Lecture 4

CSE 110

6 July 1992

1 More Operators

C has many operators. Some of them, like `+`, are *binary*, which means that they require two operands, as in `4 + 5`. Others are *unary*, which means they require only one operand. We'll see an example of this in section 3.3.

1.1 Arithmetic

Arithmetic operators include `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. Division is a little odd: Its semantics change depending on the types of its operands. If both operands are <`int`>s, then `/` represents *integer division*, in which the fractional part of the result is discarded. For example, the value of the expression `13/5` is 2.

`%` denotes the *modulus* operator: If a and b are integer expressions, then a`%`b is the remainder when a is divided by b. For example, the value of the expression `13 % 5` is `3`, because 3 is the remainder when you divide 13 by 5. If one of the operands is an expression whose value isn't of integer type, that's an error and the compiler won't compile the program.

1.2 Assignment

`+=` is an assignment operator. Like `=`, its left operand must be an lvalue. `x += 2` means the same thing as `x = x + 2`. It's more natural to think "Add 2 to `x`" and to write `x += 2;` than it is to think "Get `x`, add 2, and put it back." and write `x = x + 2;`. Furthermore, in an expression like

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are the same, or to wonder why they're not.[1]

Similarly, there are `-=`, `*=`, `/=`, `%=` operators, all defined analogously.

1.3 Increment and Decrement

Adding one to something is such a common operation that there's a special name for it and yet another notation for it. Adding 1 to a quantity is called *incrementing* it. There's a special increment operator, `++`. If we write `x++` or `++x` in an expression, then, sometime before the computer executes the next statement, it will add 1 to the value stored at `x`.[2]

The values of `x++` and `++x` differ, however: if `x` is 12, then the value of `x++` is 12 and the value of `++x` is 13. The notion is that if you use `++x`, the compiler increments `x` before getting its value, and if you use `x++`, the compiler increments `x` after geting its value. So if the value of `x` is 119, then after

```
y = x++ ;
```

`y` would be 119 and `x` would be 120, but after

```
y = ++x ;
```

`y` would be 120 and `x` would be 120.

When the compiler actually chooses to do the increment is None of Your Business, as long as the increment happens before the next statement.[3]

Question: What happens if `x` is 119, and we do:

```
y = x++ + x++ ;
```

? Depending on when the incrementing actually happens, `y` might get 238, or 239, right? Wrong. For various technical reasons the standard says that if you try to modify the same object twice in one statement,[4] you get undefined behavior—the compiler is allowed to do whatever it likes, including nothing, printing a warning, erasing itself, or teleporting elephants into the room. Similarly

[1] I swiped this example from *The C Programming Language*, by Kernighan and Ritchie.

[2] Of course, when we say "the value stored at `x`," we are implying that `x` is an lvalue—you can't do `4++` to increment the value of 4; the compiler will complain.

[3] Actually it has to happen before the next *sequence point*. We know about two kinds of sequence points: semicolons are sequence points, and also there is a sequence point just before every function call.

[4] Really it says twice without a sequence point in between.

```
x = x++ ;
```

is undefined. On the other hand, if `x` is 2, then

```
y = x * x++ ;
```

is perfectly legal, (`x` and `y` are each modified only once before the end of the statement) and might assign `y` the value 4 or the value 6, depending on whether the compiler does the increment before or after it evaluates the first `x`. In general it's best to avoid such situations.

There is a `--` operator, which is just like `++`, except that it subtracts 1 instead of adding 1. This is called *decrementing*.

1.4 Precedence

Consider this expression:

```
2 * 3 + 4
```

does the compiler do `2 * 3` first and then add `4`, yielding 10? Or does it do `3 + 4` first and then multiply by `2`, yielding 14?

The rules are consistent with regular mathematics: Multiplication and division (meaning `*`, `/`, and `%`) happen before `+` and `-`. So the example above evaluates to 10.

Assignment happens way late, after almost everything else, because if you write

```
x = y + 4 ;
```

you never ever mean that you want to store the value of `y` in `x` and then add 4 to that result—you always mean that you want to add 4 to the value in `y` and then store that result into `x`.

As in mathematics, expressions in parentheses (`(` and `)`) get evaluated first. So the value of

```
2 * ( 3 + 4 )
```

is 14.

`++` and `--` have higher precedence than most other things, including everything we've seen so far. That's so that the `++` in `y * x++` applies just to the `x` and not to the `y * x`.

2 The Preprocessor

Before the compiler starts in on its work in earnest, it transforms your program a little. On some systems this transformation is done by a separate program, called a *preprocessor*, but in Turbo C++, the preprocessor is built into the compiler. There are three important transformations and a few unimportant ones that we won't discuss.

2.1 Macros

If your source code contains a line like

```
#define PI 3.141592654
```

the preprocessor defines a *macro*. What this means is that from now on, every time it sees the symbol `PI`, it will replace it with the sequence `3.141592654`. The compiler proper will never find out about `PI` at all; as far as it's concerned, you wrote out `3.141592654` in full every time.

You can use this for three things:

- You can use it to clarify the code, by writing things like `PI`, rather than `3.141592654` all over the place.
- You can use it to set up *manifest constants* that might change over time. For example, suppose you are writing accounting software for an insurance company; the medical insurance deductible is $200, and you want to compute the payment. You could write

  ```
  payment = claims - 200 ;
  ```

  but then if the deductible ever changed, you'd have to go and find all the `200`'s in your code and change every one. Worse, it might be that not every `200` is actually a deductible—you'd have to decide which ones to change. It's much better to do

  ```
  #define DEDUCTIBLE 200
  ```

  and then you can write

  ```
  payment = claims - DEDUCTIBLE ;
  ```

  and if you need to change the deductible, you just change it in the `#define` directive and nowhere else.
- You can use the macro facility to make your code into an unreadable horror. We will not see an example of how to do this.

Conventionally the names we give to macros contain only capital letters.

2.2 Include Files

If you write

```
#include <file.h>
```

the compiler immediately pauses what it was doing, seeks out a *header file* called `file.h` in some standard place,[5] and pretends that the entire contents of that file appeared in your source file in place of the `#include` directive. If the compiler doesn't find `file.h` in any of the standard places, it complaints.

One of the good things about C is that you can have a program whose source lives in more than one file; then if you make changes to one file you don't have to recompile all the others to make an executable. But if there's some information they need to share, you can put it in one header file and have all of the source files `#include` it; again, if the information changes, you only need to change the one copy in the header file instead of going around and changing each source file.

Similarly, when you want to use a library function like `printf`, there might be information you need to give to the compiler about that function. The people who wrote `printf` can put whatever information is necessary into a header file, and then you can include the header file in your program before you use `printf`. In fact, in order to use `printf`, you have to `#include` the header file `stdio.h`, or else you get undefined behavior.[6]

If you write `#include "file.h"` instead of `#include <file.h>`, the preprocessor looks for `file.h` in the current directory before it looks in the standard places.

Header files don't have to have a `.h` extension, but they always do.

2.3 Comments

The preprocessor provides a facility for including explanatory text, called *comments*, into your program, without confusing the compiler. The rule is this: The sequences `/*` and `*/` delimit comments. The preprocessor replaces the `/*`,

[5] Just what place this is is *implementation defined*, which means that it's well-defined, and that it must be documented, but it differs from system to system.

[6] Our `square` program from Thursday had undefined behavior for this reason. Fortunately, it did the right thing anyway.

the `*/`, and everything in between with blank space, which the compiler ignores.

It's too early for an enormous rant about the importance and proper style of comments, but we'll have several later.

3 Conditions

If a program did the same thing every time we ran it, it wouldn't be useful. We have to have a way to perform certain actions only when certain conditions are true.

3.1 The `if` Statement

The `if` statement has this form:

`if (`*condition*`)` *statement*

The condition is just an expression. We say that the condition is *false* if the value of the expression is zero, and we say that the condition is *true* otherwise. To execute an `if` statement, the computer first evaluates the expression to decide if the condition is true or not. If the condition is true, the computer executes the statement. Otherwise, it doesn't.

The statement could be a compound statement, or it could even be another `if` statement.

3.2 Relational Operators

C provides operators for comparing numbers. The operator `==` tests two expressions for equality, for example. The expression `a == b` has the value 0 if `a` and `b` are not equal and `1` if they are equal. So we can write

```
if ( a == b )
 printf("a and b are equal\n");
```

If `a` and `b` are equal, the expression `a == b` evaluates to 1, so the condition is true, and the `printf` statement gets executed. But otherwise, `a == b` evaluates to 0, so the condition is false and the `printf` is not executed.

a `!=` *b* is true when the value of expression *a* is not equal to the value of expression *b*. Similarly, we have `<`, `>`, `<=`, and `>=`, for testing whether one

expression's value is less than another's, greater than another's, less than or equal to another's, or greater than or equal to another's.

Relational operators have low precedence, just before assignments, but after arithmetic.

3.3 Boolean Operators

Suppose we want the user to enter an integer between 1 and 10, inclusive. We want to write some code to print a rude message if the user didn't do what we wanted. Suppose the user's number is stored in the variable `x` Then we could write

```
if ( x < 1 )
  <print rude message> ;
if ( x > 10 )
  <print rude message> ;
```

but then the code to print the rude message is the same both times. That's bad, because someday someone is going to change one and not the other, and then the program will have different behavior where it used to have the same behavior; or else someday both statements might break[7] and someone might fix only one of them by mistake. A principal rule of programming is to never ever have two pieces of code to do the same thing. Fortunately there's a better way to accomplish what we want:

```
if ( x<1 || x>10 )
  <print rude message> ;
```

`||` reads as 'or', so we say "if `x` is less than 1 **or** `x` is greater than 10...". `||` requires two operands, and an expression with an `||` operator is true if either of its operands are true, false if neither is true.

`||` has a special property: it *short-circuits*. In the example above, suppose the value of `x` is 0. The computer compares `x` with 1, and finds that `x < 1` is true, and so we already know that the rude message will be printed. There's no longer any reason to computer whether or not `x > 10` is true; either way we'll print the message. And in fact in C when the computer is evaluating an `||` expression, if the left-hand operand is true, then the computer never evaluates the right-hand one at all.

[7] Maybe because the print function changed or something.

Similarly, there's a `&&` operator, which is pronounced 'and'. For example:

```
if ( x>=1 && x<=10 )
  <print polite message> ;
```

"If `x` is greater than or equal to 1 **and** `x` is less than or equal to 10, then print the polite message." An expression with `&&` is true if both its operands are true, false otherwise. `&&` also short-circuits, so if `x` is 0 in the example above, then the computer will only bother to evaluate the `x >= 1` part above; since that part is false, it already knows that the whole `&&` expression is false, and there's no point in evaluating the `x <= 10` part.

`||` and `&&` are called *logical operators* because they operate on logical conditions such as `x < 10`, rather than on raw numbers like `12`. There is one other logical operator: `!`, pronounced *not*. `!` is a unary operator; it takes only one operand. When the computer wants to evaluate something like `!`x, it first evaluates x, and then if x is true, `!`x is false, and vice-versa.

`!` has higher precedence than anything else we've seen yet. `&&` and `||` have lower precedence than anything except assignment. `&&` has higher precedence than `||`, which is consistent with conventional mathematics.[8][9]

[8] There is a table of operator precedence in your text, on pages 690–691.
[9] Incidentally, `&&` and `||` are both sequence points.