What about arrays?

- Arrays are declared:
  ```
  int arr[8];
  ```
  - This declares an array of 8 integers
  - Memory allocated for it, but not initialized

- Arrays are referenced:
  ```
  arr[4] = 5;
  ```
  ```
  z = arr[4];
  ```
  - No bounds checking; arr[51] won’t signal an error
  - Not easy to find the length of an array
Arrays and pointers

- Arrays are implemented with pointers
  - \&(arr[0]) == arr == address of the first element of the array
  - arr[j] == *(arr + j)
- Any integer can be added to a pointer
  - Negative ones, too!
  - As before, C doesn’t check to see if it looks OK

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Accessing arrays

- Common to access arrays in loops
- Two forms of loops are common
  - Array index changes
  - Pointer to element changes (incremented)
- Both forms are OK (and equivalent)

```c
int arr[50];
int j;
int *p;
int sum = 0;

for (j = 0; j < 50; j++) {
    sum += arr[j];
}

for (p = arr; p < (arr+50); p++) {
    sum += *p;
}
```
Strings in C

- Strings in C are simply arrays of characters
  - Strings must be terminated by NUL (\0)
  - This makes it impossible to include \0 in a string…
- Since arrays can’t be resized, nor can strings!
  - They can be copied onto one another using functions like strcpy()
- Strings can be declared like this:
  - char s1[50];
  - char s2[50] = “Hello!”;
  - Now, s2 is an array of 50 bytes, with the first 7 initialized
  - char *s3 = “testing”;
  - Now, s3 points to an array of at least 8 characters (including the trailing \0)
  - Some smart compilers may share storage for multiple strings initialized to “test”…
  - Don’t modify strings initialized the third way!

Strings in C

- A structure is a mechanism for grouping values together
  - Similar to class in Java
  - No associated methods!
  - May contain builtin types as well as pointers and other structures
- Refer to elements of a structure with “.” notation
  - Different notation for referring to elements of a structure being pointed to
- Useful for building complex data structures
  - Pointers may refer to memory used for a structure
  - How is memory allocated?

```
sstruct treenode {
    int strLength;
    char s[8];
    struct treenode *l;
    struct treenode *r;
};
```

```
struct treenode root;
root.strLength = 5;
// Copy 5 characters into s
bcopy (value, root.s, 5);
```
Pointers and structures

- Pointers may point at structures
  - p = &root;
- Elements referred to with “->” notation
  - (*p).strLength is equivalent to p->strLength
- How is space allocated?
  - Library call malloc
  - Compute necessary memory size using sizeof function
    - Works on built-in types, too
    - May modify the value

```c
struct treenode {  
  int strLength;
  char s[8];
  struct treenode *l;
  struct treenode *r;
};

struct treenode root;
struct treenode *p;
root.strLength = 5;
// Copy 5 characters into s
bcopy (value, root.s, 5);
p->strLength = 6;
bcopy (nValue, p->s, 6);
nodep = (struct treenode *)malloc (sizeof (struct
treenode));
root.l = nodep;
```

Passing parameters

- Built-in types (int, pointers, FP) are passed by value
  - Function cannot change the value in the caller
  - Functions may change the value a pointer points at
- Arrays and structures must be passed by reference
  - Pass a pointer to the array or structure
  - Pointer doesn’t change
  - Array or structure may change

```c
void addOneWrong (int x) {
  x = x+1;
}
void addOneRight (int *x) {
  *x = *x +1;
}

int main () {
  int value = 5;
  addOneWrong (value);
  printf (“Value is %d\n”,
    value);
  addOneRight (&value);
  printf (“Value is %d\n”,
    value);
  return (0);
}
```

Value is 5
Value is 6