – в регистре ECX. Поскольку в процессе прохождения по массиву значения этих регистров изменяются, процедура сохраняет исходные значения в регистрах, чтобы восстановить из при возврате.

ArraySum PROC USES esi ecx

xor eax, eax ; Обнулим значение регистра EAX

L1:

add eax, [esi] ; Прибавим значение очередного элемента массива

inc esi, 4 ; Перейдем к следующему элементу

loop L1

ret

ArraySum ENDP

В результате ассемблер сгенерирует приведенный ниже код.

ArraySum PROC

**push esi ; Сохранение регистров в стеке**

**push ecx**

xor eax, eax ; Обнулим значение регистра EAX

L1:

add eax, [esi] ; Прибавим значение очередного элемента массива

inc esi, 4 ; Перейдем к следующему элементу

loop L1

**pop ecx ; Восстановление регистров из стека**

**pop esi ; Обратите внимание, что восстановление регистров**

**; необходимо выполнять в ОБРАТНОМ порядке по отношению**

**: к тому, как они были помещены в стек**

ret

ArraySum ENDP

При использовании отладчика, входящего в пакет Microsoft Visual Studio, вы можете просмотреть машинные команды, автоматически сгенерированные компилятором MASM при обработке сложных операторов и директив. Для этого из меню **View** выберите команду **Debug Windows**, а затем – пункт **Disassembly**. В результате откроется окно дизассемблера, в котором мы сможете увидеть исходный код своей программы, а также скрытые машинные команды, автоматически сгенерированные компилятором.

Существует одно важное исключение из сформулированного нами выше правила о сохранении в стеке модифицируемых в процедуре регистров. Оно относится к тем регистрам, в которых в вызвавшую процедуру возвращаются значения. Очевидно, что в подобных случаях нам незачем сохранять, а затем восстанавливать значения таких регистров. Например, если бы в процедуре ArraySum мы бы сохранили и затем восстановили значение модифицируемого регистра ЕАХ, то возвращаемое процедурой в этом регистре значение суммы было бы потеряно.

# Локальные переменные

Локальными называются переменные, которые создаются, используются и аннулируются в пределах одной процедуры.

При рассмотрении примеров ранее мы объявляли все переменные в сегменте данных. Такие переменные называются статическими глобальными. Термин статический говорит о том, что такая переменная существует все время, пока выполняется программа, в которой она объявлена. Другими словами, время жизни статической переменной совпадает со временем выполнения программы. Термин глобальный определяет область действия переменной. Глобальную переменную можно использовать во всех процедурах, находящихся в текущем исходном файле.

Локальные переменные имеют следующие важные отличия от глобальных переменных:

* ограниченная область действия локальной переменной позволяет быстрее выявить ошибку на этапе отладки, поскольку изменить ее значение может только ограниченное количество команд программы;
* применение локальных переменных позволяет более эффективно расходовать память компьютера, поскольку занимаемый ими участок оперативной памяти можно освободить и перераспределить для других переменных;
* одно и то же имя переменной может использоваться в нескольких процедурах, при этом не возникает конфликта имен.

Локальные переменные всегда создаются в области программного стека, поэтому им нельзя присвоить начальные значения еще на этапе компиляции программы. Локальные переменные можно инициализировать только во время выполнения программы.

## Директива LOCAL

Директива LOCAL предназначена для объявления одной или нескольких локальных переменных внутри процедуры. В исходном коде она должна располагаться сразу за директивой PROC. Синтаксис директивы LOCAL следующий:

LOCAL список\_переменных

Здесь под списком переменных понимается перечень описаний переменных, разделенных запятой, который может занимать несколько строк. Описание каждой переменной задается в следующем виде:

имя: ТИП

В качестве имени переменной можно выбрать любой допустимый в языке ассемблера идентификатор. В качестве типа можно задать один из встроенных типов языка ассемблера, таких как WORD, DWORD, либо один из нестандартных типов, определенных программистом.

Например, в процедуре MySub создается одна локальная переменная var1, которая имеет тип BYTE:

MySub PROC

LOCAL var1: BYTE

В процедуре BubbleSort создаются две локальные переменные. Одна из них имеет имя temp и занимает двойное слово, а другая называется SwapFlag и занимает в памяти один байт:

BubbleSort PROC

LOCAL temp: DWORD, SwapFlag: BYTE

В процедуре Merge создается одна локальная переменная рАггау, в которой будет храниться указатель на слово, расположенное в памяти:

Merge PROC

LOCAL pArray: PTR WORD

Переменная TempArray является массивом из десяти двойных слов. Обратите внимание, что размер массива указывается в квадратных скобках:

LOCAL TempArray[10]: DWORD

Рассмотрим подробнее, что происходит при создании локальных переменных, и какой код на самом деле сгенерирует компилятор ассемблера при использовании в программе локальных переменных. Приведенный ниже фрагмент программы содержит определение процедуры с локальными переменными:

BubbleSort PROC

LOCAL temp: DWORD, SwapFlag: DWORD

ret

BubbleSort ENDP

Вот что получится после ассемблирования:

BubbleSort PROC

**push ebp ; Создание кадра стека для процедуры**

**mov ebp, esp**

**add esp, 0FFFFFFF8h ; Вычитание из регистра ESP числа 8**

**; 8 – размер локальных данных**

**mov esp, ebp ; Уничтожение кадра стека**

**pop ebp**

ret

BubbleSort ENDP

Команда ADD прибавляет число -8 к регистру ESP. В результате указатель стека смещается на 8 байтов вниз, что создает пространство для размещения в области стека двух локальных переменных. Адрес начала области локальных переменных процедуры загружается в регистр ЕВР, через который всегда выполняется доступ к локальным переменным.

Если вы планируете создавать в своей программе локальные переменные, являющиеся массивами переменной длины, необходимо позаботиться о том, чтобы при загрузке программы операционная система выделила достаточное количество памяти под стек. Для этого необходимо указать достаточно большое число в директиве .STACK. Ранее рассмотренные программы использовали стек размером в 4 килобайта, что явно недостаточно для программ, активно использующих локальные данные. Для таких программ рекомендуется использовать стек размером не менее 1 мегабайта.

## Стековые параметры

Существует два основных способа передачи параметров в процедуры – через регистры и через стек. Регистровые параметры используются при выполнении оптимизации скорости работы программы. Однако часто это приводит к излишнему загромождению кода вызывающей программы. Кроме того, обычно при загрузке в регистры значений аргументов приходится сохранять в стеке их текущее состояние.

Альтернативой регистровым являются стековые параметры. При этом перед вызовом процедуры нужные параметры сначала нужно поместить в стек.

Для удобства программиста в компиляторе MASM предусмотрена специальная директива INVOKE, которая позволяет с помощью одного оператора поместить в стек аргументы и вызвать процедуру. Кроме того, есть еще одна причина, по которой вам нужно освоить стековый способ передачи параметров: он используется практически во всех компиляторах языков высокого уровня. Поэтому, если вы хотите, например, вызвать одну из библиотечных функций системы Windows, вы должны передать ей аргументы через стек.

## Директива INVOKE

Директива INVOKE является очень гибким средством вызова процедур и, по сути, позволяет писать программы на языке ассемблера без явного использования команды CALL. Она позволяет передать в процедуру несколько аргументов. Синтаксис директивы INVOKE приведен ниже:

INVOKE имя\_процедуры [, список\_аргументов]

Аргументы, передаваемые процедуре в директиве INVOKE, перечисляются через запятую и могут отсутствовать вовсе. В директиве INVOKE можно указать произвольное число аргументов, причем их список может занимать несколько строчек исходного кода.

## Директива PROC

Директива PROC предназначена для описания имени процедуры и списка передаваемых ей параметров. Ее упрощенный синтаксис показан ниже:

имя\_процедуры PROC, параметр\_1, параметр\_2, ...

Синтаксис описания одного параметра процедуры выглядит так:

имя\_параметра: ТИП

имя\_параметра выбирается произвольно программистом и должно удовлетворять соглашению, принятому в языке ассемблера для имен меток. Область действия данного параметра ограничена текущей процедурой, поэтому она называется локальной. Это позволяет иметь одинаковые имена параметров в разных процедурах, однако учтите, что они не должны совпадать с именами глобальных переменных или меток кода. Параметр может иметь один из перечисленных ниже типов: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD или TBYTE. Кроме того, параметр может являться указателем на переменную одного из стандартных типов. В этом случае перед именем типа используется модификатор PTR.

Приведенной ниже процедуре в качестве параметров передаются два двойных слова:

AddTwo PROC, a1: DWORD, a2: DWORD

mov eax, [a1] ; Загружаем в регистр EAX значение первого аргумента

add eax, [a2] ; Прибавляем значение второго аргумента

ret ; Возврат из процедуры, сумма будет передана в EAX

AddTwo ENDP

Выполнить вызов процедуры AddTwo можно при помощи директивы INVOKE:

INVOKE AddTwo, 2, 3

## Процедура сравнения строк

Приведенная в листинге 6.1 программа определяет процедуру strCompare, выполняющую сравнение двух строк s1 и s2, передаваемых ей в виде параметров.

Сравнение строк выполняется байт за байтом, при этом используются соответствующие символам 8-разрядные ASCII-коды. Операция сравнения учитывает регистр используемых символов, поскольку в ASCII-таблице для прописных и строчных букв предусмотрены разные коды. Результат сравнения возвращается в регистре EAX.

|  |
| --- |
| **Листинг 6.1** |
| TITLE Процедура сравнения строк  .686  .MODEL FLAT, C  .STACK 4096  ; импортируемые функции из стандартной библиотеки языка C  printf PROTO format: PTR BYTE, args: VARARG  gets PROTO s: PTR BYTE  ; экспортируемые функции  PUBLIC main  .CONST  sLess BYTE "str1 < str2", 0Ah, 0  sEqu BYTE "str1 == str2", 0Ah, 0  sGreat BYTE "str1 > str2", 0Ah, 0  .DATA?  MaxStr = 256  str1 BYTE MaxStr DUP(?) ; первая строка  str2 BYTE MaxStr DUP(?) ; вторая строка  .CODE  ; strCompare  ; Сравнение строк  ;  ; param s1 Первая строка  ; param s2 Вторая строка  ; param n Максимальная длина сравниваемых строк  ; return в EAX результат сравнения,  ; < 0, если s1 < s2  ; = 0, если s1 == s2  ; > 0, если s1 > s2  strCompare PROC USES ecx esi edi, s1: PTR BYTE, s2: PTR BYTE, n: DWORD    xor eax, eax ; сбросим EAX в 0  mov ecx, [n] ; максимальная ожидаемая длина строки  jecxz @exit ; если передана нулевая длина, то выходим  mov esi, [s1] ; адрес первой строки  mov edi, [s2] ; адрес второй строки  cld ; сбрасываем флаг направления  repe cmpsb  jz @exit ; если строки равны, то выход  ; в EAX уже записан 0  mov al, [esi - 1] ; загрузим в AL последний проверенный символ  sub al, [edi - 1] ; и определим знак - какой из символов меньше  movsx eax, al ; расширяем результат до 32 бит  @exit:  ret    strCompare ENDP  ; main  ; Точка входа в программу  main PROC  ; попросим пользователя ввести строки  INVOKE gets, ADDR str1  INVOKE gets, ADDR str2  ; выполняем сравнение строк  INVOKE strCompare, ADDR str1, ADDR str2, MaxStr  ; выведем результат  cmp eax, 0  jg @great  je @equal    INVOKE printf, ADDR sLess  jmp @exit  @equal:  INVOKE printf, ADDR sEqu  jmp @exit  @great:  INVOKE printf, ADDR sGreat  @exit:  mov eax, 0  ret  main ENDP  END |

# Структуры

Структурой (structure) называется упорядоченная группа логически связанных между собой переменных. Переменные, входящие в структуру, называются полями (fields). В программе можно работать со структурой как с единым элементом (например, передавать ее адрес в процедуру), либо обращаться к отдельным ее полям (например, устанавливать или считывать значение поля).

В языках программирования высокого уровня структуры используются уже очень давно, практически с самого момента их появления. Ценность структур состоит в том, что они позволяют легко передать в процедуру большое количество разнородных данных. Предположим, например, что мы должны передать в процедуру, которая обслуживает драйвер жесткого диска, двадцать различных входных параметров. Очевидно, что передавать все параметры в определенном порядке при вызове такой процедуры весьма непрактично. Лучше объединить их в один блок, который и называется структурой, и передать в процедуру адрес структуры. При этом из стека для размещения параметров выделяется совсем немного памяти (всего одно двойное слово для адреса структуры). Кроме того, при таком подходе процедура при необходимости может внести изменения в поля структуры.

Следует отметить, что структуры в языке ассемблера практически аналогичны структурам, используемым в языках высокого уровня, например в С.

Давайте рассмотрим простой пример – структуру POINT2, которая может использоваться для представления координат элемента X и Y на экране. Поле структуры под названием X, располагается со смещением 0 относительно начала структуры, а поле Y – со смещением 4, как показано ниже:

POINT2 STRUCT

X SDWORD ? ; Смещение +00h

Y SDWORD ? ; Смещение +04h

POINT2 ENDS

Для использования структуры в программе необходимо:

* определить структуру;
* объявить переменные типа структуры, которые называются структурными переменными (structure variables);
* использовать команды, которые используют структурные переменные, или к поля структур.

## Определение структуры

Структура определяется с помощью директив STRUCT и ENDS. Внутри структуры ее поля определяются точно так же, как и обычные переменные в программе. Общий синтаксис определения структуры следующий:

имя\_структуры STRUCT

объявление\_полей

имя\_структуры ENDS

Количество полей в структуре практически неограничено.

Инициализаторы поля. При объявлении полей структуры для них можно указать один из инициализаторов, который определяет их значение по умолчанию. Возможные типы инициализаторов перечислены ниже.

* Неопределенное значение. Если значение поля заранее не определяется, то структурная переменная описывается с помощью спецификатора ?.
* Строковое значение. Поле структуры может содержать строку символов, которая заключается в кавычки.
* Целочисленное значение. Чтобы присвоить полю структуры целочисленное значение, воспользуйтесь целочисленной константой или выражением.
* Массивы. Если поле структуры является массивом, то для его инициализации используется оператор DUP.

В качестве примера давайте рассмотрим определение структуры Student, которая описывает информацию о студенте и содержит такие поля: идентификатор студента, фамилию, дату поступление в университет (год) и оценки по программированию за четыре семестра. Ниже приведено определение структуры, которое должно находиться в программе:

Student STRUCT

IdNum BYTE "000000000"

Name BYTE 50 DUP(0)

Year WORD 0

ProgMarks BYTE 0, 0, 0, 0

Employee ENDS

## Объявление структурных переменных

После определения структуры в программе нужно создать один или несколько ее экземпляров, которые называются структурными переменными, и при необходимости присвоить им начальные значения. Если при определении структурной переменной используются пустые угловые скобки, то ассемблер присваивает ее полям значения, заданные по умолчанию с помощью инициализаторов во время определения самой структуры. Чтобы задать элементам структуры новые исходные значения, укажите их в угловых скобках. Ниже приведены примеры определения экземпляров структур POINT2 и Student, в которых применена как явная, так и неявная инициализация их элементов:

.DATA

pt1 POINT2 < 5, 10 >

pt2 POINT2 <>

john Student <>

Существует также возможность при создании структурной переменной переопределить стандартное значение некоторых или всех ее полей. В приведенном ниже примере переопределяется поле IdNum структурной переменной jack, имеющей тип Student:

jack Student < "111222333" >

Если значение инициализатора для строкового поля короче самого поля, оставшиеся позиции заполняются пробелами. Важно отметить, что при создании структурной переменной, в конце ее строковых полей компилятор автоматически не помещает нулевой байт. Поэтому, если вы планируете вызывать в программе библиотечные функции обработки строк, не забудьте вручную разместить в конце строки нулевой байт.

Если при инициализации структурной переменной нужно пропустить несколько полей, укажите вместо них запятую. Например, в приведенном ниже фрагменте кода инициализация поля IdNum пропускается, а полю LastName присваивается значение "Иванов":

s1 Student < , "Иванов" >

Если поле структурной переменной является массивом, то для инициализации части или всех элементов массива используется оператор DUP. Если значение инициализатора короче размера поля структуры, оставшиеся его позиции заполняются нулями. Ниже в качестве примера мы проинициализировали первые два элемента поля ProgMarks, а оставшиеся два элемента заполнили нулями:

s1 Student < , "Иванов", , 2 DUP(5) >

Можно создать массив структур, то есть такой массив, каждый элемент которого является структурой. В приведенном ниже фрагменте кода каждому элементу массива AllPoints присваивается значение <0, 0>:

NumPoints = 3

AllPoints POINT2 NumPoints DUP(< 0, 0 >)

## Обращение к структурным переменным

При обращения к структурной переменной и ее отдельным полям часто используются операторы TYPE и SIZEOF. Эти операторы вычисляют размер структурной переменной в байтах. Например, в следующем примере значения, вычисленные операторами TYPE и SIZEOF, будут равны. Эти константные выражения вычисляются на этапе компиляции.

TYPE Student

SIZEOF Student

При непосредственном обращении к отдельным полям структуры в качестве префикса должно быть указано имя структурной переменной или самой структуры.

TYPE Student.ProgMarks

LENGTHOF Student.ProgMarks

SIZEOF Student.ProgMarks

Ниже показан пример обращения к полям структурной переменной worker на этапе выполнения программы:

.DATA

s1 Student <>

.CODE

mov dx, [s1.Year] ; Загрузим в регистр DX год

mov [s1.ProgMarks], 5 ; Сохраним оценку за первый семестр

mov [s1.ProgMarks + 1], 5 ; Сохраним оценку за второй семестр

mov esi, OFFSET s1.LastName ; Загрузим в esi адрес строки,

; содержащей имя студента

Косвенную адресацию удобно применять для обращения к полям структурной переменной, если в один из регистров общего назначения (например в ESI) загрузить ее адрес. Благодаря ей можно без проблем передать адрес структурной переменной в процедуру или обрабатывать элементы массива структур. При косвенном обращении к полям структурной переменной используется оператор PTR, как показано ниже:

mov esi, OFFSET s1 ; Помещаем в ESI адрес структурной переменной

mov ax, (Student PTR [esi]).Year ; Загрузим в регистр AX год

При обработке элементов массива структур в цикле обычно используется косвенная или базово-индексная адресация. В приведенном ниже примере выполняется обращение элементу массива AllPoints, индекс которого хранится в регистре EDI.

mov (POINT2 PTR AllPoints[edi]).X, ax ; Устанавливаем координату X

mov (POINT2 PTR AllPoints[edi]).Y, ax ; Устанавливаем координату Y

## Вычисление площади прямоугольника

Программа из листинга 6.2 выполняет вычисление площади прямоугольника со сторонами, параллельными осям координат. Для хранения прямоугольника программа использует структуры Point2 и Rect2. Вычисление площади выполняется процедурой area, принимающей в качестве параметра указатель на прямоугольник, и возвращающей вычисленное значение площади в регистре EAX.

|  |
| --- |
| **Листинг 6.2** |
| TITLE Площадь прямоугольника  .686  .MODEL FLAT, C  .STACK 4096  ; импортируемые функции из стандартной библиотеки языка C  printf PROTO format: PTR BYTE, args: VARARG  scanf PROTO format: PTR BYTE, args: VARARG  ; экспортируемые функции  PUBLIC main  ; определяемые типы данных  ; точка на плоскости  Point2 STRUC  X SDWORD 0  Y SDWORD 0  Point2 ENDS  ; прямоугольник на плоскости  Rect2 STRUC  lftp Point2 <> ; верхний левый угол  btrt Point2 <> ; правый нижний угол  Rect2 ENDS  .CONST  sInput BYTE "%d %d %d %d", 0  sArea BYTE "area is %d", 0Ah, 0  .DATA  r1 Rect2 <>  .CODE  ; area  ; Вычисление площади прямоугольника  ;  ; param r Указатель на структуру Rect2  ; return в EAX площадь прямоугольника  area PROC USES edx esi, r: PTR Rect2  mov esi, [r] ; загрузим адрес структуры в ESI  mov eax, (Rect2 PTR [esi]).btrt.X ; вычисляем длину одной стороны  sub eax, (Rect2 PTR [esi]).lftp.X  mov edx, (Rect2 PTR [esi]).btrt.Y ; вычисляем длину другой стороны  sub edx, (Rect2 PTR [esi]).lftp.Y  imul edx ; перемножаем полученные длины сторон  ret    area ENDP  ; main  ; Точка входа в программу  main PROC  ; попросим пользователя ввести данные (x1, y1, x2, y2)  INVOKE scanf, ADDR sInput, ADDR r1.lftp.X, ADDR r1.lftp.Y,  ADDR r1.btrt.X, ADDR r1.btrt.Y  ; вычисляем площадь прямоугольника  INVOKE area, ADDR r1  ; выведем результат  INVOKE printf, ADDR sArea, eax  mov eax, 0  ret  main ENDP  END |

# Задания к работе

1. Разработайте процедуры, выполняющие операции со строками: вычисление длины строки, копирование одной строки в другую, добавление одной строки в конец другой. Напишите демонстрационную программу для разработанных процедур.
2. Разработайте процедуры, выполняющие сложение и умножение матриц четвертого порядка. Напишите демонстрационную программу для разработанных процедур.
3. Разработайте процедуру, выполняющую вычисление периметра произвольного прямоугольника. Для хранения вершин прямоугольника используйте массив структур Point2.
4. (🖉) Напишите на языке C функцию, выполняющую обмен двух целочисленных переменных, переданных ей в виде аргументов. Такая функция должна иметь прототип void swap(int \*, int \*). Используя окно дизассемблера, исследуйте ассемблерный код этой функции. Где хранятся аргументы функции и как осуществляется к ним доступ? Где хранятся локальные переменные функции и как осуществляется к ним доступ? Как выглядит пролог и эпилог этой функции? Сравните ассемблерный код, полученный для пролога и эпилога функции, написанной на языке C с пологом и эпилогом функции, написанной на языке ассемблера с использованием директив PROC со списком параметров, USES и LOCAL.
5. (★) Разработайте процедуру, выполняющую сортировку массива целых чисел при помощи алгоритма быстрой сортировки.