CS 4400
Computer Systems

LECTURE 8

Array allocation and access

Arrays in C

- Array declaration `T A[N];`
  - allocates a contiguous region of $L \cdot N$ bytes, L is the size of T
  - introduces A as a constant pointer to the beginning of the array
- Let `xA` be address stored in A, element `i` is stored at $xA + L \cdot i$.

Arrays in C

- With IA32's flexible addressing modes, translation to assembly code is straightforward.

- Suppose:

  ```
  int E[]
  %edx = &E
  %ecx = i

  movl (%edx,%ecx,4),%eax
  ```

  results in `%eax = E[i]`

- Optimizing compilers are particularly good at simplifying address computations, which may make assembly code hard to read

Pointer Arithmetic

- Computed value is scaled according to size of data type.
  - for `int* p`, expression `p+k` has value `xp+4·k`
  - for `char* str`, what is the value of expression `str+j`?

- Array subscripting operation can be applied to array names and other pointers.
  - `A[i]` equivalent to `*(A+i)`

Pointer Arithmetic

- Examples
  - `%edx`: address of `E`
  - `%ecx`: value of `i`
  - `%eax`: result

```
E[2]

E+i-1

*(&E[i]+i)
```

Pointer Arithmetic

- Examples
  - `%edx`: address of `E`
  - `%ecx`: value of `i`
  - `%eax`: result

```
E[2]            movl 8(%edx),%eax

E+i-1

*(&E[i]+i)
```

Pointer Arithmetic

- Examples
  - `%edx`: address of `E`
  - `%ecx`: value of `i`
  - `%eax`: result

```
E[2]          movl 8(%edx),%eax

E+i-1         leal -4(%edx,%ecx,4),%eax

*(&E[i]+i)
```

Pointer Arithmetic

- Examples
  - `%edx`: address of E
  - `%ecx`: value of `i`
  - `%eax`: result

```
E[2]            movl 8(%edx),%eax

E+i-1           leal -4(%edx,%ecx,4),%eax

*(&E[i]+i)      leal (%edx,%ecx,4),%eax
                leal (%eax,%ecx,4),%eax
```

or

```
                leal (%edx,%ecx,8),%eax
```

A Few Pointer Arithmetic Rules

- OK to add an integer to a pointer
- OK to subtract an integer from a pointer
- OK to subtract a pointer from a pointer
  - But only if both pointers have same type and point into the same array
  - Otherwise, behavior is undefined
  - Result is an integer
- NOT OK to add a pointer to a pointer

Exercise: Pointer Arithmetic

- For: short `S[]` and index `i`
- Let: `%edx = &S` and `%ecx = i`
- Put a pointer result in `%eax`
- Put short result in `%ax`

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

Exercise: Pointer Arithmetic

- For: short `S[]` and index `i`
- Let: `%edx = &S` and `%ecx = i`
- Put a pointer result in `%eax`
- Put short result in `%ax`

	Type	Value	Assembly Code
`S+1`	`short*`	`xS+2`	`leal 2(%edx),%eax`
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

Exercise: Pointer Arithmetic

- For: short `S[]` and index `i`
- Let: `%edx = &S` and `%ecx = i`
- Put a pointer result in `%eax`
- Put short result in `%ax`

	Type	Value	Assembly Code
S+1	short*	xS+2	leal 2(%edx),%eax
S[3]	short	*(xS+6)	movw 6(%edx),%ax
&S[i]			
S[4*i+1]			
S+i-5			

Exercise: Pointer Arithmetic

- For: short `S[]` and index `i`
- Let: `%edx = &S` and `%ecx = i`
- Put a pointer result in `%eax`
- Put short result in `%ax`

	Type	Value	Assembly Code
`S+1`	`short*`	`xS+2`	`leal 2(%edx),%eax`
`S[3]`	`short`	`*(xS+6)`	`movw 6(%edx),%ax`
`&S[i]`	`short*`	`xS+2*i`	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`			
`S+i-5`			

Exercise: Pointer Arithmetic

- For: short `S[]` and index `i`
- Let: `%edx = &S` and `%ecx = i`
- Put a pointer result in `%eax`
- Put short result in `%ax`

	Type	Value	Assembly Code
`S+1`	`short*`	`xS+2`	`leal 2(%edx),%eax`
`S[3]`	`short`	`*(xS+6)`	`movw 6(%edx),%ax`
`&S[i]`	`short*`	`xS+2*i`	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	`*(xS+8*i+2)`	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`			

Exercise: Pointer Arithmetic

- For: short `S[]` and index `i`
- Let: `%edx = &S` and `%ecx = i`
- Put a pointer result in `%eax`
- Put short result in `%ax`

	Type	Value	Assembly Code
`S+1`	`short*`	`xS+2`	`leal 2(%edx),%eax`
`S[3]`	`short`	`*(xS+6)`	`movw 6(%edx),%ax`
`&S[i]`	`short*`	`xS+2*i`	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	`*(xS+8*i+2)`	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short*`	`xS+2*i-10`	`leal 10(%edx,%ecx,2),%eax`

Question

Suppose we have declared `int arr[N]`.

Are the following is equivalent to the reference `arr[i]`?

A. `*(arr + 4 * i)`
B. `*(&arr[0] + i)`
C. `*((int*)((char*)arr + 4 * i))`

Question

Suppose we have declared `char* arr[N]`.

Which of the following correctly puts `arr[i]` in `%eax`? (Suppose that `arr` in `%edx` and `i` in `%eax`.)

A. `leal (%edx,%eax),%eax`
B. `leal (%edx,%eax,4),%eax`
C. `movl (%edx,%eax),%eax`
D. `movl (%edx,%eax,4),%eax`
E. none of the above

Arrays and Loops

- Array references in loops often have very regular patterns.

```
for(i = 0, val = 0; i < 5; i++)
    val = (10 * val) + x[i];
```

- For efficiency, optimizing compilers exploit these patterns.

```
    xorl %eax,%eax              ;val=0
    leal 16(%ecx),%ebx          ;xend=x+4
.L12:
    leal (%eax,%eax,4),%edx     ;compute 5*val
    movl (%ecx),%eax            ;compute *x
    leal (%eax,%edx,2),%eax     ;compute *x+2*(5*val)
    addl $4,%ecx  ;x++
    cmpl %ebx,%ecx              ;compare x-xend
    jbe .L12                    ;if x<=xend, goto loop
```

- Uses pointer arithmetic instead of loop index `i`.

```
int* xend = x + 4;
do {
    val = (10 * val) + *x;
} while(++x <= xend);
```

Multi-Dimensional Arrays

- The same principles hold for arrays of arrays.
  - `int A[4][3];` is an array of four 3-integer arrays ("rows")
  - arrays are linearized in memory in row-major order
- `A[i][j]` is at memory address `xA+ L (C · i + j)`.
- Example (`%eax`: address of `A`, `%edx`: value of `i`, `%ecx`: value of `j`)

```
sall $2,%ecx  ;j*4
leal (%edx,%edx,2),%edx  ;i*3
leal (%ecx,%edx,4),%edx  ;j*4 + i*12
movl (%eax,%edx),%eax    ;read A[i][j]
```

fixed-sized array

```c
#define N 16
typedef int fix_matrix[N][N];

int fix_prod(fix_matrix A, fix_matrix B, int i, int
k) {
  int j, result;

  for(j = 0, result = 0; j < N; j++)
    result += A[i][j] * B[j][k];

  return result;
}
```

```c
int fix_prod(fix_matrix A, fix_matrix B,
                          int i, int k) {
  int *Aptr, *Bptr, cnt, result;
  Aptr = &A[i][0];
  Bptr = &B[0][k];
  cnt = N-1;
  result = 0;

  do {
    result += (*Aptr) * (*Bptr);
    Aptr++;
    Bptr += N;
    cnt--;
  } while(cnt >= 0);

  return result;
}
```

compiler optimizations

```
Aptr is in %edx
Bptr is in %ecx
result is in %esi
cnt is in %ebx

.L23:
  movl (%edx),%eax
  imull (%ecx),%eax
  addl %eax,%esi
  addl $64,%ecx
  addl $4,%edx
  decl %ebx
  jns .L23
```

Exercise: Nested Arrays

```c
#define M ??
#define N ??

int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j) {
  return mat1[i][j] + mat2[j][i];
}
```

```
movl 8(%ebp),%ecx
movl 12(%ebp),%eax
leal 0(,%eax,4),%ebx
leal 0(,%ecx,8),%edx
subl %ecx,%edx
addl %ebx,%eax
sall $2,%eax
movl mat2(%eax,%ecx,4),%eax
addl mat1(%ebx,%edx,4),%eax
```

Question

Will the following compile successfully?

```
#define N 100
int foo(int arr[][N], int i, int j) {
   return arr[i][j];
}
```


CS 4400—Lecture 8

Dynamic Memory Allocation

- For allocation of memory at run time, library routine `malloc` is used.
  - arguments specify number of bytes to be allocated
  - return value is a pointer to the allocated memory or `NULL`
- `malloc` allocates one contiguous block (of specified size).

```
NODE* head = malloc(sizeof(NODE));  // implicit
head->next = malloc(sizeof(NODE));  // cast
```

- To release dynamically-allocated memory, the library routine free is used.
  - argument is the pointer to the block of memory to be released

```
free(ptr);
```

Question

Suppose we have

```
short* arr = malloc(user_input*sizeof(short));
```

Which of the following references the second element?

A. `arr[1]`
B. `*(arr+1)`
C. `*(arr+2)`
D. exactly 2 of the above
E. all of A-C
F. none of A-C

Question

Suppose we have

```
short* matrix = malloc(N*N*sizeof(short));
```

Which of the following references the element in the second row and second column?

A. `arr[1]`
B. `arr[N]`
C. `arr[N+1]`
D. `arr[1][1]`
E. none of the above

Exercise: Compiler Optimizations

```
#define N 16
typedef int fix_matrix[N][N];

void fix_set_diag(fix_matrix A,
                       int val) {
  int i;
  for (i = 0; i < N; i++)
    A[i][i] = val;
}
```

```
   movl 8(%ebp),%ecx
   movl 12(%ebp),%edx
   movl $0,%eax
.L14:
   movl %edx,(%ecx,%eax)
   addl $68,%eax
   cmpl $1088,%eax
   jne .L14
```

- Write a function fix_set_diag_opt that uses optimizations similar to those in the assembly code. Do not assume that N is 16.