Arrays

Young W. Lim

2017-02-07 Tue

Outline

Based on

"Self-service Linux: Mastering the Art of Problem Determination", Mark Wilding
"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Array Declaration (1)

- T A[N]
- allocation of contiguous region of NL bytes
- L : the byte size of the data type T
- x_A denote the starting address of the region
- index between 0 and N-1
- the i-th element is at x_A + iL

Array Declaration (2)

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

- $x_A + i$
- $x_B + 4i$
- $x_C + 8i$
- $x_D + 4i$

Array Declaration (3)

```
int     E[12];
```

- E[i] access
  - %edx : the starting address of E
  - %ecx : the index value i

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

- (%edx + %ecx)
- $x_E + 4i$

Pointer Arithmetic (1)

- leal - to generate and address
- movl - to reference memory

```
movl %edx, %eax                 ; (int *) E
```

- x_E

```
movl %(%edx),%eax               ; (int) E[0]
```

- M[x_E]

```
movl (%edx,%ecx,4), %eax        ; (int) E[i]
```

- M[x_E+4i]

Pointer Arithmetic (2)

```
leal 8(%edx),%eax              ; (int *) &E[2]
```

- $x_E + 8$

```
leal -4(%edx,%ecx,4),%eax      ; (int *) E+i-1
```

- $x_E + 4i - 4$

```
movl(%edx,%ecx,8),%eax         ; (int) *(&E[i]+i)
```

- $x_E + 8i$

```
movl %ecx,%eax                 ; (int) &E[i]-E
```

- i

Array and Loop (1)

```
int decimal5(int* x) {
  int i;
  int val=0;

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

```
int decimal5_opt(int *x) {
  int val = 0;
  int *xend = x + 4;

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

  return val;
}
```

Array and Loop (2)

```
  movl 8(%ebp), %ecx
  xorl %eax, %eax
  leal 16(%ecx), %ebx
.L12:
  leal (%eax,%eax,4), %edx
  movl (%eax,%edx,2), %eax
  leal (%eax,%edx,2), %eax
  addl $4, %ecx
  cmpl %ebx, %ecx
  jbe .L12
```

```
int decimal5_opt(int *x) {
  int val = 0;
  int *xend = x + 4;

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

  return val;
}
```

Nested Array

```
int A[4][3]
```

```
typedef int Row[3];  // Row type definition
Row A[4];            // A is an array of Row type
```

2-D Array (1)

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

  . . .

fmatrix A;  // int A[16][16]
fmatrix B:  // int B[16][16]
```

2-D Array (2)

```
int iprod(fmatrix A,
fmatrix 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;
}
```

```
int iprod_opt(fmatrix A,
fmatrix B, int i, int k) {
  int *Ap = &A[i][0];
  int *Bp = &B[0][k];
  int result = 0;

  do {
    result += (*Ap) * (*Bp);
    Ap += 1;
    Bp += N;
    cnt--;
  } while (cnt >= 0);
  return result;
}
```

2-D Array (3)

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

```
int iprod_opt(fmatrix A,
fmatrix B, int i, int k) {
  int *Ap = &A[i][0];
  int *Bp = &B[0][k];
  int result = 0;

  do {
    result += (*Ap) * (*Bp);
    Ap += 1;
    Bp += N;
    cnt--;
  } while (cnt >= 0);
  return result;
}
```

Fixed Size Array (1)

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

int FProd(FM A, FM B, int i, int k)
{
  int j;
  int result = 0;

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

  return result;
}
```

```
int FProd_opt(FM A, FM B, int i
{
    int *Ap = &A[i][0];
    int *Bp = &B[0][k];
    int cnt = N-1;
    int result = 0;

    do {
      result += (*Ap) * (*Bp);
      Ap += 1;
      Bp += N;
      cnt--;
    } while (cnt>=0);
    return result;
}
```

Fixed Size Array (2)

```
int FProd_opt(FM A, FM B, int i
{
    int *Ap = &A[i][0];
    int *Bp = &B[0][k];
    int cnt = N-1;
    int result = 0;

    do {
      result += (*Ap) * (*Bp);
      Ap += 1;
      Bp += N;
      cnt--;
    } while (cnt>=0);
    return result;
}
```

Dynamically Allocated Array (1)

```
typedef int *vmatrix;
. . .
vmatrix A  // int *A;
. . .
vmatrix func(int n) {
  return (vmatrix)
    calloc(
      sizeof(int), n*n);
}
. . .
int foo(vmatrix A, int i,
int j, int n) {
  return A[(i*n) + j);
}
```

```
movl 8(%ebp), %edx
movl 12(%ebp), %eax
imull 20(%ebp), %eax
addl 16(%ebp), %eax
movl (%eax, %eax, 4), %eax
```

Dynamically Allocated Array (2)

```
typedef int *vmatrix;

int vprod(vmatrix A,
vmatrix 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;
}
```

```
int vprod(vmatrix A,
vmatrix B, int i,
int k, int n) {
  int *Ap = &A[i*n];
  int nT = n, result = 0;

  if (n <= 0) return result;
  do {
    result += (*Ap) * B[nT];
    Ap++;
    nT+= n;
    cnt--;
  } while (cnt);
  return result;
}
```

Dynamically Allocated Array (3)

```
.L37:
  movl 12(%ebp), %eax
  movl (%ebx), %edi
  addl $4, %ebx
  imull (%eax,%ecx,4), %edi
  addl %edi, %esi
  addl 24(%ebp), %ecx
  decl %edx
  jnz .L37
```

```
int vprod(vmatrix A,
vmatrix B, int i,
int k, int n) {
  int *Ap = &A[i*n];
  int nT = n, result = 0;

  if (n <= 0) return result;
  do {
    result += (*Ap) * B[nT];
    Ap++;
    nT+= n;
    cnt--;
  } while (cnt);
  return result;
}
```