Strange Referencing Examples

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

### Reference | Address | Value | Guaranteed?
--- | --- | --- | ---
`pgh[3][3]` | $76+20\times3+4\times3 = 148$ | 2 | Yes
`pgh[2][5]` | $76+20\times2+4\times5 = 136$ | 1 | Yes
`pgh[2][-1]` | $76+20\times2+4\times-1 = 112$ | 3 | Yes
`pgh[4][-1]` | $76+20\times4+4\times-1 = 152$ | 1 | Yes
`pgh[0][19]` | $76+20\times0+4\times19 = 152$ | 1 | Yes
`pgh[0][-1]` | $76+20\times0+4\times-1 = 72$ | ?? | No

```c
zip_dig
pgh[4];
```
Using Nested Arrays

• Strengths
  – C compiler handles doubly subscripted arrays
  – Generates very efficient code
  – Avoids multiply in index computation
  – (See figure 3.28)

• Limitation
  – Only works for fixed array size

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

/* Compute element i,k of fixed matrix product */
int fix_produ_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```

```plaintext
a

\[ \begin{pmatrix}
      \text{i-th row} \\
      \end{pmatrix}
\]

b

\[ \begin{pmatrix}
      \text{j-th column} \\
      \end{pmatrix}
\]
```
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);
}
```

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

```
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

• Without Optimizations
  – Multiplies: 3
    • 2 for subscripts
    • 1 for data
  – Adds: 4
    • 2 for array indexing
    • 1 for loop index
    • 1 for data

```c
/* Compute element i,k of variable matrix product */
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;
}
```
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

- **Compiler can optimize regular access patterns**

```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;
}
```
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**

```
# %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;
};
```

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

```c
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];
}
```

```
# %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
```