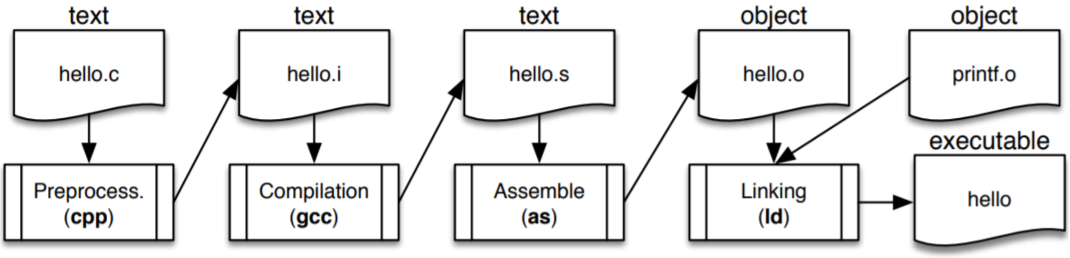
Modules and Makefiles



- source code → hello.**c**
  output/relocatable → hello.**o**
  executable → hello

- #ifndef //if it is not defined, it executes
  #define
  //function declaration
  #endif

Data sizes

Integer types

Type	Storage size	Value range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int	4 bytes (IA-32)	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes (IA-32)	0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Floating-point types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Makefile simple structure

{exec file name}: dependency1 dependency2…
/*hello: hello.o main.o*/

{dependency1}: dependencies
/*hello.o: hello.h hello.c*/

{dependency2}: dependencies

```
    /*main.o: hello.h main.c*/

{additional rules}
    /*run
        ./{exec file name}*/ → (command) make run
```

Makefile complex structure

```
INCLUDES = hello.h
SOURCES = hello.c main.c
OBJFILES = hello.o main.o
EXEC = hello

.SUFFIXES: .c .o

.c.o:
	gcc -Wall -g -c $<

${EXEC}: ${OBJFILES}
	gcc -Wall -g -o ${EXEC} ${OBJFILES}

${OBJFILES}: ${SOURCES} ${INCLUDES}

run: ${EXEC}
	./${EXEC}

clean:
	rm -f ${OBJFILES} ${EXEC}
```

Debug

valgrind

gdb

```
(gdb) -tui ./file
(gdb) b file
(gdb) run
(gbd) display <var>
(gdb) n //next function
(gdb) s //single step
(gdb) print <var>
```

- strip {file} //reduces size by removing debug information

Bit-level operations

Operations &, |, ~, ^ (C) AND, OR, NOT, XOR (Assembly)

Apply to any “integral” data type → long, int, short, char…
Nota: NEG (Assembly) = ! (C) changes the sign.

Shift operations <<, >> (C) SHR/SAR, SHL/SAL (Assembly)

Shift left → Multiplication by 2
Shift right → Division by 2

- **Left shift: x << y**
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - **Fill with 0’s on right**

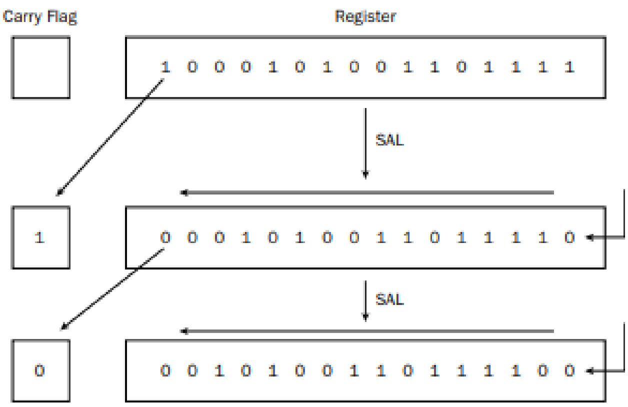
Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

- **Right shift: x >> y**
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - **Fill with 0’s on left**
  - Arithmetic shift
    - **Replicate most significant bit on left**

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

SHL/SAL (the “ << ” operator in C)

- SHL and SAL are equivalent operations
  - They exist for consistency with the right shift; we will see why in a moment
- Shifts bits to the left; zeros enter on the right and the last bit to exit left goes to the carry flag



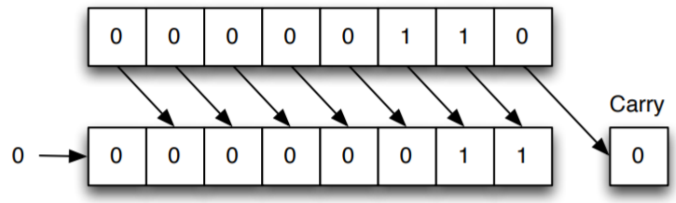
SHL/SAL - Three formats

- SHL *destination* (or SAL *destination*)
  - shifts the *destination* value left one position (equivalent to $destination *= 2$)
- SHL %cl, *destination* (or SAL %cl, *destination*)
  - shifts the *destination* value left by the number of times specified in the CL register (equivalent to $destination *= 2^{CL}$)
- SHL $n, *destination* (or SAL $n, *destination*)
  - shifts the *destination* value left by the number of times specified by a constant value n (equivalent to $destination *= 2^{n}$)

- In all formats, *destination* can be a memory address or a register
- The SHL/SAL instructions can operate on numbers of 8(b), 16(w) or 32(l) bits

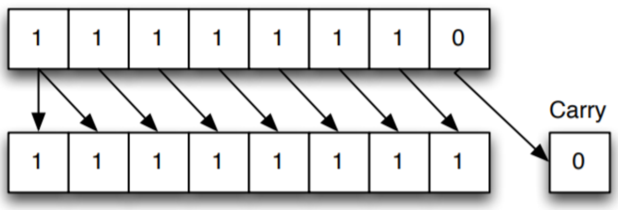
SHR (the " >> " operator in C)

- Logic shift to the right; Therefore, *does not preserve the signal* of the number
- Shifts bits to the right; zeros enter on the left and the last bit to exit to the right goes to the carry flag (similar to the left shift)



SAR (the " >> " operator in C, when applied to signed integers)

- Arithmetic shift to the right; Therefore, *preserves the signal* of the number
- Shifts bits to the right; either clears or sets the bits entered on the left, according to the sign of the integer. The last bit that exits to the right goes to the carry flag

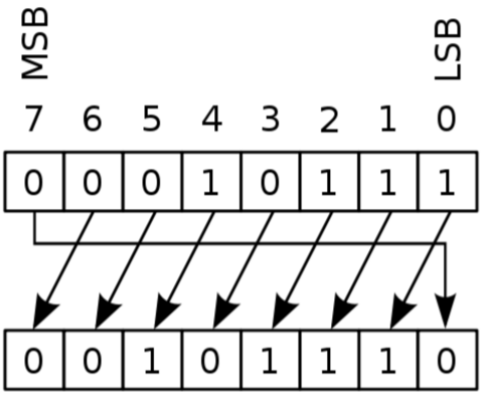


Rotating bits

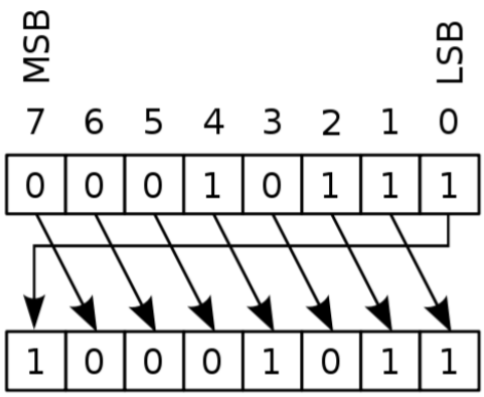
ROL/ROR (no equivalent operation in C)

- ROL - Bit rotation to the left
- ROR - Bit rotation to the right
- Perform just like the shift instructions, except the overflow bits are pushed back into the other end of the value instead of being dropped.

Rotate Left



Rotate Right



ROL/ROR - Three formats (similar to shift instructions):

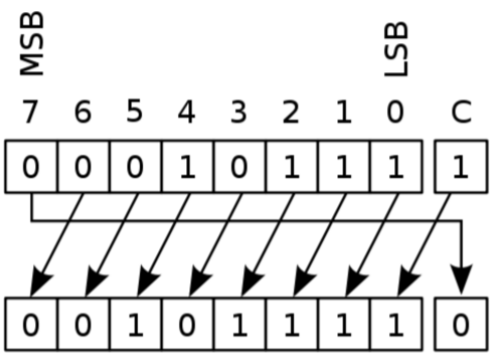
- RO{L/R} *destination*
- RO{L/R} %cl, *destination*
- RO{L/R} $n, *destination*

- In all formats, *destination* can be a memory address or a register
- The ROL/ROR instructions can operate on numbers of 8(b), 16(w) or 32(l) bits

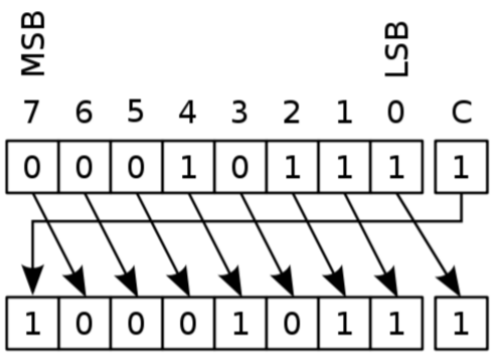
RCL/RCR (no equivalent operation in C)

- RCL - Bit rotation to the left *with carry*
- RCR - Bit rotation to the right *with carry*
- Perform bit rotation, but the first entering bit comes from carry and overflow bits go to carry.

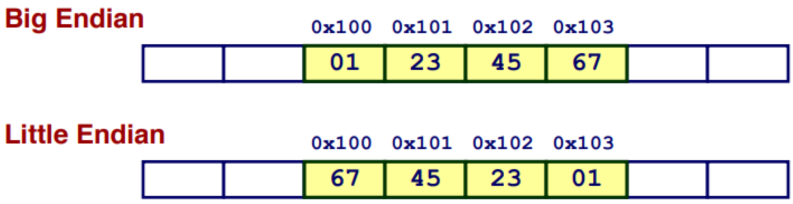
Rotate Left with Carry



Rotate Right with Carry



Byte ordering



Assembly

Special characters

. – starts an assembler directive
– starts a comment
% – starts a register name
$ – starts a value

Sections

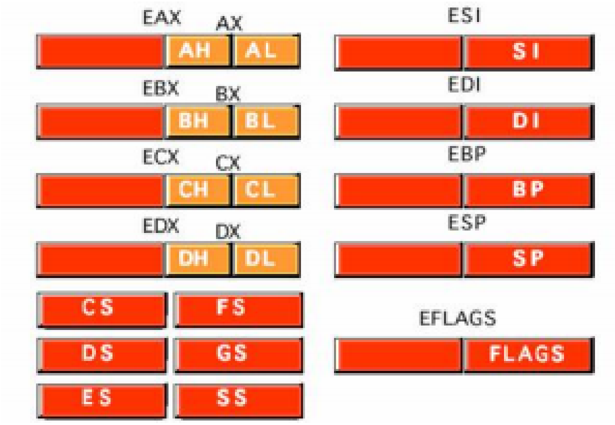
.data - declare initialized variables
.bss - define uninitialized memory areas
    .comm - Declares a global memory area
    .lcomm - Declares a local memory area
.text - assembly instructions

Data types

.octa – 128 bits (16 bytes) integer
.**q**uad – 64 bits (8 bytes) integer
.double – floating point number with double precision (8 bytes)
.**l**ong – 32 bits (4 bytes) integer
.int – 32 bits (4 bytes) integer aka **w**ord
.float – floating point number (4 bytes)
.short – 16 bits (2 bytes) integer
.**b**yte – 8 bits
.ascii – string
.asciz – string automatically terminated by zero
.equ - constant (use with $<var>)

Nota: To avoid memory alignment issues, variables that occupy the most, should be declared first.

Registers



Instructions

ADC (add with carry)

adc origin, destination

The ADC instruction can be used to add two integer values, along with the value contained in the carry flag (set by a previous addition)
Performs the operation: destination = destination + origin + CF

SBB (subtract with carry)

sbb origin, destination

The SBB instruction can be used to subtract two integer values, along with the value contained in the carry flag (set by a previous subtraction)
Performs the operation: destination = destination − (origin + CF)

MOVS (move with sign extend)

movsX origin, destination

Origin can be a memory address or a register (8 or 16 bit)
Destination can be a register of 16(w) or 32(l) bits

```
movb $-1, %al
movsbw %al, %ax
movsbl %al, %eax
movswl %ax, %eax
```

MOV (with offset)

movX offset (register), destination
movX offset (base_address, index, size), destination

Indirect addressing with offset: When we add a constant value (positive or negative) to the address stored in a register. The address is given by base_address+offset+index*size

MUL (unsigned) | IMUL (signed)

Size of origin	*operand2*	*destination*
8 bits	AL	AX
16 bits	AX	DX:AX
32 bits	EAX	EDX:EAX

imul origin
imul origin, destination (destination = destination × origin)
imul multiplier, origin, destination (destination = origin × multiplier → integer constant)

DIV (unsigned) | IDIV (signed)

Size of divisor	*dividend*	*quotient*	*remainder*
8 bits	AX	AL	AH
16 bits	DX:AX	AX	DX
32 bits	EDX:EAX	EAX	EDX

The **cbw** (convert byte to word), **cwd** (convert word to double word) and **cdq** (convert double word to quad word) instructions can be used to produce a correct dividend before a division instruction.

- **cbw** - converts the signed byte in AL to a signed word in AX by copying the sign bit of AL
- **cwd** - converts the signed word in AX to a signed double word in EAX by copying the sign bit of AX
- **cdq** - converts the signed double word in EAX to a signed quad word in EDX:EAX by copying the sign bit of EAX to all bits of EDX

LOOP

`loop` **instruction example**

```
    ...
    movl $100, %ecx
my_loop:
    ...
    loop my_loop
    ...
```

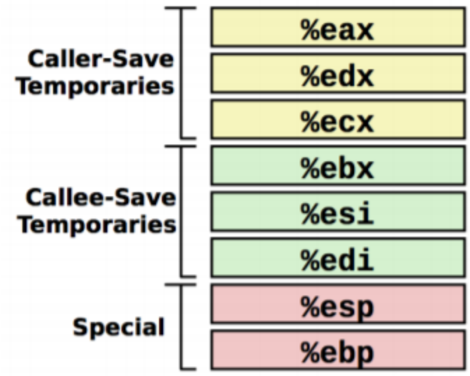
- `loop` automatically decrements `%ecx` and jumps to the label if `%ecx` is different from 0
- `loope/loopz`: decrements `%ecx` and jumps to the label if `%ecx` is different from 0, *and the flag ZF is active*
- `loopne/loopnz`: decrements `%ecx` and jumps to the label if `%ecx` is different from 0, *and the flag ZF is* ***not*** *active*

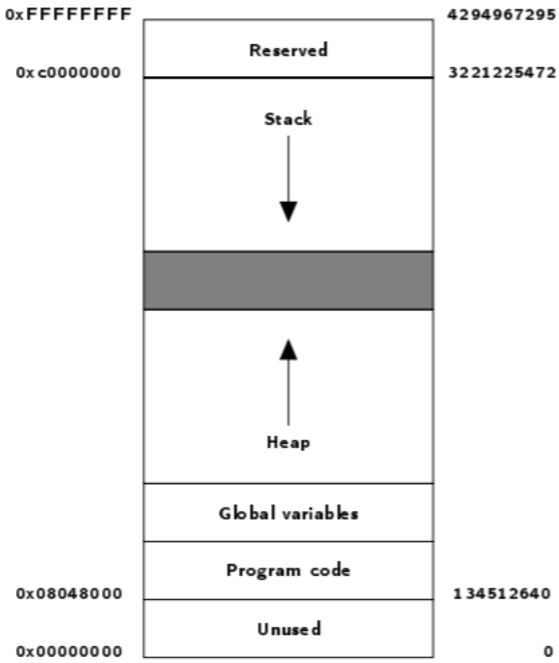
Code Template

```
# the data section allows to declare initialized variables
.section .data # the ".section" can be ommitted

        .equ LINUX_SYS_CALL, 0x80       # the .equ directive defines a
                                        # constant
output_int:
        .asciz "imprimir inteiro:"      #definition of a string

# the bss section is used to define uninitialized memory areas
.section .bss

        .comm buffer, 10000      # global array of 10000 bytes
        .lcomm buffer2, 500      # array of 500 bytes, only visible in
                                 # current module (source file)

# the text section has the assembly instructions
.section .text

        .global sum              #defines the function as global

sum:        # start of the function
    ...     # instructions
    ret
```

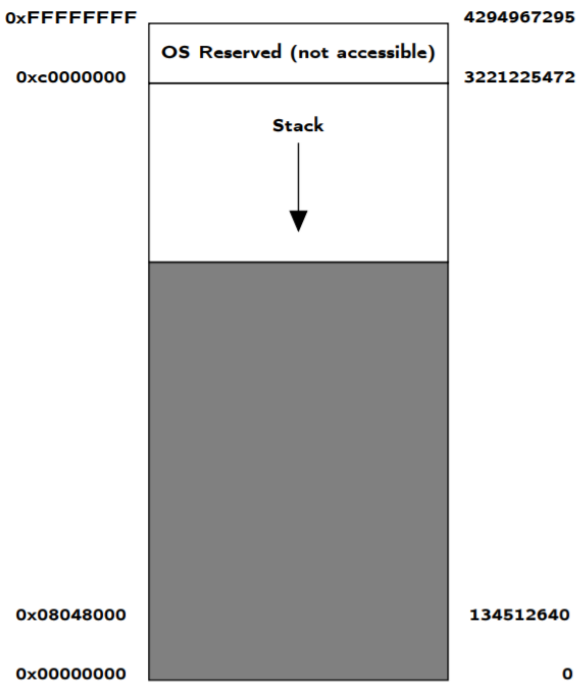
```
function:
   # prologue
   pushl %ebp       # save the original value of EBP
   movl %esp,%ebp   # copy the current stack pointer to EBP

   # callee is responsible for:
   # %ebx, %esi and %edi
   # save only those that are used
   ...

   # function body
   ...

   # restore callee saved registers
   ...

   #epilogue
   movl %ebp,%esp   # retrieve the original ESP value
   popl %ebp        # restore the original EBP value
   ret
```

Stack



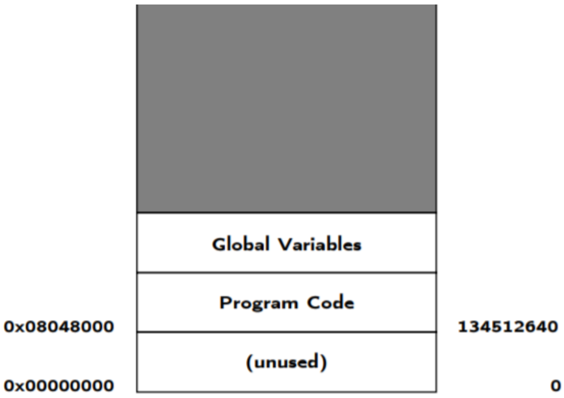


- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest allocated position on the stack (i.e., address of top element)

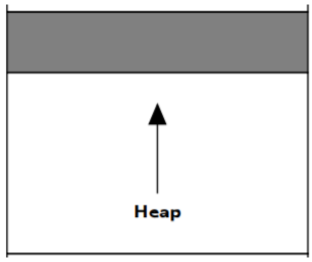


- The topmost region of the address space is reserved for operating system's code and data
- At the top of the user's virtual address space is the *stack*
- The user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts

- The lower region of the address space holds the code and global variables defined by the user's process



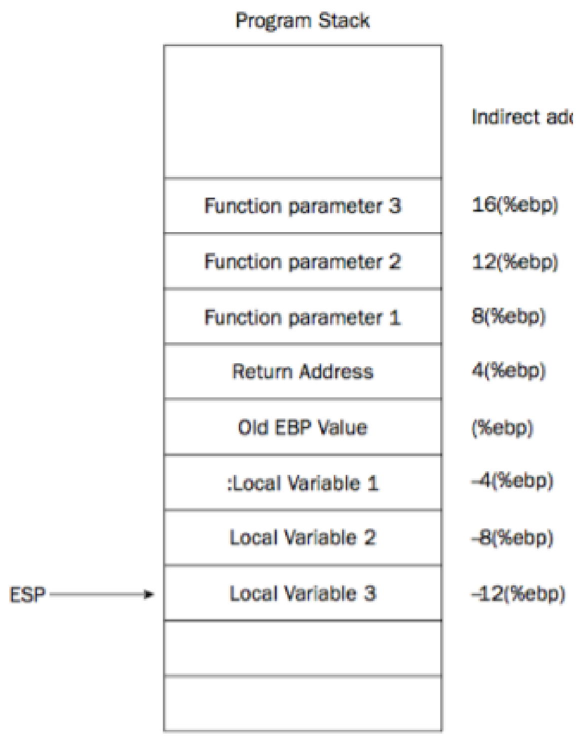
- The code and global variables are followed immediately by the *heap*
- The heap expands and contracts dynamically at run time as a result of calls to allocate/free memory



Local variables

Using a Local Variable for Temporary Storage

```
power:
    pushl % ebp              # prologue
    movl %esp , %ebp
    subl $4 ,%esp            # reserves space for a local variable
    pushl % ebx              # callee saves %ebx before using it
    movl 8(%ebp), %ebx       # base on %ebx
    movl 12(%ebp), %ecx      # exponent on %ecx
    movl $1 , -4(%ebp)       # initial result in local variable
power_loop_start:
    cmpl $0 , %ecx           # if exponenent is 0, go to end_power
    je end_power
    movl -4(%ebp), %eax      # multiplies current result by base
    imull %ebx
    movl %eax, -4(%ebp)
    decl %ecx                # decreases the exponent
    jmp power_loop_start     # jumps to next expoenent
end_power :
    movl -4(%ebp), %eax      # final result on %eax
    popl %ebx                # calle restores %ebx before ret
    movl %ebp, % esp         # epilogue
    popl %ebp
ret
```



Return Value

- To make our Assembly functions return:
  - a 32-bit value, leave that return value in the %eax register
  - a 64-bit value, leave the return value in the %edx:%eax registers

Flags

- CF - carry flag (bit 0) Set on most significant bit carry or borrow; cleared otherwise.
  - *adc origin, destination (destination = destination + origin + CF)*
  - *sbb origin, destination (destination = destination − (origin + CF))*
- PF - parity flag (bit 2) Set if least significant eight bits of result contain an even number of "1"bits; cleared otherwise.
- ZF - zero flag (bit 6) Set if result is zero; cleared otherwise.
- SF - sign flag (bit 7) Set equal to the most significant bit of result (0 if positive, 1 if negative).
- OF - overflow flag (bit 11) Set if result is too large (a positive number) or too small (a negative number, excluding its sign bit) to fit in destination operand; cleared otherwise.

Jumps

Trigger: *cmp operand1, operand2*

- JE – Jump if equal (ZF=1)
- JL – Jump if less (SF<>OF)
- JG – Jump if greater (ZF=0 e SF=OF)
- JC – Jump if carry (CF=1) **unsigned**
- JO – Jump if overflow (OF=1) **signed**

Arrays

Declaring arrays in C and Assembly

Array declaration in C

```c
// uninitialized array
long long int array_c[6];

// initialized array
int array_e[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60};
```

Array declaration in Assembly

```asm
.section .bss          # section BSS (uninitialized variables)

.comm array_c, 48      # space for 6 long long (6x8), name: array_c

.section .data         # section data (initialized variables)

array_e:               # name: array_e
    .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60 # array initialization
```

Declaring strings (arrays of `char`) in C and Assembly

String (array of char) declaration in C

```c
// uninitialized string (array of char)
char str_a[10];

// initialized string (array of char)
char str_b[] = "computer architecture";
```

String (array of char) declaration in Assembly

```asm
.section .bss         # section identifier

.comm str_a, 10       # space for 10 bytes; variable name: str_a

.section .data        # section identifier

str_b:                # variable name
    .asciz "computer architecture"
```

Iterate through a string

```
.section .data
str:
        .asciz "computer architecture"

.section .text
.global iterate_string # declare function as global
iterate_string:        # function start
# prologue
  ...

# body
  movl   $str, %edx    # address of string in %edx (notice the $)
string_loop:
  movb   (%edx),%cl    # copy char pointed by %edx to %cl
  cmpb   $0,%cl        # check if char is zero (end of string)
  je     end_loop      # no more chars in string

  ...                  # do something with char...

  incl   %edx          # moves pointer to next 1 byte
  jmp    string_loop   # jumps to next iteration

end_loop:              # we reached the end of the string...

# epilogue
  ...
```

Structures

Address of a struct member

Structure Declaration

```
struct rec {
  int x;
  int a[3];
  int *p;
}
```

Access in C

```
int* find_a(struct rec *r,
            int i)
{
    /* return the address
       of member a[i] */
    return &r->a[i];
}
```

Access in Assembly

```
# %ecx = i
movl 12(%ebp),%ecx
# %edx = r
movl 8(%ebp),%edx
# %eax = r + 4 + (4 * i)
leal 4(%edx,%ecx,4),%eax
```

Pointer and changing a structure member

Structure Declaration

```
struct rec {
  int x;
  int a[3];
  int *p;
}
```

Access in C

```
void set_p(struct rec *r)
{
  /* return the address
  of member a[i],
  where i is the value
  of member x */
  r->p=&r->a[r->x];
}
```

Access in Assembly

```
# %edx = r
movl 8(%ebp),%edx
# %ecx = r->x
movl (%edx),%ecx
# %eax = r + 4 + 4*(r->x)
leal 4(%edx,%ecx,4),%eax
# r->p = %eax
movl %eax,16(%edx)
```

Data alignment

The alignment rules are expressed as a function of the data size K:

- K = 1 byte: char
  No restrictions

- K = 2 bytes: short
  The least significant bit must be 0 (that is, the address must be a multiple of 2)
- K = 4 bytes: int, long int, float, char *, ...
  The 2 least significant bits must be 0 (that is, the address must be a multiple of 4)
- K = 8 bytes: double, long long, ...
  Windows: The 3 least significant bits must be 0 (that is, the address must be a multiple of 8)
  Linux: The 2 least significant bits must be 0 (that is, the address must be a multiple of 4); similar to K = 4) **only in IA32**

Dynamic memory allocation

- **void* malloc(size_type size);**
  Allocates a continuous memory block of **at least size** bytes and returns a pointer to it
  If malloc() encounters a problem (e.g., the program requests a block of memory that is larger than the available virtual memory), then it returns **NULL**
  The allocated memory is **not initialized**
- **void* calloc(size_type n, size_type size);**
  Reserves a continuous block of memory of **at least n * size** bytes. Returns a pointer to the allocated memory block, or **NULL** in error
  The memory block is **initialized to 0**
- **void *realloc(void *ptr, size_type size);**
  Changes the size of a memory block previously allocated with malloc() ou calloc()
  Returns a pointer to the allocated memory block, or **NULL** in error
  Data is kept if size is increased, or truncated if size is decreased
- **void free(void *ptr);**
  Free allocated heap memory blocks previously allocated with a memory allocation call. **The ptr argument must point to the beginning of an allocated block** that was obtained from a memory allocation call

malloc, calloc and realloc must be cast to (ptr_type) when called.

```
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointers */
int *ptr_int=NULL, ptr_tmp=NULL;
/* number of ints to allocate */
int n=100;

/* allocate n integers in the heap */
ptr_int=(int *) calloc(n, sizeof(int));

/* allocate one more integer in the heap
   NOTE: return is stored in temporary pointer */
ptr_tmp=(int *) realloc(ptr_int,(n+1) * sizeof(int));

/* check realloc() return */
if(ptr_tmp!=NULL){
        ptr_int=ptr_tmp;
        ptr_tmp=NULL;
}

/* free memory */
free(ptr_int);
```

Multidimensional arrays

Static

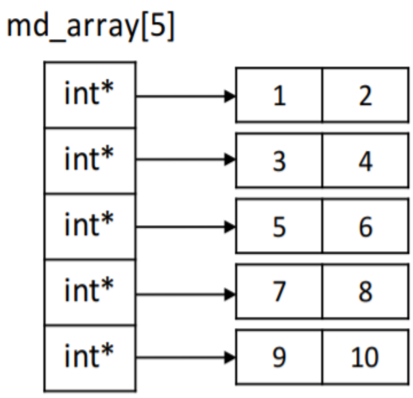
Accessing value in C

```c
int get_value(int md_array[5][2], int i, int j){
  return md_array[i][j];
}
```

Accessing value in Assembly

```
get_value_asm:
        pushl %ebp
        movl %esp, %ebp

        movl 8(%ebp),%edx        #matrix address
        movl 12(%ebp),%ecx       #i
        shll $3,%ecx             #each line has 2 integers (8 bytes)
        addl %ecx,%edx           #address of line i
        movl 16(%ebp),%ecx       #j
        movl (%edx,%ecx,4),%eax  #m[i][j]

        movl %ebp, %esp
        popl %ebp
        ret
```

Dynamic



Expression	Type
$md\_array[2]$	Pointer to integer
md_array	Pointer to array of two integers
$md\_array + 1$	Pointer to array of two integers
$*(md\_array + 1)$	Pointer to integer
$*(md\_array + 2) + 1$	Pointer to integer
$*(*(md\_array + 2) + 1)$	Integer (md_array[2][1])
$*md\_array$	Pointer to integer
$**md\_array$	Integer (md_array[0][0])
$*(*md\_array + 1)$	Integer (md_array[0][1])

Creating a dynamic array

Allocate variable-size multidimensional array

```c
int main(void)
{
    int m;
    int y=5,k=10; /* number of lines (Y) and columns (K) */
    int **a;      /* address of the multi-dimensional array */

    /*  array of int* with size Y */
    a = (int**) calloc(y,sizeof(int*));
    if(a == NULL){
        printf("Error reserving memory.\n"); exit(1);
    }

    for(m = 0; m < y ; m++){
        /* in each position of the pointer array,
           reserve memory for K integers */
        *(a+m) = (int*) calloc(k,sizeof(int)); //Note: *(a+m) same as a[m]
        if(a[m] == NULL){
            printf("Error reserving memory.\n"); exit(1);
        }
    }

    /* free memory */
    for(m = 0; m < y ; m++) free(*(a+m));
    free(a);
    return 0;
}
```

Accessing a dynamic array

Accessing value in C

```c
int get_value(int **md_array, int i, int j){
  return md_array[i][j];
}
```

Accessing value in Assembly

```
get_value_asm:
        pushl %ebp
        movl %esp, %ebp

        movl 8(%ebp),%edx        #matrix address
        movl 12(%ebp),%ecx       #i
        movl (%edx,%ecx,4),%edx  #address of line i (first memory access)
        movl 16(%ebp),%ecx       #j
        movl (%edx,%ecx,4),%eax  #m[i][j] (second memory access)

        movl %ebp, %esp
        popl %ebp
        ret
```