Lecture 9

CSE 110

14 July 1992

1 More about Object Contexts and Value Contexts

If you look back at the notes for lecture 3, section 3, and 3.2 in particular, you'll recall the following facts:

- The word *object* is short for the phrase "a particular part of the computer's memory."
- Some expressions, such as `x`, refer to objects. (For example, `x` refers to the variable `x`, which is a particular part of the computer's memory and is therefore an object.) Most expressions, such as `4` or `3*x + y`, do not refer to objects.
- An expression which refers to an object is called an *lvalue expression*.[1]
- A *context* is a place in your program where you are allowed to put an expression.
- When the computer encounters a certain expression in one context, it may behave differently than when it encounters the same expression in another context.
- There are only two kinds of contexts in C: *object contexts* and *value contexts*. What the computer does with an expression depends only on the expression itself and on whether that expression is in an object context or a value context.
- Most contexts are value contexts.

[1] The 'l' in 'lvalue' stands for 'left', because the thing on the **left** side of an assignment must be an lvalue expression.

- The only contexts that are not value contexts are:

  1. The left-hand-side of an assignment expression, and
  2. The operand of the `&` operator.

  These two contexts are object contexts, not value contexts.

- Any expression may appear in a value context.
- An expression in a value context is evaluated to produce a value.
- Only an lvalue expression may appear in an object context.
- An lvalue expression in an object context is not evaluated to produce a value. Instead, the computer figures out what object it refers to, and then

  1. If the lvalue expression was on the left-hand-side of an assignment, the computer uses the object it represents as a place to store the value it computed on the right-hand-side of the assignment, or
  2. If the lvalue expression was the operand of the `&` operator, then the computer gets the address of the object that the lvalue expression represents.

1.1 A Pointer Value Points to an Object

If p is a valid expression whose type is <pointer to something>, then p points to an object of type <something>. Then $*p$ is an expression which represents the object that p is pointing to. Since $*p$ therefore refers to an object, $*p$ is an lvalue expression, and may appear on the left-hand side of an assignment statement, as in

```
*n1 = temp ;
```

. `temp` is in a value context, so it's evaluated, and the computer gets the value stored in the variable `temp`. On the other hand, `*n1` is in an object context, and so once the computer finds out what part of the blackboard `*n1` represents, it does *not* go get the value stored there. Rather, it uses it as a place to store the value of `temp`. On the other hand, in

```
temp = *n1 ;
```

the `*n1` is in a value context; once the computer figures out what region of the blackboard `*n1` represents, it goes and gets the value stored there, and then it stores that value into the region of the blackboard represented by `temp`. `temp` is in an object context, not a value context, so it is not evaluated and the computer does never retrieves the value stored in `temp`.

1.2 The operand of `&` is in an Object Context

We have seen exactly two examples of object contexts so far. One is the left-hand side of an assignment.

The other is that in the expression `&`*foo*, *foo* is in an object context: It is not evaluated to produce a value, and the value of the entire `&`*foo* expression has nothing whatever to do with the value actually stored in *foo* itself. Also, *foo* must be an lvalue expression, because `&` asks where in memory its operand is stored, so therefore its operand must be an object, and only lvalue expression, by definition, represent objects.

1.3 So What?

This is going to become frightfully important tomorrow. In the meantime, we'll do something else.

2 Arrays

Suppose we want to write a program which reads in a list of 100 numbers from the user, and then prints them out in reverse order. To do that, we must remember all 100 numbers until the end; there's no way around it. We could write a program with a very long declaration:

```
int main(void)
{
  int n0, n1, n2, n3, n4, n5, n6, n7, n8, n9;
  . . .
  int n90, n91, n92, n93, n94, n95, n96, n97, n98, n99;

  . . .
}
```

and then use 100 `scanf`'s to read them in and 100 `printf`'s to write them out backwards, but of course there has to be a better way.

2.1 A Better Way

If we write

```
int n[100];
```

instead, the compiler creates an *array* of 100 <`int`>s for us. That means that it finds enough space for 100 <`int`> variables, reserves the space, and arranges that `n` refers to that space. The space is all contiguous, meaning that if an <`int`> is 2 bytes long, the compiler will find 200 bytes of space all in the same place—it won't find us 37 bytes here and 22 bytes there and 41 bytes somewhere else.

We can ask for these 100 <`int`>s individually: The expression `n[0]` refers to the first one, and `n[1]` to the second one, and soforth, up to the last one which is `n[99]`. These 100 <`int`> variables are called the *elements* of the array; each element has an *index* which says where it is in the array. The index of the first element is 0, the index of the second is 1, and the index of the last one is 99. So here's our program:

2.2 Program to Read 100 Integers from the User and Write them out in Reverse Order

```
#include <stdio.h>

int main(void)
{
  int n[100], i;

  for (i=0; i<100; i++)
    scanf("%d", &(n[i]));

  for (i=99; i>=0; i--)
    printf("%d\n", n[i]);

  return 0;
}
```

2.3 Notes on the Program

In the first loop, `i` starts at 0 and goes up by ones until it gets to 99, the index number of the last element in the array `n`. Each time through the loop, the computer figures out what `n[i]` is (it's the `i`'th element of the array `n`), but, because the `n[i]` is in an object context, not a value context (it's the operand of the `&` operator), the computer does not go and get the value of `n[i]`. Instead, it computes the address of `n[i]`, and passes that address to `scanf`, which reads an <`int`> value from the user and stores the value in the variable `n[i]`. `scanf` could change the value of `n[i]` because it knew where on the blackboard `n[i]` was. `scanf` knew where on the blackboard `n[i]` was because we passed it the address of `n[i]`.

The first number the user enters gets stored in `n[0]`, and the second gets stored in `n[1]`, and soforth, until finally the user enters the hundredth number, which is stored in `n[99]`.

Then the computer executes the second loop. `i` starts at 99, the index number of the last element of the array `n`, and counts backwards until it gets to 0. After 0, the `for` loop is done. Each time through the loop, the computer figures out what `n[i]` is (it's the `i`'th element of the array `n`), and then, because the `n[i]` is in a value context, the computer gets the value of `n[i]` and passes that value to `printf` as an argument. `printf` then prints out the value of `n[i]`.

Since `i` starts at 99, the first value the computer prints out is that of `n[99]`, which was the last number the user entered. Then the computer decrements `i`, runs through the loop again, and prints out the value of `n[98]`, whose value is the next-to-last number the user entered. The computer keeps running backwards through the array, until finally `i` is 0; then the computer prints out the value of `n[0]`, which is the first number the user entered. Then the computer decrements `i` to -1, exits the loop, and quits the program.

2.4 Initializing Arrays

When you declare an ordinary variable, you can initialize it thus:

```
int n = 57;
```

This means that `n` starts out with the value 57; the 57 here is called an *initializer*. We can do a similar thing with an array, only we have to specify more than one initializer: If we write

```
int n[5] = { 1, 3, 9, 27, 81} ;
```

then `n[0]` starts out with the value 1, `n[1]` starts out with the value 3, `n[2]` starts out with the value 9, `n[3]` starts out with the value 27, and `n[4]` starts out with the value 81.

If you make the initializer list too short for the array, as in

```
int n[5] = { 1, 3 };
```

the computer initializes the elements of the array until it runs out of initializers, and initializes the rest of the elements of the array with zero values. So the example above is the same as if we had written `int n[5] = { 1, 3, 0, 0, 0};`.

If you make the initializer list too long, the compiler will complain.

If you include an initializer, you can omit the array size:

```
int n[] = { 1, 3, 5};
```

The compiler counts the number of initializers and makes an array that is just big enough to hold all of them. In this case it makes an array with 3 elements because you specified 3 initializers.

2.5 Character Arrays

Of course you can have an array of variables of any type, including <`char`>s. Here's a program to print someone's name out backwards:

```
#include <stdio.h>

int main(void)
{
  char name[] = { 'J', 'e', 'a', 'n', ' ', 'O', 'g', 'r', 'i', 'n', 'z' };
  int i;

  for (i=10; i>=0; i--)
    printf("%c", name[i]);

  printf("\n");
  return 0;
}
```

Note that we use `%c` to print out a <`char`> with `printf`. Also note we had to hard-wire the length of the character array into the loop. We'll learn how to get around that tomorrow.

It's a pain having to type all those character constants, and C gives us a shortcut: We're allowed to initialize a character array this way: Instead of writing

```
char name[] = { 'J', 'e', 'a', 'n', ' ', 'O', 'g', 'r', 'i', 'n', 'z' };
```

we can do this:

```
char name[] = "Jean Ogrinz";
```

. The shorthand `"Jean Ogrinz"` denotes a special array of characters, called a *string of characters* or usually just a *string*. Strings are how we handle things like peoples' names in C. We'll see a lot of them in the rest of the course.

In case you hadn't made the connection yet, the first argument to `printf` is always a string.