CIS 450
Computer Architecture and Organization

Lecture 12: Pointers and Arrays

Mitch Neilsen
neilsen@ksu.edu

219D Nichols Hall

Arrays

Familiar definition: type name[size]

Familiar usage: name[index]

Differences from Java:

- name is not an object
- name is like a constant “pointer”

Array index lookups are not bounds checked: name[size+10000] isn’t an error!

Address cannot be changed at run-time because name is a constant integer address value

Arrays

Arrays are not objects

Arrays have a:

- **Type ([#] of T)**
- **Size (sizeof(T)*size)**
- **Base address (name)**

Other Array Operations

***array : the value of the first element in the array**

(array + i) : adjust the base address to a new address pointing to the (i+1)th element (indexing starts at 0)

***(array + i) : the value of the (i+1)th element in the array**

array[i] = *(array + i)

Array Operations

x[i] = *(x + i)

x[i] is "syntactic sugar": looks nicer but offers no additional functionality

(x + i) is called "pointer arithmetic"

* is the dereference operator and means: "take next type of" (possibly changing the expression value too)

Dereferencing

The dereference operator * is sometimes called the "contents of" operator which strictly speaking is incorrect.

Mr. T says, "Don't fall into this trap, sucka. Don't be a foo'."

Array Example

- **Walk through array1.cpp**

Example: Array1.cpp

```
..
const int size = 4;

int main()
{
        int x[size];
        char ch;
        for (int i = 0; i < size; ++i)
            x[i] = i + 1;
        cout << "sizeof(x) = " << sizeof(x) << endl;
        cout << "sizeof(int) = " << sizeof(int) << endl;
        for (int i = 0; i < size; ++i)
            cout << "x[" << i << "] = " << x[i] << endl;
        cout << "x = " << x << endl;
        for (int i = 0; i < size; ++i)
            cout << "x + " << i << " = " << x + i << endl;
        …
```

```
sizeof(x) = 16
sizeof(int) = 4
x[0] = 1
x[1] = 2
x[2] = 3
x[3] = 4
x = 0xbfd28b10
x + 0 = 0xbfd28b10
x + 1 = 0xbfd28b14
x + 2 = 0xbfd28b18
x + 3 = 0xbfd28b1c
```

Example: Array1.cpp (cont.)

```
for (int i = 0; i < size; ++i) {
    cout << "*(x + " << i << ") = " << *(x + i) << endl;
 }
 cout << "*x = " << *x << endl;
 cout << "offset x = " << reinterpret_cast<int>(x + 1)
    - reinterpret_cast<int>(x) << endl;

 return 0;
```

```
*(x + 0) = 1
*(x + 1) = 2
*(x + 2) = 3
*(x + 3) = 4
*x = 1
offset x = 4
```

Multidimensional Arrays

Again, definition is similar to single dimensional array:

- `type name[size1][size2]...[sizeN]`

Same operations, only more abstraction:

- `x[i][j][k] = *(*(*(x + i) + j) + k)`
- `*(x + i)`
- ***NOTE:* "*" is *not* "contents of"**
- *** here means take away one type**

Pointer Math Types

```
int x[2][3][5];
```

Types:

- x : [2] of [3] of [5] of int
- *x : [3] of [5] of int
- **x : [5] of int
- ***x : int (only last * changes the value!)

Pointer Math Sizes

Learn this early! Your success depends on it!

`int x[10];`

- sizeof(x) == 40
- sizeof(*x) == 4

`int x[2][3][5];`

- sizeof(x) == 120
- sizeof(*x) == 60
- sizeof(**x) == 20
- sizeof(***x) == 4

Pointer Math Offsets

```
int x[2][3][5];
```

Offset size:

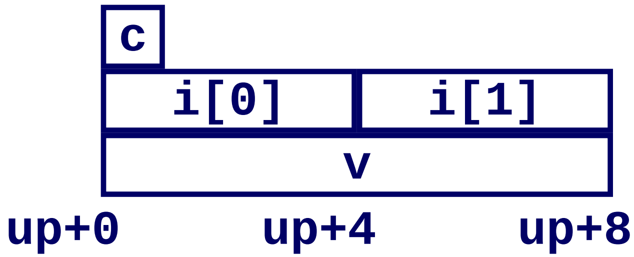
- **Offset x: 60**
- **Offset *x: 20**
- **Offset **x: 4**
- **Offset ***x: No such thing...**

Union Allocation

Principles

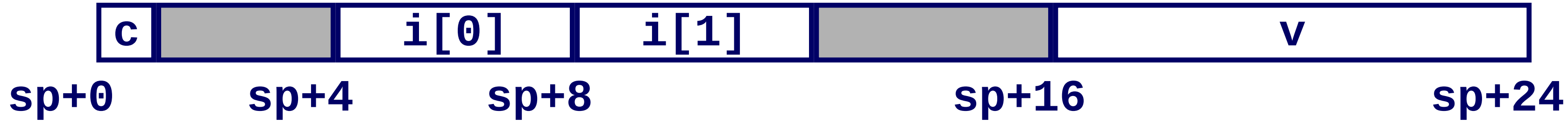
- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

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



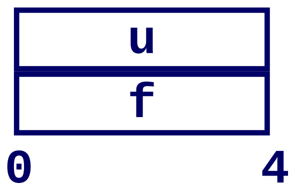

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

(Windows alignment)



Using Union to Access Bit Patterns

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



- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`

```
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;
}
```

Byte Ordering Revisited

Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Most significant byte has lowest address
- PowerPC, Sparc

Little Endian

- Least significant byte has lowest address
- Intel x86

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

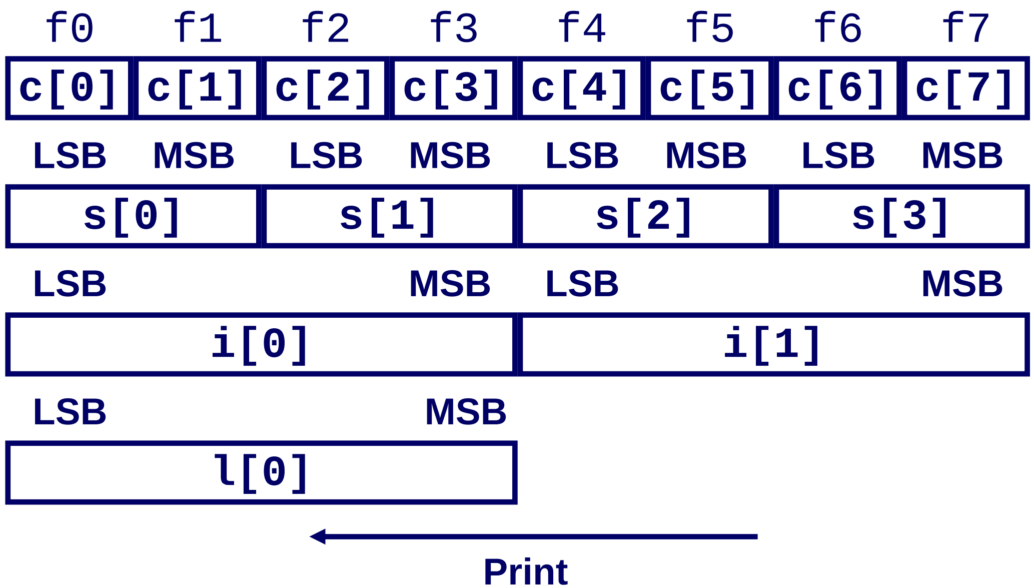
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

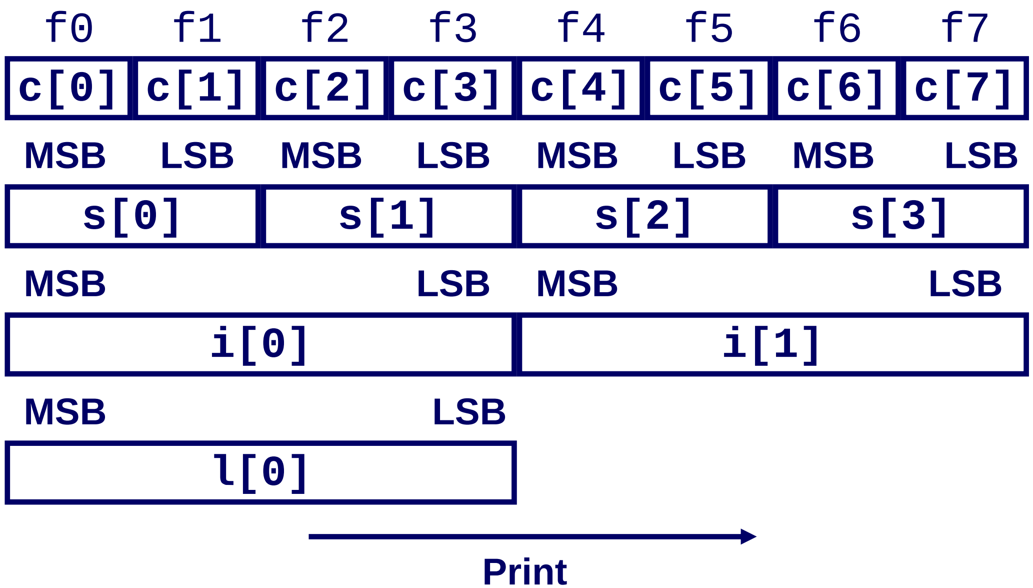


Output on IA32:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian

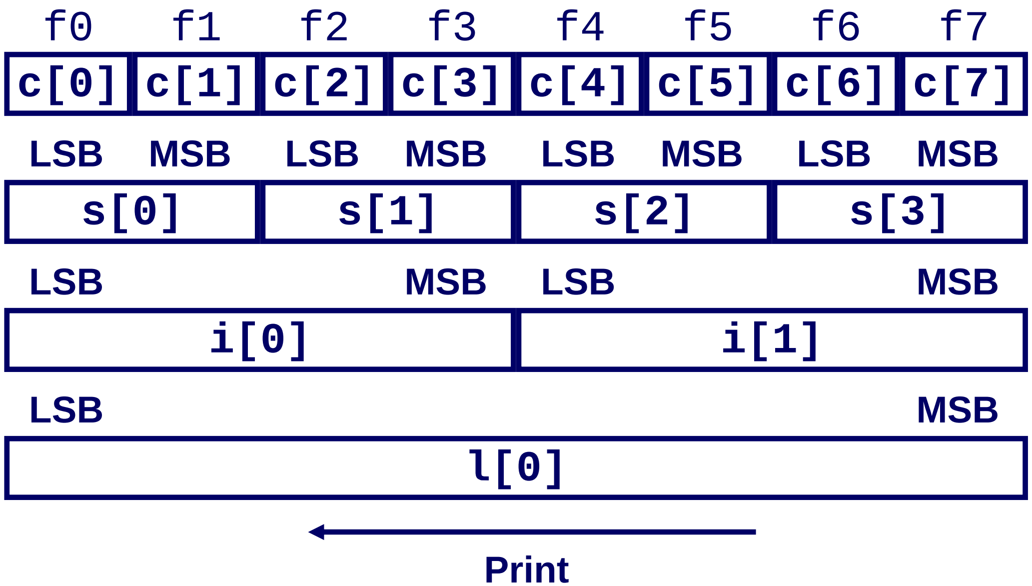


Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Summary

Arrays in C

- Contiguous allocation of memory
- 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

Quiz #1 - Review