Lecture 19

CSE 110

5 August 1992

> You taught me Language, and my profit on't
> is I know how to curse.
> — William Shakspere, *The Tempest*, I, ii.

1 Left-Over Language Features

Today was the day we saw the last of the language features, things that are rarely or never important. Each of these I expected to bring up when the appropriate time came, at some moment when it would be useful, and the moment never came.

We'll see these features in increasing order of usefulness.

1.1 The , Operator

The `,` operator takes two operands, which must be expressions. To evaluate an expression of the form *expr1* `,` *expr2*, the computer evaluates *expr1*, and then evaluates *expr2*. The value of the entire expression is the value of *expr2*.

The `,` operator has two interesting properties: It guarantees that *expr1* will be evaluated first[1], and it guarantees a sequence point between the evaluation of the two expressions. This means that any side effects such as increments or assignments that are scheduled as a result of evaluating *expr1* will be completely resolved before *expr2* is evaluated.

Nevertheless, the `,` operator is mostly used when you want to stick two expressions where only one normally fits, such as in the condition of a `while`

[1] Most operators, such as `+`, do not guarantee which of their operands will be evaluated first.

statement. The example that Kernighan and Ritchie give for the `,` operator is a program to reverse a string in place, which we've already seen: (They called it `reverse`.)

```
void strrev(char *s)
{
  int c, i, j;

  for (i=0, j=strlen(s)-1;  i<j;  i++, j++) {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}
```

The commas that separate a function's arguments, the variables in a declaration, and soforth, are *not* `,` operators.

1.2 The `?:` Operator

The `?:` is unusual in that it has *three* operands. To evaluate the expression *expr1* `?` *expr2* `:` *expr3*, the computer first evaluates *expr1*. If *expr1* is true, the computer evaluates *expr2* and that value is the value of the entire expression. Otherwise, the computer evaluates *expr3* and that value is the value of the entire expression.

A `?:` expression is like a miniature `if`–`then`. Here's an example: This function prints out the elements of an array, separated by spaces, with ten elements per line.

```
void print_array(int *a, int nelems)
{
  int i;

  for (i=0; i<nelems; i++)
    printf("%d%c", a[i], (i+1)%10 ? ' ' : '\n');
}
```

Here's a simpler example: It sets the value of `a` to the value of either `x` or `y`, whichever is smaller:

```
a = (x > y) ? y : x ;
```

I swiped this example from K&R also.

1.3 The `switch` Statement

A `switch` *statement* has this form:

```
switch (expression) {
  case value1:
    statement11
    statement12
    ...
  case value2:
    statement21
    statement22
    ...
  ...
  default:
    statementd1
    statementd2
    ...
}
```

To execute a `switch` statement, the computer evaluates the *expression*. Control then passes to the statement immediately after one of the three following things, which are listed in order of preference:

1. A *case label* in the *statement*, which has the form

   `case` *constant* `:`

   where the value of the *constant* is the same as the value of the *expression*. If there is no case label whose value matches that of the *expression*, control passes to the statement immediately after:

2. A `default` *label* in the *statement*, which has the form

   `default :`

   . If there is no `default` *label*, control passes to the statement immediately after:

3. The end of the `case` statement.

Once into the compound statement part of the `switch`, control flows from top to bottom as usual. Control does *not* leave the compound statement except by flowing off the bottom of the statement or by encountering a `break` or `return` as usual. In particular, after the computer executes the last statement in one of the `case` sections, it continues on to the statements under the next `case` section. This usually isn't what we want, and so we almost always end each `case` section with a `break` statement.

It's so rare to actually allow control to fall through from one `case` section to the next that you should always put in a comment when you do let control fall through.

`case` can always be replaced by `if`–`else if`–`else`, but the restrictions on it allow the compiler to generate more efficient code, particularly if there are a lot of `case` labels.

1.4 Block-Scoped variables

We've only seen variables declared at the head of a function. These variables had names that were known only inside the function in which they were declared, and the variables were created each time the function was called and destroyed again when the function returned.

In fact, you can declare new variables at the head of *any* compound statement. The variables are created when control enters the statement, and destroyed again when control leaves the statement. The names of the variables are known only within the statement.

It's good to declare variables as belonging to as small a compound statement as possible, because it keeps the definition and use of the variable close together, because it restricts communication, and because it's easier to understand when you're reading the code—you know you can forget about the variable's value unless you're reading the code for the compound statement in which it's declared.

We've actually seen this before: I used it in the sample solution to the guessing game project.

1.5 true and false

We've been saying all semester that **false** was a zero value and that **true** was a nonzero value. That's only part of the story.

In any condition context, such as the condition part of a `while` or `for` loop, or the first operand of a `?:` expression, the following values mean **true** and **false**:

Type	**false**	**true**
number	0	non-zero
pointer	`NULL`	not `NULL`
character	`'\0'`	anything else

This means, in particular, that all those times we wrote `if (c != '\0')` we could have just written `if (c)`, and whenever we wrote `if (newnode != NULL)` we could just have written `if (newnode)` instead.

Opinion seems to be divided on whether or not this is bad style. Since opinion is divided, and about half the world does in fact use the short forms, it is therefore good style.

2 The Compiler and the Linker

The compiler is really two programs in one. The first part, the *compiler*, translates the C into whatever machine language is appropriate for the machine you're using. The translated version of the program is called an *object file*.

The second part of the compiler, the *linker*, does a simpler job. It looks through the object file for function calls, and it arranges that the right functions are called. When it's done, it might discover that you called some functions for which you didn't provide code, such as `printf`; these are called *unresolved references*. The linker then looks through *library files*, which contain pre-compiled functions, for functions whose names match the unresolved references. If it finds any, it includes the object code for those functions into your program. If there are still unresolved references, the linker complains that it couldn't finish compiling your program; otherwise it has resolved all the references and creates an executable program.

Having a linker means that you can write your program in several different files, compile them separately, months apart, have functions in one file call functions in another, and the linker will sort out all the calls at the end. If you change one function, you only need to recompile the file it's in and then re-link the program; you don't need to recompile the unchanged source files.

We didn't see how to do this because we never wrote a program long enough to deserve it.

3 Idle Questions

We had a lot of idle questions about what 'object-oriented' means; how long it takes to write a big program and how big one is; how one organizes a big programming project, features of other languages, how widespread various languages are, and soforth. I can't possibly remember all the nonsense we discussed, but I do remember that there was one point I wanted to make that I didn't get a chance to bring up: