SYS2/CCCP: Operators and IO

For Java and Python programmers

Dr David Griffin

Operators

Basic arithmetic operators work exactly like you'd expect, so these are not going to be covered here in any depth

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulo - remainder on division)

Relational Operators

All of these work exactly as you'd expect

- `==` (equality)
- `!=` (inequality)
- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal)
- `>=` (greater than or equal)

Increment and Decrement

C provides two increment and two decrement operators

`++x, --x`	`x++,x--`

These add or subtract one from the variable

Prefix operators `++x, --x` apply their operation and then return the value	Postfix operators `x++, x--` return the value and then apply their operation

So for example, if x = 1

`(++x == 2)` is true, and after this statement `x` is 2	`(x++ == 2)` is false, but after this statement `x` is 2

Logical Operators

`&&, ||, !`

- `&&` = logical and
- `||` = logical or
- `!` = logical not

These work on C truthy values and output 1 or 0

- 0 = False
- Not 0 = True

Meaning that all of these are true

- `(4 && 3) == 1`
- `(4 && 0) == 0`
- `(3 || 0) == 1`
- `!0 == 1`
- `!23 == 0`
- `!!23 == 1`

Caveats on Logical Operators

In `(this_returns_false() && this_might_return_true())`, we could short-circuit the and operator and not execute `this_might_return_true()` at all

However, C does not specify if this should happen, or the order in which the operands are evaluated

So be careful if you manipulate state inside a logical comparison like this

As seen before Increment and Decrement can manipulate variables

So when `x = 7`, is `((++x == 8) && (x == 7))` true or false?

?

Caveats on Logical Operators

In `(this_returns_false() && this_might_return_true())`, we could short-circuit the and operator and not execute `this_might_return_true()` at all

However C does not specify if this should happen, or the order in which the operands are evaluated

So be careful if you manipulate state inside a logical comparison like this

As seen before Increment and Decrement can manipulate variables

So when `x = 7`, is `((++x == 8) && (x == 7))` true or false?

C doesn't specify which order to evaluate the two parts in, so there's no way to know if it is true or false. You also can't guarantee `x == 8` after this statement.

Bitwise Operators

These manipulate the individual bits of variables

`char x = 6` has the bits `00000110`

`char y = -1` has the bits `11111111`

Bitwise operators are

`&` bitwise and	`|` bitwise or	`^` bitwise xor	`~` bitwise not	`<<` left shift	`>>` right shift
• `6 & 4 = 4` • `6 & 1 = 0`	• `6 | 3 = 7` • `6 | 1 = 7`	• `6 ^ 3 = 5` • `6 ^ 1 = 7`	• `~1 = -2` • `~-3 = 2`	• `6 << 1 = 12` • `1 << 1 = 2`	• `6 >> 2 = 1` • `12 >> 1 = 6`

Assignment Operators

= works how you'd expect

In-place arithmetic/bitwise operators work how you'd expect

`+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=`

i.e. expand "`x += y;`" to "`x = x + y;`"

In C, assignment operators return the value.

"`x = y = 5;`" sets both `x` and `y` to be `5`

If `y = 1`, "`x = y += 5`" sets both `x` and `y` to be `6`

Easy bug: using assignment instead of equality

When is `(x = 5)` true?

When is `(x == 5)` true?

Assignment Operators

= works how you'd expect

In-place arithmetic/bitwise operators work how you'd expect

`+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=`

i.e. expand "`x += y;`" to "`x = x + y;`"

In C, assignment operators return the value.

"`x = y = 5;`" sets both `x` and `y` to be `5`

If `y = 1`, "`x = y += 5`" sets both `x` and `y` to be `6`

Easy bug: using assignment instead of equality

`(x = 5)` is valid C and is always true

`(x == 5)` is only true when `x` is `5`

Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(C99)	
2	++ --	Prefix increment and decrement[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 2]	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-Left
14[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

- Is a bit of a mess, really, but generally like Java
- Either it'll be obvious to you
- Or you should just use brackets to make sure you get the right meaning

Input and Output

We've been using `printf` to print stuff to screen

`printf` is one of C's basic IO functions

And the way `printf` works is very useful in understanding how other functions work

Format Strings

`printf` takes at least one argument

The first argument to `printf` is a format string

This tells `printf` what type of data is represented in the remaining arguments, so it can print them

These days most compilers are sufficiently aware to know if you give them an incorrect format string

```
printf("Coord (%.1f, %.1f)", x, y); /*
prints (x, y) coord, 1 decimal place*/
```

```
%s = space terminated char string
%50s = 50-character char string
%d = integer
%u = unsigned integer
%c = char
%x = unsigned integer as hex
%f = floating point
%.2f = floating point with 2 decimal
points of precision
```

Note: Format specifiers occupy a whole page of textbook - they can do a lot more than this, so look them up if needed!

Reading input from user

- `scanf` function (inside `<stdio.h>`)
  - `printf` takes a format string and some variables, and prints to screen
  - `scanf` takes a format string and some variables, and reads user input into those variables
- You must be careful when using `scanf` (and similar functions). If you try to read a string or array into a variable which doesn't have enough space to hold it, C will let you do this.
  - And then, probably, crash.
  - Or worse.

scanf examples

```
// read an int

int x;

scanf(“%d”, &x); /* & means “give this variable to scanf” here
*/


// read a string

char s[50];

scanf(“%50s”, s);; /* don’t need to “give” an array due to the
way C works. Format specifier gives length of array to avoid
buffer overflow. Input strings are space terminated /*


/* read a string then a space, then an int */

scanf(“%50s %d”, s, %d);
```

Other options for user input/output

`int getchar()`: Read one character from input – if more than one character entered before user presses enter, only returns one character

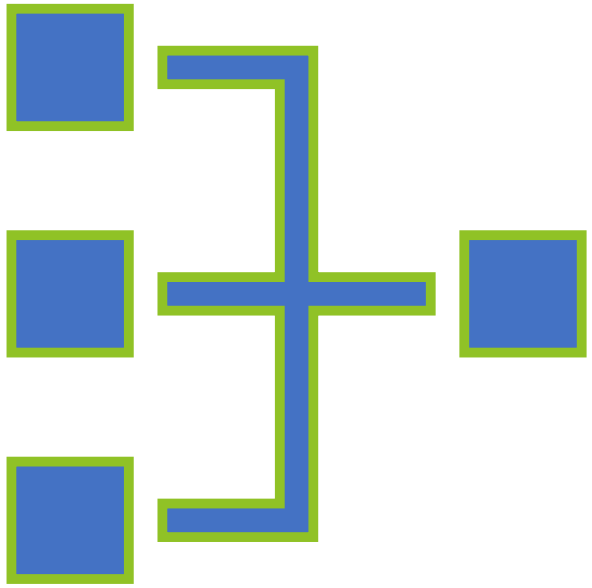
1

`int putchar(int c)`: Write one character to screen

`void puts(char[] string)`: Like printf, but doesn't support format strings.

`void gets(char[] string)`: **DO NOT USE**. Reads a string from input, but has no bounds checking, meaning a user can overflow the target string just by holding a button to make a really long input. Removed from modern C as a security risk.

Handling Files



- `stdio.h` defines the `FILE*` type and file manipulation functions
- `FILE* fopen(const char[] path, const char[] mode);` takes a path and a mode parameter as character strings, returns a `FILE*` that represents the open file
  - Mode parameters: r, rb, w, wb, a, ab – read/write/append (binary)
- `fflush(FILE* fp);` forces all pending writes to actually go to disk
- `fclose(FILE* fp);` flushes and then closes the file so other processes can open it safely

Text Files

- `fprintf` and `fscanf`
- Work exactly like `printf` and `scanf`
- Only the first argument is a `FILE*` from `fopen`
- `fprintf(file, "%d", x);` writes integer `x` to a file
- `fscanf(file, "%50s", s);` reads up to a 50 character string from a file
- Also `fputc` and `fputs` which are like `putc` and `puts` but have a *second* argument for the `FILE*`
- And finally `fgets(char[] out, int n, FILE* file);` which reads a string of at most length `n`, and so is safe – unlike `gets`

Binary Files

Binary files take much less space than text files

And can store structs natively

But unless you know how to read them, you're probably not going to be able to read them at all

`fread` and `fwrite` are C's binary file functions

fwrite

- `int fwrite(void[] p, const int size, const int count, FILE* file);`
- `fwrite` has an argument which is void, because fwrite does not care what the type is
- Instead it copies the memory byte-for-byte from into the file
- So we need to tell it how much memory to copy (using `sizeof`) and how many things to copy (1 or size of array)
- We also need to "give" it the variable with the same rules as `scanf`
  - Use `&v` to give a non-array variable, or just `arr` for arrays
- Returns the number of bytes actually written

fwrite Example

```
#include <stdio.h>
int main(){
  int a[50];
  struct {int x; int y;} glb;
  // put some data in a and glb
  FILE* fp;
  fp = fopen("t.bin", "wb");
  fwrite(&glb, sizeof(glb), 1, fp);
  fwrite(a, sizeof(a[0]), 50, fp);
  fclose(fp);
}
```

fread

- `int fread(void[] p, const int size, const int count, FILE* file);`
- Very similar to fwrite!
  - `fread` has an argument which is void, because `fread` does not care what the type is
  - Instead it copies the data byte-for-byte from the file to memory
  - So we need to tell it how much memory to read (using `sizeof`) and how many things to read (1 or size of array)
  - We also need to "give" it the variable with the same rules as `scanf`
    - Use `&v` to give a non-array variable, or just `arr` for arrays
- Returns the number of bytes actually read

fread Example

```
#include <stdio.h>
int main() {
  int a[50];
  struct {int x; int y;} glb;
  FILE* fp;
  fp = fopen("t.bin", "rb");
  fread(&glb, sizeof(glb), 1, fp);
  fread(a, sizeof(a[0]), 50, fp);
  fclose(fp);
}
```

Conclusions

Everyone should have learned about

C Operators	Input Output and Format Strings	File handling	Text and Binary File IO

However, there was quite a lot of ‘magic’ bits in this lecture, like “giving” variables and this FILE* type. Next time, we’ll go into what this means

Assignment

This lectures assignment has a bit on operators

A bit on format strings

And a bit on file reading/writing

There's nothing too difficult in there, but these are all fairly important things to know how to do in C