

CS 24

# Introduction to Computer Systems

6: Data

# Questions?

- Compiler/linker errors for `-lgc` or `libgc`.
- Remember: GDB mini-lecture in lab, 8:30.

# Data Representation

- For today, some C-like data types.
- Integral types.
- Pointers.
- Arrays.
- Structures.

# Types of Integers

- C has a few:
  - int
  - short int (or just short)
  - long int (long)
  - long long int (long long)
  - unsigned int (unsigned short int, etc.)
  - char
- Differ by # of bits, signed/unsigned.
- In general, sizes depend on platform.  
For x86 under 32-bit Windows/Linux:
  - int and long are both 32 bits.
  - short is 16 bits.
  - char is 8 bits.
  - long long is 64 bits. (Doesn't fit in a register – needs special treatment.)

# Alignment

- When storing integers in memory,
  - a 16-bit value should be stored at an even-numbered address.
  - a 32-bit value should be stored at an address divisible by 4.
- Why?
  - Because Intel's architects expected you to.
  - Unaligned memory accesses slow things down.
  - Also, alignment affects layout of structs (later). For C compatibility, must pay attention.

# Pointers

- For those new to C, meet the operators `&` (address-of) and `*` (contents-of).

# Pointers

- For those new to C, meet the operators & (address-of) and \* (contents-of).

```
int x = 3;      /* x is a variable storing an int (3). */
```

# Pointers

- For those new to C, meet the operators & (address-of) and \* (contents-of).

```
int x = 3;      /* x is a variable storing an int (3). */  
int *y;        /* y stores the address of an int.  
               * i.e., y is a pointer to an int.   */
```

# Pointers

- For those new to C, meet the operators & (address-of) and \* (contents-of).

```
int x = 3;      /* x is a variable storing an int (3). */
int *y;        /* y stores the address of an int.
               * i.e., y is a pointer to an int.      */
y = &x;        /* y assigned the address of x.
               * y "points to" x.                    */
```

# Pointers

- For those new to C, meet the operators & (address-of) and \* (contents-of).

```
int x = 3;      /* x is a variable storing an int (3). */
int *y;        /* y stores the address of an int.
               * i.e., y is a pointer to an int.      */
y = &x;        /* y assigned the address of x.
               * y "points to" x.                      */
int z = *y;    /* z assigned value at address y.
               * i.e., z is now 3.                      */
```

# Pointers

- For those new to C, meet the operators & (address-of) and \* (contents-of).

```
int x = 3;      /* x is a variable storing an int (3). */
int *y;        /* y stores the address of an int.
               * i.e., y is a pointer to an int.      */
y = &x;        /* y assigned the address of x.
               * y "points to" x.                    */
int z = *y;    /* z assigned value at address y.
               * i.e., z is now 3.                    */
*y = 17;       /* 17 stored to location y.
               * i.e., x is now 17.                    */
```

# Pointers as Addresses

```
int x = 3;
```

```
int *y = &x;
```

```
int z = *y;
```

```
*y = 17;
```

# Pointers as Addresses

```
int x = 3;
```

```
movl $3, -4(%ebp)
```

```
int *y = &x;
```

```
int z = *y;
```

```
*y = 17;
```

# Pointers as Addresses

```
int x = 3;
```

```
movl $3, -4(%ebp)
```

```
mov %ebp,%eax
```

```
sub $4,%eax
```

```
int *y = &x;
```

```
mov %eax,-8(%ebp)
```

```
int z = *y;
```

```
*y = 17;
```

# Pointers as Addresses

```
int x = 3;
```

```
movl $3, -4(%ebp)
```

```
int *y = &x;
```

```
lea -4(%ebp),%eax  
mov %eax,-8(%ebp)
```

```
int z = *y;
```

```
*y = 17;
```

# Pointers as Addresses

```
int x = 3;
```

```
movl $3, -4(%ebp)
```

```
int *y = &x;
```

```
lea -4(%ebp),%eax  
mov %eax,-8(%ebp)
```

```
int z = *y;
```

```
mov -8(%ebp),%eax  
mov (%eax),%eax  
mov %eax,-12(%ebp)
```

```
*y = 17;
```

# Pointers as Addresses

```
int x = 3;
```

```
movl $3, -4(%ebp)
```

```
int *y = &x;
```

```
lea -4(%ebp),%eax  
mov %eax,-8(%ebp)
```

```
int z = *y;
```

```
mov -8(%ebp),%eax  
mov (%eax),%eax  
mov %eax,-12(%ebp)
```

```
*y = 17;
```

```
mov -8(%ebp),%eax  
movl $17,(%eax)
```

# Pointer Size

- Pointers are addresses.
- Addresses are 32 bits.
- All pointers are the same size, regardless of what type they point to.

`int * / long * / char * / ...`

# Pointer Size

- Pointers are addresses.
- Addresses are 32 bits.
- All pointers are the same size, regardless of what type they point to.

`int * / long * / char * / ...`

- **Aside:**
  - Pointer is same size as int.
  - C lets you cast pointers to ints and vice versa.
  - Classic type-unsafety!!

# Arrays

- Bunch of things, all the same size.
- Just put into contiguous block of memory.

# Array Indexing

```
int a[20];  
a[4] = 56;
```

```
int i;  
a[i] = 87;
```

```
short b[20];  
b[i] = 46;
```

# Array Indexing

```
int a[20];  
a[4] = 56;
```

```
lea adisp(%ebp),%eax  
movl $56,16(%eax)
```

```
int i;  
a[i] = 87;
```

```
short b[20];  
b[i] = 46;
```

# Array Indexing

```
int a[20];  
a[4] = 56;
```

```
lea adisp(%ebp),%eax  
movl $56,16(%eax)
```

```
int i;  
a[i] = 87;
```

```
lea adisp(%ebp),%eax  
mov idisp(%ebp),%ecx  
movl $87,(%eax,%ecx,4)
```

```
short b[20];  
b[i] = 46;
```

# Array Indexing

```
int a[20];  
a[4] = 56;
```

```
lea adisp(%ebp),%eax  
movl $56,16(%eax)
```

```
int i;  
a[i] = 87;
```

```
lea adisp(%ebp),%eax  
mov idisp(%ebp),%ecx  
movl $87,(%eax,%ecx,4)
```

```
short b[20];  
b[i] = 46;
```

```
lea bdisp(%ebp),%eax  
movl idisp(%ebp),%ecx  
movw $46,(%eax,%ecx,2)
```

# Array Bounds

Consider the C statement:

```
a[i] = x;
```

*What happens if index is out of bounds?*

# Array Bounds

Consider the C statement:

```
a[i] = x;
```

*What happens if index is out of bounds?*

- C language specification doesn't say (iirc).
  - Compilers allowed to ignore the issue. Most do.
- If it happens, store to bad index can overwrite other data.

# Arrays and Pointers

For those new to C...

```
void f() {  
    int a[25];           /* 100-byte block in stack frame */  
    g(a,7);             /* after call, a[7] == 12.          */  
}                       /* array destroyed on return.     */  
  
void g(int a[],int i) { /* Needs address of a, not contents. */  
    a[i] = 12;         /* (so length of array is irrelevant) */  
}
```

# Arrays and Pointers

For those new to C...

```
void f() {  
    int a[25];          /* 100-byte block in stack frame */  
    g(a,7);            /* after call, a[7] == 12.          */  
}                      /* array destroyed on return.    */
```

```
void g(int a[],int i) { /* Needs address of a, not contents. */  
    a[i] = 12;          /* (so length of array is irrelevant) */  
}
```

```
void h(int *a,int i) { /* This is equivalent to g.          */  
    *(a+i) = 12;      /* Pointer arithmetic: i implicitly scaled */  
}                      /* by sizeof(int).                      */
```

# Arrays and Pointers

For those new to C...

```
void f() {  
    int a[25];           /* 100-byte block in stack frame */  
    g(a,7);             /* after call, a[7] == 12.          */  
}                       /* array destroyed on return.     */  
  
void g(int a[],int i) { /* Needs address of a, not contents. */  
    a[i] = 12;         /* (so length of array is irrelevant) */  
}  
  
void h(int *a,int i) { /* This is still equivalent to g.    */  
    a[i] = 12;         /* Can use [] notation with pointers! */  
}
```

# Arrays and Pointers

For those new to C...

```
void f() {  
    int a[25];           /* 100-byte block in stack frame */  
    g(a+4,3);           /* (was g(a,7); – equivalent!) */  
}                       /* array destroyed on return. */  
  
void g(int a[],int i) { /* Needs address of a, not contents. */  
    a[i] = 12;         /* (so length of array is irrelevant) */  
}  
  
void h(int *a,int i) { /* This is still equivalent to g. */  
    a[i] = 12;         /* Can use [] notation with pointers! */  
}
```

# Structures

- Arrays have unbounded length, but all elements are same size.
- Structures have fixed number of *fields*, which may be different sizes.

```
/* Declare type. */  
struct mystructtype {  
    char x;  
    int y;  
    short z;  
};
```

```
struct mystructtype s;
```

```
s.x = 'A';  
s.y = 40000;  
s.z = 93;
```

# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```

# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```

Structure must start on 4-byte boundary.  
(address must be multiple of 4)

# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```

**x**

# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```



# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```



# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```



# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```



# Representation

- Just put fields one after other, in order.
- But pay attention to alignment!
  - May have to waste some space.
  - Align structure according to most-demanding member.
  - Pad between fields where needed;  
Pad at end to make multiple of largest alignment.

```
struct mystructtype{ char x; int y; short z; };
```



If *only your code* will touch a struct, lay it out however you like.  
If structs will be passed between your code and C code, follow these rules.

# Access



- Once layout decided, access is easy.
- If struct `s` begins at address `A`,
  - `s.x` is at `A`.
  - `s.y` is at `A+4`.
  - `s.z` is at `A+8`.
  - Offsets can be statically computed!
- Simpler than arrays, except for alignment issues.

# Union Types

- `struct foo {int x; struct bar *y;}`  
means a foo consists of an int and a pointer to a bar.
- `union foo {int x; struct bar *y;}`  
means a foo consists of an int or a pointer to a bar.
  - Not both at the same time.
  - Can't necessarily tell which (programmer must keep track).

```
{    union foo f,g;  
    f.x = 37;  
    g.y = (struct bar *) ... ;  
}
```

# Representing Unions

*How do we represent a union value?*

# Representing Unions

*How do we represent a union value?*

- Need a block of memory that is
  - At least big enough for largest “field” of union.
  - Aligned properly for most demanding field.

# Representing Unions

*How do we represent a union value?*

- Need a block of memory that is
  - At least big enough for largest “field” of union.
  - Aligned properly for most demanding field.
- Example:  
`union myunion { int k; struct mystructtype m; void *p;};`

