Lecture 5

CSE 110

7 July 1992

1 More About `if`

1.1 `else`

Consider this code:

```
if ( x > 0 )
  printf("X has a positive value\n");
if ( ! (x > 0) )
  printf("X does not have a positive value\n");
```

Suppose the value of `x` is 0. We know that exactly one of the `printf` statements will always be executed, and so we know that if it isn't the first one, it must be the second. So as smart humans, we don't have to perform both tests. The computer, on the other hand, is dumber, and we have to tell it explicitly that the two `if` statements here represent an exclusive partitioning of all possible situations. We do that with an `else` clause:

```
if ( x > 0 )
  printf("X has a positive value\n");
else
  printf("X does not have a positive value\n");
```

This tells the computer to compute the value of the expression `x > 0`, and if that expression is true, then to print `X has a positive value`. Otherwise, the computer prints `X does not have a positive value`. Exactly one of the `printf` statements is ever executed each time through this code.

When we want to select one of many conditions, we use a similar construc-

tion:

```
if ( profits < 0  )
  printf("The company is losing money\n");
else if ( profits == 0 )
  printf("The company exactly broke even\n");
else
  printf("The company turned a profit\n");
```

The computer evaluates the conditions, one at a time, until it finds one that is true. Then it executes the associated statement, and then skips all the rest of the clauses. If none of the conditions are true, the computer executes the statement associated with the `else` clause, if there is one.

1.2 Nested if–else Statements

Here's some code for printing out information about average test grades: `total` holds the sum of all the grades in the class, and `count` holds the number of students in the class.

```
if (count != 0)
  if ( total/count < 80 )
    printf("This class is doing badly\n");
  else
    printf("This class is not doing too badly\n");
else
  printf("There are no students in this class\n");
```

An `if` construction or an `if`–`else` construction is a statement, and can go anywhere a simple or a compound statement can. Here, we first check to see if `count` is not 0; if it is 0, we print `There are no students in this class`. Otherwise, we compute whether or not the average grade is below 80 and print a message depending on whether it is or not. This second `if`–`else` statement is said to be *nested within* the first.

We indent the nested statement to make our meaning clearer.

No suppose the final `else` and the last `printf` weren't in the example above. How does the compiler know that we meant what we did, rather than

```
if (count != 0)
  if ( total/count < 80 )
    printf("This class is doing badly\n");
else
  printf("This class is not doing too badly\n");
```

, which would print `This class is not doing too badly` whenever `count` was equal to 0.

If you said the compiler knows the difference because of the indentation, you're mistaken: the compiler ignores all white space completely. The indentation is only there for the benefit of human readers.

The C language resolves this ambiguaity by fiat: The rule is that if there's more than one `if` that an `else` could go with, it goes with the nearest one. So the `else` above matches with the second `if`, not the first. If for some reason you really wanted the meaning suggested by the second indentation, you could write this:

```
if (count != 0) {
  if ( total/count < 80 )
    printf("This class is doing badly\n");
} else
  printf("This class is not doing too badly\n");
```

The curly braces here delimit the statement that follows the first `if`. That statement doesn't include the `else` clause, so the compiler can't construe the `else` as being under control of the first `if`—it must be parallel to it.

2 Example Program: Solving Quadratic Equations

Now we'll write a program which lets the user enter a quadratic equation. If the roots of the equation are real, the computer will find them and print them out, and otherwise the computer will print a message saying that the roots are complex. We did this in class, so here's the code:

2.1 The Program

Program to solve quadratic equations $ax^2 + bx + c = 0$.
Known bugs: Fails when $a = 0$.

```
#include <stdio.h>
#include <math.h>

int main(void) {
   double a, b, c;
   double x1, x2;
   double discriminant;

   /* get inputs a, b, and c from user. */
   printf("Please input a, b, and c, one per line.\n");
   scanf("%lf", &a);
   scanf("%lf", &b);
   scanf("%lf", &c);

   discriminant = b*b - 4*a*c;

   if ( discriminant < 0 ) {
     printf("The roots are imaginary.  I can't find them.\n");
     return 1;
   } else {
     x1 = ( -b + sqrt(discriminant) )  / (2*a);
     x2 = ( -b - sqrt(discriminant) )  / (2*a);

     printf("The roots are %f and %f.\n", x1, x2);

     return 0;
   }

}
```

2.2 Notes About this Program

The program's small enough that we didn't bother with any functions other than `main`.

The declaration

```
double discriminant;
```

is analogous to the `int foo;` declarations we've seen, but it allocates enough space for a variable of type <`double`> instead of one of type <`int`>. A <`double`> is a double-precision floating-point number and can represent a fraction or a number much larger than an <`int`> can. We'll need that for this program.

We are going to need the `sqrt()` function, which takes one <`double`> argument and returns the <`double`> which is its square root. Someone else wrote the `sqrt()` function and put it in the library for us, so we don't have to supply it. But here's a problem: How does the compiler know that `sqrt()` takes a <`double`> argument and returns a <`double`> result? It needs to know, or else it won't be able to pass the argument or interpret the return value properly. The answer is that we must tell it. We `#include` the file `<math.h>`, which contains a line that tells the compiler what types the arguments and return value of `sqrt()` will have. Such a line is called a *prototype*; it looks just like a function header with no body:

```
double sqrt(double x);
```

This gives the compiler the information it needs to compile our calls to `sqrt()` corectly. Otherwise it would assume that `sqrt()` returned an <`int`> value, and all sorts of nastiness would ensue. We didn't have to write this, because the authors of `sqrt()` put it in `<math.h>` for us, so all we have to do is `#include` that file.

The `#include <stdio.h>` is there for a similar reason, to tell the compiler what kinds of arguments and return types `printf()` and `scanf()` have.

We haven't seen `scanf()` yet, and we'll talk about it in detail later. But it's the reverse of `printf()`: `printf()` prints data out, and `scanf()` reads data in. `scanf()`'s first argument, like `printf()`'s first argument, describes the format of input to be read, and the remaining arguments, rather than getting printed out, describe where to store the input values entered by the user. The `"%lf"` says that the input will be in the form of a **l**ong **f**loat, which is an obsolete synonym for a <`double`>. The `&a` is more interesting and important, and so it gets a section to itself.

2.3 Pointers and the & Operator

We need to tell `scanf()` where to store the value that the user enters. Say we want to store that value in the variable `a`. Now, `scanf()` can't normally do that

because it doesn't know where `a` is. If we wrote something like

```
scanf("\%lf", a);
```

that would be no good at all, because `scanf()` would get the *value* of `a` and would never find out what it really needs to know, which is where `a` actually is.

The `&` operator, called the *address-of* operator, says where a variable is. The expression `&a` yields a value of a special type, called a <pointer to `double`>, because it 'points to' a <`double`>—that is, it says where a certain <`double`> object can be found. So, in short, we are telling `scanf()` where to find the variable in which we want it to store its result. Later on, we'll see more of `&`, and of its complementary operation, `*`, which takes a pointer and finds to which it points. The main point here is that since we explicitly told `scanf()` where `a` is stored, `scanf()` can change the value of `a`.

The call

```
scanf("\%lf", &a);
```

instructs `scanf()` to read input from the keyboard, parsing and interpreting it as a floating-point value, and to store the value that that data represents into the space pointed to by `&a`.

Note the parentheses in the assignment expressions, to preserve clarity and also to get the compiler to do what we want.

Note that we use `%f` to get `printf()` to print out a <`double`> value, whereas we used `%d` to print out an <`int`>.