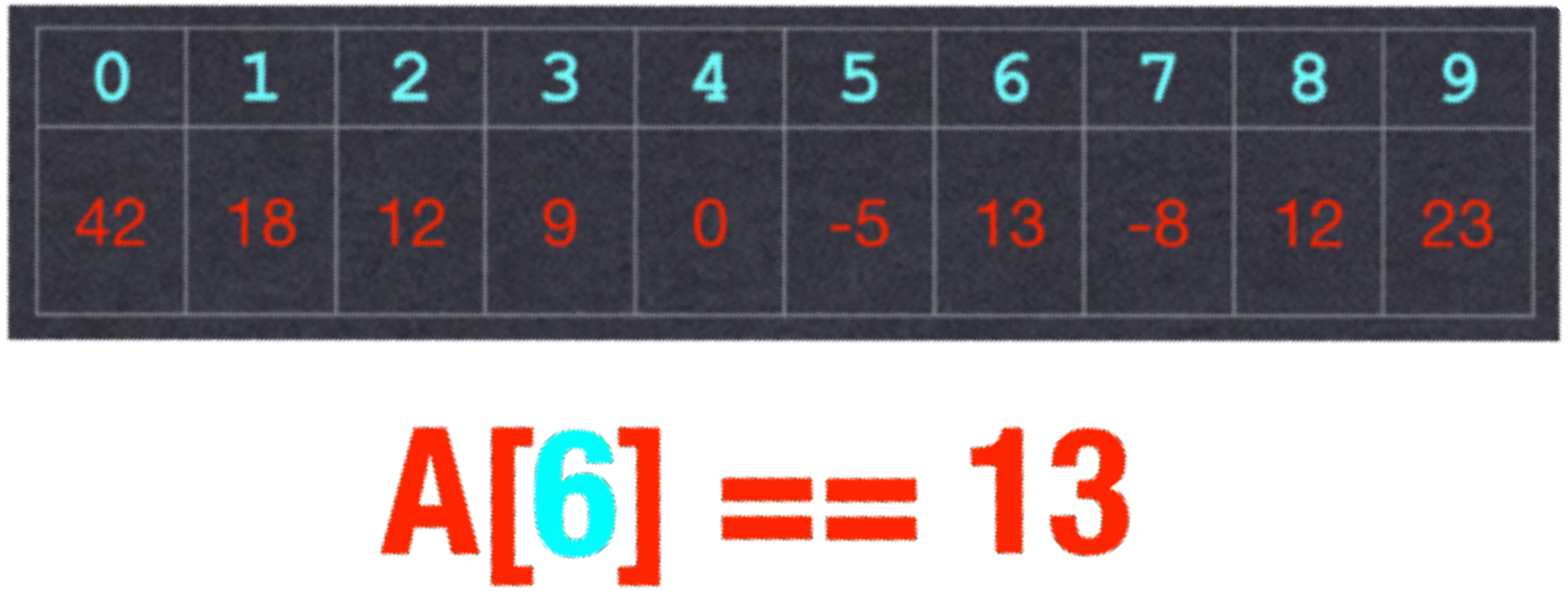
CS 107
Lecture 12:
Assembly Part IV

Friday, February 24, 2023

Computer Systems
Winter 2023
Stanford University
Computer Science Department




```
0x40055d <val_at_index>      movslq %esi,%rsi
0x400560 <val_at_index+3>    mov    (%rdi,%rsi,4),%eax
0x400563 <val_at_index+6>    retq
```

Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.1-3.4

Lecturer: Chris Gregg

Today's Topics

- Reading: Chapter 3.7.4-3.7.6, 3.8-3.9
- Programs from class: `/afs/ir/class/cs107/samples/lect12`
- Logistics
  - Midterm requests in by Thursday
- Final day of x86 Assembly Language
  - Procedures (from last slide-deck)
    - Local storage on the stack
    - Local storage in registers
    - Recursion
  - Arrays
  - Structures
  - Alignment
  - Function pointers
- Assembly wrap-up, and comments on bank assignment.

Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are *very* good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type *T* and integer constant *N:*

T **`A[`***N***`]`**

- The starting location is designated as x_A
- The declaration allocates *N* **`* sizeof(`***T***`)`** bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of x_A.

Array Allocation and Access

- Example:

```
char      A[12];
char     *B[8];
int       C[6];
double   *D[5]
```

Array	Element Size	Total Size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

- The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access `C[3]` above. If the address of C is in register `%rdx`, and `3` is in register `%rcx`
- The following copies `C[3]` into `%eax`,

```
movl (%rdx,%rcx,4), %eax
```

Pointer Arithmetic

- C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
- The array reference **`A[i]`** is identical to **`*(A+i)`**
- Example: if the address of array E is in **`%rdx`**, and the integer index, **`i`**, is in **`%rcx`**, the following are some expressions involving E:

Expression	Type	Value	Assembly Code
`E`	int *	x_E	`movq %rdx, %rax`
`E[0]`	int	$M[x_E]$	`movl (%rdx), %eax`
`E[i]`	int	$M[x_E+4i]$	`movl (%rdx,%rcx,4) %eax`
`&E[2]`	int *	x_E+8	`leaq 8(%rdx), %rax`
`E+i-1`	int *	x_E+4i-4	`leaq -4(%rdx,%rcx,4), %rax`
`*(E+i-3)`	int	$M[x_E+4i-12]$	`movl -12(%rdx,%rcx,4) %eax`
`&E[i]-E`	long	i	`movq %rcx,%rax`

Pointer Arithmetic

- Practice: x_S is the address of a **`short`** integer array, S, stored in **`%rdx`**, and a long integer index, **`i`**, is stored in register **`%rcx`**.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in **`%rax`** if it is a pointer, and the result should be in register **`%ax`** if it has a data type **`short`**.

Expression	Type	Value	Assembly Code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

Pointer Arithmetic

- Practice: x_S is the address of a **`short`** integer array, S, stored in **`%rdx`**, and a long integer index, **`i`**, is stored in register **`%rcx`**.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in **`%rax`** if it is a pointer, and the result should be in register **`%ax`** if it has a data type **`short`**.

Expression	Type	Value	Assembly Code
`S+1`	`short *`	x_S + 2	`leaq 2(%rdx),%rax`
`S[3]`	`short`	M[x_S + 6]	`movw 6(%rdx),%ax`
`&S[i]`	`short *`	x_S + 2i	`leaq (%rdx,%rcx,2),%rax`
`S[4*i+1]`	`short`	M[x_S + 8i + 2]	`movw 2(%rdx,%rcx,8),%ax`
`S+i-5`	`short *`	x_S + 2i - 10	`leaq -10(%rdx,%rcx,2),%rax`

Structures

- The C `struct` declaration is used to group objects of different types into a single unit.
- Each "field" is referenced by a name, and can be accessed using dot (`.`) or (if there is a pointer to the struct) arrow (`->`) notation.
- Structures are kept in contiguous memory
- A pointer to a struct is to its first byte, and the compiler maintains the byte offset information for each field.
- In assembly, the references to the fields are via the byte offsets.

Structures

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type **`int`**, a two-element array of type **`int`**, and an 8-byte **`int`** pointer, for a total of 24 bytes:

Offset	0	4	8		16	24
Contents	i	j	a[0]	a[1]	p	

- The numbers along the top of the diagram are the byte offsets of the fields from the beginning of the structure.
- Note that the array is embedded in the structure.
- To access the fields, the compiler generates code that adds the field offset to the address of the structure.

Structures

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

Offset	0	4	8		16	24
Contents	i	j	a[0]	a[1]	p	

- Example: Variable `r` of type struct `rec *` is in register `%rdi`. The following copies element `r->i` to element `r->j`:

```
movl (%rdi), %eax    // get r->i
movl %eax, 4(%rdi)   // store in r->j
```

- The offset of `i` is `0`, so `i`'s field is `%rdi`. The offset of `j` is `4`, so the offset of `4` is added to the address of `%rdi` to store into `j`.

Structures

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type **`int`**, a two-element array of type **`int`**, and an 8-byte **`int`** pointer, for a total of 24 bytes:

Offset	0	4	8		16	24
Contents	`i`	`j`	`a[0]`	`a[1]`	`p`	

- We can generate a pointer to a field by adding the field's offset to the struct address. To generate **`&(r->a[1])`** we add offset **`8 + 4 = 12`**. For a pointer **`r`** in register **`%rdi`** and long **`int`** variable **`i`** in **`%rsi`**, we can generate the pointer value **`&(r->a[i])`** with one instruction:

```
leaq 8(%rdi,%rsi,4), %rax // set %rax to &r->a[i]
```

Structures

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

Offset	0	4	8		16	24
Contents	i	j	a[0]	a[1]	p	

- The following code implements `r->p = &r->a[r->i + r->j];`

```
// r in %rdi
movl 4(%rdi),%eax          // get r->j
addl (%rdi),%eax           // add r->i
cltq                       // extend %eax to 8 bytes, %rax
leaq 8(%rdi,%rax,4), %rax  // compute &r->a[r->i + r->j]
movq %rax, 16(%rdi)        // store in r->p
```

Structures

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type **`int`**, a two-element array of type **`int`**, and an 8-byte **`int`** pointer, for a total of 24 bytes:

Offset	0	4	8	12	16	24
Contents	i	j	a[0]	a[1]	p	

- Notice that all struct manipulation is handled at compile time, and the machine code doesn't contain any information about the field declarations or the names of the fields.
- The compiler does all the work, keeping track as it produces the assembly code.
- BTW, if you're curious about how the compiler actually does the transformation from C to assembly, take a compilers class, e.g., CS143.

Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value *K* (normally 2, 4, or 8).
- These *alignment restrictions* simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any **`double`** will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:

K	Types
1	`char`
2	`short`
4	`int, float`
8	`long, double, char *`

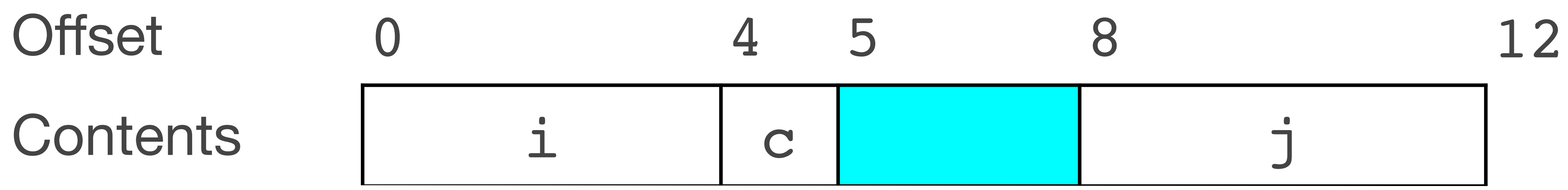
Data Alignment

- The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.
- For example, let's look at the following struct:

```
struct S1 {
    int i;
    char c;
    int j;
};
```

- If the compiler used a minimal allocation:

- This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:



- So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!

Function Pointers in Assembly

- Let's look at the following code:

```
void *gfind_max(void *arr, int n, size_t elemsz,
                int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```

Function Pointers in Assembly

- Let's look at the following code:

```
void *gfind_max(void *arr, int n, size_t elemsz,
                int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```

- Because **`compar`** is a function pointer, the compiler calls the function via the address that is in the compar variable.
- Let's take a look at this in gdb.

References and Advanced Reading

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64

Extra Slides