Introduction to Computer Systems
15-213, fall 2009
8th Lecture, Sep. 16th

Instructors:
Majd Sakr and Khaled Harras
Last Time

- **Arrays**

  ```
  int val[5];
  ```

- **Nested**

  ```
  int pgh[4][5];
  ```

- **Multi-level**

  ```
  int *univ[3]
  ```
Dynamic Nested Arrays

- **Strength**
  - Can create matrix of any size

- **Programming**
  - Must do index computation explicitly

- **Performance**
  - Accessing single element costly
  - Must do multiplication

```c
int * new_var_matrix(int n)
{
    return (int *)
        calloc(sizeof(int), n*n);
}

int var_ele
    (int *a, int i, int j, int n)
{
    return a[i*n+j];
}
```

```assembly
movl 12(%ebp),%eax # i
movl 8(%ebp),%edx # a
imull 20(%ebp),%eax # n*i
addl 16(%ebp),%eax # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```
Dynamic Array Multiplication

- **Per iteration:**
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 for array indexing
    - 1 for loop index
    - 1 for data

/* Compute element $i,k$ of variable matrix product */

```c
int var_prod_ele
(int *a, int *b,
  int i, int k, int n)
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```

```plaintext
\[ a \times b = x \]
```
Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to `-O2`

- **Code Motion**
  - Expression `i*n` can be computed outside loop

- **Strength Reduction**
  - Incrementing `j` has effect of incrementing `j*n+k` by `n`

- **Operations count**
  - 4 adds, 1 mult

```c
{  
    int j;  
    int result = 0;  
    for (j = 0; j < n; j++)  
        result +=  
            a[i*n+j] * b[j*n+k];  
    return result;  
}
```

```c
{  
    int j;  
    int result = 0;  
    int iTn = i*n;  
    int jTnPk = k;  
    for (j = 0; j < n; j++)  
    {  
        result +=  
            a[iTn+j] * b[jTnPk];  
        jTnPk += n;  
    }  
    return result;  
}
```
Today

- Structures
- Alignment
- Unions
Structures

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

**Concept**
- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

**Accessing Structure Member**
```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

**IA32 Assembly**
```assembly
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```
Generating Pointer to Structure Member

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

```c
int *find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```c
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax
leal 4(%eax,%edx),%eax
```

**What does it do?**

**Will disappear blackboard?**
Generating Pointer to Structure Member

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

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

```
int *find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```
Structure Referencing (Cont.)

- C Code

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

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

What does it do?

```assembly
# %edx = r
movl (%edx),%ecx     # r->i
leal 0(,%ecx,4),%eax  # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx)  # Update r->p
```
Today

- Structures
- Alignment
- Unions
Alignment

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (IA32)

- **1 byte**: `char`, ...
  - no restrictions on address

- **2 bytes**: `short`, ...
  - lowest 1 bit of address must be $0_2$

- **4 bytes**: `int`, `float`, `char *`, ...
  - lowest 2 bits of address must be $00_2$

- **8 bytes**: `double`, ...
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be $000_2$
  - Linux:
    - lowest 2 bits of address must be $00_2$
    - i.e., treated the same as a 4-byte primitive data type

- **12 bytes**: `long double`
  - Windows, Linux:
    - lowest 2 bits of address must be $00_2$
    - i.e., treated the same as a 4-byte primitive data type
Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address

- **2 bytes: short, ...**
  - lowest 1 bit of address must be 0₂

- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be 00₂

- **8 bytes: double, char *, ...**
  - Windows & Linux:
    - lowest 3 bits of address must be 000₂

- **16 bytes: long double**
  - Linux:
    - lowest 3 bits of address must be 000₂
    - i.e., treated the same as a 8-byte primitive data type
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K =$ Largest alignment of any element
  - Initial address & structure length must be multiples of $K$

- **Example (under Windows or x86-64):**
  - $K = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Different Alignment Conventions

- **x86-64 or IA32 Windows:**
  - $K = 8$, due to `double` element
  
  ```
  struct S1 {
    char c;
    int i[2];
    double v;
  } *p;
  ```

- **IA32 Linux**
  - $K = 4$; `double` treated like a 4-byte data type
Saving Space

- Put large data types first

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```

- Effect (example x86-64, both have \(K=8\))

```
p+0  p+4  p+8  p+16  p+24

3 bytes   i[0]   i[1]  4 bytes   v
```
Arrays of Structures

- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

```c
short get_j(int idx) {
    return a[idx].j;
}
```

```assembly
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```
Today

- Structures
- Alignment
- Unions
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
Using Union to Access Bit Patterns

typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u ?

Same as (unsigned) f ?
So Far:

- **Structures**

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

- **Alignment**

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- **Unions**

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```
Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking

- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment

- **Unions**
  - Overlay declarations
  - Way to circumvent type system