От Ассемблера к C++

18 февраля 2026 г.

largestvalueindex.s

```
.global _start
.section .data
# Number of elements
numberofnumbers:
        .quad 7
# The data elements themselves
mynumbers:
        .quad 5, 20, 33, 80, 52, 10, 1
# This program will find the larges value in the array
.section .text
_start:
        ### Initialize Registers ###
        # Put the number of elements of array in %rcx
        movq numberofnumbers, %rcx
```

largestvalueindex.s

```
# Put the index of the first element in %rbx
movq $0, %rbx

# Use %rdi to hold the current-high value
movq $0, %rdi

### Check preconditions ###

# If there are no numbers, stop
cmp $0, %rcx
je endloop

### Main loop ###
```

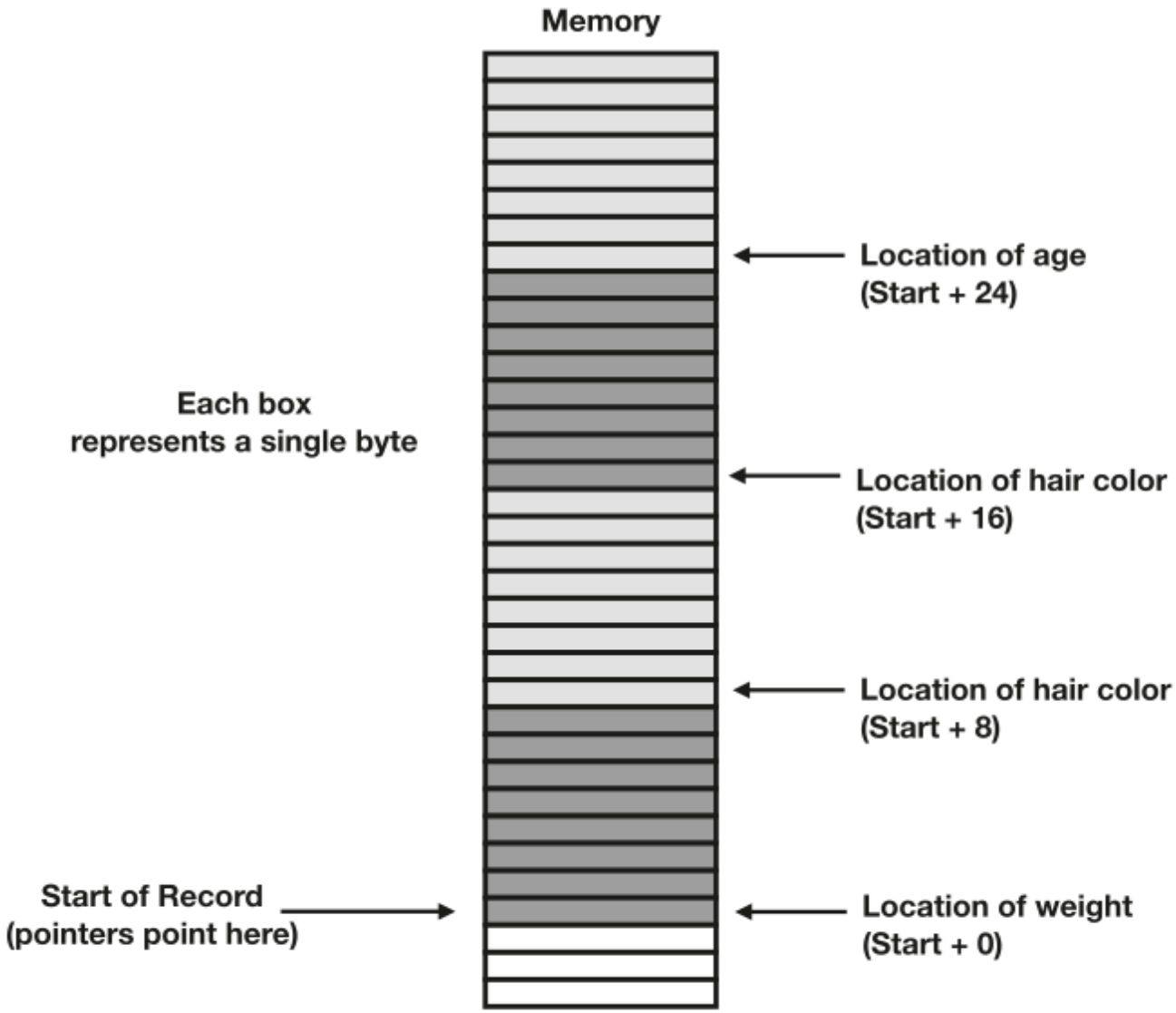
largestvalueindex.s

```
myloop:
        # Get the next value of mynumbers indexed by %
        movq mynabers(, %rbx, 8), %rax

        # If it is not bigger, go to the end of the loo
        cmp %rdi, %rax
        jbe loopcontrol
        # Otherwise, store this as the biggest elemtn
        movq %rax, %rdi

loopcontrol:
        # Move %rbx to the next index
        incq %rbx

        # Decrement %rcx and keep going until %rcx is
        loopq myloop
```

largestvalueindex.s

```
        ### Cleanup and Exit ###
endloop:
        # We are done - exit

        movq $60, %rax
        syscall
```

Структуры. Размещение данных о персоне в памяти



Структуры. Директива .equ

```
.global WEIGHT_OFFSET, HAIR_OFFSET, HEIGHT_OFFSET,
        AGE_OFFSET
.equ WEIGHT_OFFSET, 0
.equ HAIR_OFFSET, 8
.equ HEIGHT_OFFSET, 16
.equ AGE_OFFSET, 24
```

Структуры

```
movq  AGE_OFFSET(%rbx), %rax
```

Структуры. Разделение программы. persondata.s

```
.section .data
.globl people, numpeople
numpeople:
        # Calculate the number of people in array
        .quad (endpeople - people)/PERSON_RECORD_SIZE
people:
        # Array of people
        .quad 200, 2, 74, 20
        .quad 280, 2, 72, 44
        .quad 150, 1, 68, 30
endpeople: # Marks the end of the struct
.globl WEIGHT_OFFSET, HAIR_OFFSET, HEIGHT_OFFSET, AGE_
.equ WEIGHT_OFFSET, 0
.equ HAIR_OFFSET, 8
.equ HEIGHT_OFFSET, 16
.equ AGE_OFFSET, 24
```

Структуры. Разделение программы. persondata.s

```
        # Total size of the sturct
.global PERSON_REOCRD_SIZE
.equ PERSON_RECORD_SIZE, 32
```

```
as persondata.s -o persondata.o
```

Разделение программы. tallest.s

```
.global _start
.section .text
_start:
        leaq people, %rbx
        movq numpeople, %rcx
        # Tallest value found
        movq $0, %rdi

        ### Check precondition ###
        # If there are no records, finish
        cmp $0, %rcx
        je finish
```

Разделение программы. tallest.s

```
mainloop:
        # %rbx pointer to the whole struct
        # This instruction grabs the height field
        # and stores it in %rax
        movq HEIGHT_OFFSET(%rbx), %rax

        # If it is less or equal to our current
        # tallest, go to the next one.
        cmp %rdi, %rax
        jbe endloop
        # Copy this value as the tallest value
        movq %rax, %rdi
```

Разделение программы. tallest.s

```
endloop:
        addq $PERSON_RECORD_SIZE, %rbx
        loopq mainloop

finish:
        movq $60, %rax
        syscall
```

Разделение программы.

```
as tallest.s -o tallest.o
ld persondata.o tallest.o -o tallest
```

Сохранение текста.

```
.global _start
.section .data
mytext:
        .ascii "This is a string of characters.\0"
.section .text
_start:
        ### Initialization ###
        # Move a pointer to the string into %rbx
        movq $mytext, %rbx
        # Count starts at zero
        movq $0, %rdi
```

Сохранение текста.

```
mainloop:
        # Get the next byte
        movb (%rbx), %al
        # Quit if we hit the null terminator
        cmpb $0, %al
        je finish
        # Go to the next byte if the value is not betw
        cmp 'a', %al
        jb loopcontrol

        cmp $'z', %al
        ja loopcontrol

        # It is lower-case Add one to %rdi
        incq %rdi
```

Сохранение текста.

```
loopcontrol:
        incq %rbx
        jmp mainloop

finish:
        movq $60, %rax
        syscall
```

Endianness. x86-64 little endian. big-endian.

10000000110000001110000011110000111110001111111001111111011111111

1. 11111111
2. 11111110
3. 11111100
4. 11111000
5. 11110000
6. 11100000
7. 11000000
8. 10000000

Writing output.

```
.global _start
.section .data
mystring:
        .ascii "Hello there\n"
mystring_end:
        .equ mystring_length, mystring_end - mystring
.section .text
_start:
```

Writing output.

```
### Display the string
# System call number
movq $1, %rax
# File descriptor
movq $1, %rdi
# Pointer to the data
movq $mystring, %rsi
# Length of the data
movq $mystring_length, %rdx
syscall

# Exit
movq $0x3c, %rax
movq $0, %rdi
syscall
```

Стек

Операционная система выделяет пространство для стека, а затем помещает указатель на эту память в указатель стека %rsp. %rsp указывает на конец памяти стека.

Стек

Вы можете добавлять элементы в стек, используя семейство инструкций push. Когда вы добавляете элемент в стек, происходит две вещи:

1. Это уменьшает значение %rsp, чтобы указать на следующее место в стеке.
2. Эта команда копирует значение в место, указанное параметром %rsp.

Стек. factorialstack.s

```
.global _start
.section .data
value:
        .quad 5
.section .text
_start:
        # Push in the sentinel value
        pushq $0
        # Grab the value
        movq value, %rax
        # Push all the vlaues from 1
        # to the current value to the stack
```

Стек. factorialstack.s

```
pushvalues:
        pushq %rax
        decq %rax
        jnz pushvalues

        # Prepare for mutiplying
        movq $1, %rax

multiply:
        # Get the next value from the stack
        popq %rcx
        # If it is the sentinel, we are done
        cmpq $0, %rcx
        je complete
```

Стек. factorialstack.s

```
        # multiply by what we have accumulated so far
        mulq %rcx

        # Do it again
        jmp multiply

complete:
        movq %rax, %rdi
        movq $60, %rax
        syscall
```

Функции

Имя: В функции имя также используется в качестве метки для начального адреса кода функции (точки входа в функцию).

Функции

Входные параметры: Функции имеют входные параметры, которые представляют собой то, что функция использует для обработки. Например, функция вычисления факториала принимает в качестве входного параметра значение, от которого вы хотите вычислить факториал. Входные параметры также называются **аргументами**.

Функции

Возвращаемое значение: Функции имеют возвращаемое значение, которое представляет собой значение, передаваемое обратно коду, вызвавшему функцию. Например, в функции вычисления факториала возвращаемое значение — это конечный результат вычисления факториала. Во многих языках программирования допускается только одно возвращаемое значение функции. Однако функция может имитировать наличие нескольких возвращаемых значений, передавая в качестве входных параметров указатели на память, где будут храниться другие результаты. Другой вариант — возвращаемое значение может быть указателем на область памяти, содержащую несколько значений.

Функции

Побочные эффекты: Побочный эффект — это изменение чего-либо, что не указано во входных или выходных параметрах. В целом, программисты стараются избегать побочных эффектов, но это не всегда возможно. Например, запись ошибки в лог — это побочный эффект; обычно лог не является параметром функции, поэтому отправка данных в файл лога является побочным эффектом.

Соглашение о вызове функции. ABI

Подобно тому, как существует соглашение для системных вызовов, существует и предопределенное соглашение для вызовов функций. Такой тип соглашения известен как двоичный интерфейс приложения(application binary interface), или **ABI**.

Отправка входных параметров

Параметры поступают в функцию преимущественно в регистрах. Параметры идентифицируются по положению, и позиции соответствуют регистрам следующим образом:

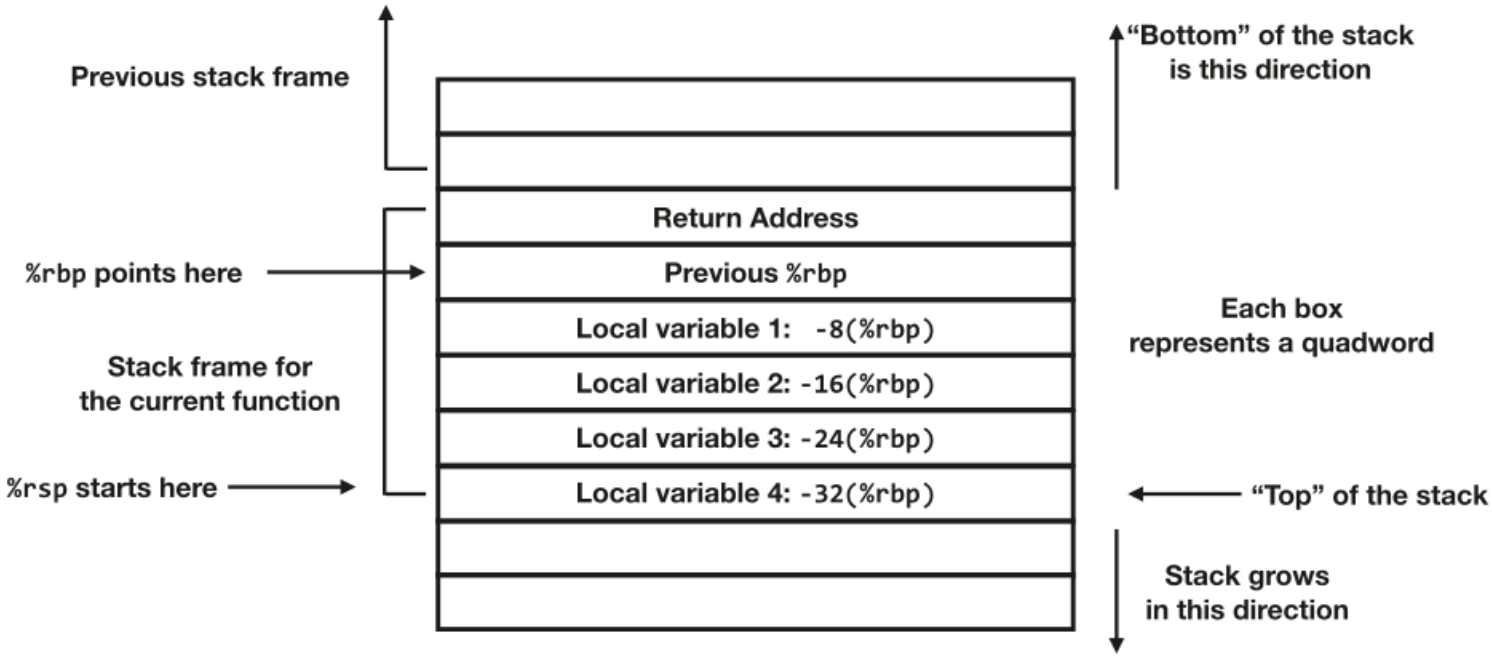
1. %rdi
2. %rsi
3. %rdx
4. %rcx
5. %r8
6. %r9

Таким образом, если параметр всего один, он передается в %rdi. Если параметров два, то в %rdi передается первый параметр, а в %rsi — второй.

Возвращение значения функции

Возвращаемое значение помещается в регистр %rax.

Организация стекового фрейма



%rbp хранит адрес начала фрейма стека. Сначало надо сохранить значение %rbp. Следовательно, первое, что следует сделать, это поместить это значение в стек. Сразу после помещения значения %rbp в стек, значение %rsp следует скопировать в %rbp. Это заставит %rbp указывать на предыдущую версию самого себя.

Стек

Теперь нам нужно освободить место для локальных переменных. Вычтите из %rsp (помните, что стек растет вниз!) столько памяти, сколько вам нужно для локальных переменных. Будем ссылаться на них как смещение относительно %rbp, но вычитание значения из %rsp резервирует пространство, чтобы любые вызовы текущей функции не привели к перезаписыванию ее локальных переменных.

Стек

Практически каждая функция начинается с этих инструкций для работы со стеком, где NUMBYTES — это количество байтов локального пространства памяти, необходимого для ее работы:

```
# Save the pointer to the previous stack frame
pushq %rbp
# Copy the stack pointer to the base pointer for a
# fixed reference point
movq %rsp, %rbp
# Reserve however much memory on the stack I need
subq $NUMBYTES, %rsp
```

Стек

Затем, в конце функции, эти шаги следует выполнить в обратном порядке:

```
# Restore the stack pointer
movq %rbp, %rsp
# Restore the base pointer
popq %rbp
```

enter, leave

Инструкция enter просто принимает значение, представляющее собой объем дополнительной памяти, которую вы хотите разместить в стеке, и выполняет все три шага за вас.

```
enter $NUMBYTES, $0
```

Аналогичным образом, код для завершения обработки стекового кадра можно просто заменить leave:

```
leave
```

call

```
pushq $next_instruction_address
jmp thefunction
next_instruction_address:
# Next instruction here
```

приведенный выше код можно заменить следующим:

```
call thefunction
```

ret

Аналогично, извлечение адреса возврата из стека и последующий переход к нему осуществляется с помощью инструкции ret. Это делается в самом конце функции с помощью следующего простого кода:

```
ret
```

exponentfunc.s

```
.globl exponent
.type exponent, @function

.section .text
exponent:
        # %rdi has the base
        # %4si has th exponent

# Create the stack frame one 8-byte local variable
# which will be referred to using -8(%rbp).
# This will store the current value of the exponent
# as we iterate through it.
# We are allocating 16 bytes so that we maintain
# 16-byte alignment
        enter $16, $0
```

exponentfunc.s

```
        # Accumulate value in %rax
        movq $1, %rax

        # Store the exponent
        movq %rsi, -8(%rbp)

mainloop:
        mulq %rdi
        decq -8(%rbp)
        jnz mainloop

complete:
        # Result is already in %rax
        leave
        ret
```

runexponent.s

```
.global _start
.section .text
_start:
        # Call exponent with 3 and 2
        movq $3, %rdi
        movq $2, %rsi
        call exponent

        # result is now in %rax
        movq %rax, %rdi
        movq $60, %rax
        syscall
```

Функция

```
as exponentfunc.s -o exponentfunc.o
as runexponent.s -o runexponent.o
ld exponentfunc.o runexponent.o -o runexponent
```

Вызов функции из другого языка

```
int exponent(int, int);
int main()
{
        return exponent(4, 2);
}
```

Функция

```
gcc runexponent.c exponentfunc.s -o runexponent
```