TCP

- **Fast path** is recovered for sending TCP.

```c
/* Note, it is the only place, where
*/
if (flag |= FLAG_WIN_UPDATE;
```

```c
/* and in FreeBSD.
*/
Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
*/
/* Update our send window.
*/
```

```c
return (after(
```

```c
/* Check that window update is acceptable.
*/
return (!(flag & FLAG_ECE) ||
```

```c
static __inline__
}
```

```c
{ return (!(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
```

```c
static __inline__
}
```

```c
return (min(
```

```c
if (!after(TCP_SKB_CB(
```

```c
if (after(TCP_SKB_CB(
```

```c
if (after(TCP_SKB_CB(
```

```c
if (after(TCP_SKB_CB(
```

```c
else {
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```

```c
*/
```
A note on 113 assignments

- Please write clear, correct code
  - meaningful variable and function names
  - helpful comments

- Goal of assignments is to practice writing C programs
  - Unlike other CS courses, where more emphasis is on theory
  - Feel free to explore and use C language features, even ones we haven’t covered in class
  - You can implement extra things not required by assignment

printf

- Syntax: `printf(format_string, val1, val2, ...);
- `format_string` can include placeholders that specify how the arguments `val1, val2, etc. should be formatted
- `%c`: format as a character
- `%d`: format as an integer
- `%f`: format as a floating-point number
- `%`: print a ` character

int i = 90;
float f = 3.0;
printf("%d roads\n", 42);
printf("i = %d%%, f = %f\n", i, f);

Reading input from keyboard

- `scanf` is the opposite of `printf`
  - Syntax: `scanf(format_string, val1, val2, ...);
  - Tries to parse input according to `format_string`
  - Like `printf`, `format_string` includes placeholders that specifies how values should be parsed

```c
int I;
printf("enter an integer: ");
scanf("%d", &I);
```

- Note the `&` before the variable name. This is required!
- Passes a pointer to the variable `I`, instead of the value of `I`
- We’ll talk much more about this later.

More `scanf` examples

- Read a float from the keyboard

```c
float f;
printf("enter a float: ");
scanf("%f", &f);
```

- Parse a date into month, day, year

```c
int month, day, year;
printf("enter a date: ");
scanf("%d/%d/%d", &month, &day, &year);
```

Variables

- Variables have a name and a type
- Restrictions on variable names
  - Must begin with a letter
  - Can contain letters, digits, and underscores (_)
  - Can’t be a reserved word (if, else, void, etc.)
  - Only the first 31 characters matter

- C has 4 basic built-in types
  - char, int, float, double
More on types

- C also defines **type qualifiers** that modify basic types
  - Short, long, unsigned, signed
  - Warning: meaning differs between compilers and machines!

<table>
<thead>
<tr>
<th>Type</th>
<th>Typical size</th>
<th>Typical range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>[-128, 127]</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte</td>
<td>[-32768, 32767]</td>
</tr>
<tr>
<td>short int</td>
<td>2 bytes</td>
<td>[-32768, 32767]</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>[-2,147,483,648, 2,147,483,647]</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4 bytes</td>
<td>[0, 4,294,967,295]</td>
</tr>
<tr>
<td>long int</td>
<td>8 bytes</td>
<td>[-2,147,483,648, 2,147,483,647]</td>
</tr>
<tr>
<td>unsigned int</td>
<td>8 bytes</td>
<td>[0, 9,223,372,036,854,775,807]</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>Approx. ±[4.94e-324 to 1.80e+308]</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>Approx. ±[1.40e-45, 3.40e+38]</td>
</tr>
<tr>
<td>long long int</td>
<td>8 bytes</td>
<td>[0, 9,223,372,036,854,775,807]</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>[-32768, 32767]</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
<td>[0, 256]</td>
</tr>
</tbody>
</table>

Variable declaration and initialization

- C requires all variables to be declared **before** any other statements
  - Although this was relaxed in C99 standard

```c
int main() {
    int x = 1, y;
    int sum;
    return 0;
}
```

- The initial value of a variable is **undefined**

```c
int i;
printf("%d", i); /* undefined behavior */
```

Other variable qualifiers

- **extern**: used to share variables across C source files
- **static**: used to prevent variables from being accessed in other source files
  - We'll see other uses of static later
- Qualifiers that are used infrequently:
  - **register**: requests that the compiler store the variable in a processor register instead of in memory
  - **volatile**: tells the compiler that the variable's value might be changed by some external force (another thread, etc.)

Example: character constants

```c
char one = '1', two = '2';
printf("one = %c, two = %c\n", one, two);
```

Characters

- Character constants are surrounded by single quotes
  - E.g. ‘a’, ‘0’, ‘\n’
- Escape sequences used to write special constants, e.g.:
  - '\n': newline
  - '\t': tab
  - '\\': backslash
- Character constants are converted to integers using ASCII value
  - ‘a’ => 97, ‘b’ => 98, ..., ‘z’ => 122
  - ‘A’ => 65, ‘B’ => 66, ..., ‘Z’ => 90
  - ‘0’ => 48, ‘1’ => 49, ..., ‘9’ => 57
  - ‘\n’ => 10, ‘\t’ => 9

Example: numeric constants

- Examples of numeric constants
  - 1234: integer constant
  - 1234L: long integer constant
  - 1234u: unsigned integer constant
  - 1.415: double constant
  - 3.1415f: float constant
  - 0x1f: integer constant, expressed in hexadecimal
  - 0134: integer constant, expressed in octal
Another example

- Print an ASCII table in decimal and hexadecimal

```c
#include <stdio.h>

int main(void) {
    char j;
    for(j=’a’; j<=’m’; j++)
        printf("%c %3d %3x
", j, j, j);
    return 0;
}
```

Type conversions

- C is very flexible with type conversions
  - C is weakly typed compared to other languages like Java
  - If an operator has operands of different types, they are all implicitly converted to the wider type
  - Conversions also occur when assigning a value of one type to a variable of another type
    - Careful: Information may be lost by this conversion!
    - Example: if f is a float and i is an int, i=f will truncate the fractional part of f

Explicit casts

- Casting lets you change the type of a value explicitly
  - Syntax: (newtype) value
  - Example:
    ```c
    float PI = 3.1415;
    float int_part = (int) PI;
    float frac_part = PI - int_part;
    ```

Type conversion example

- Type conversions can cause subtle bugs
  - Q: What is the value of mean after this statement?
    - A: 3, because integer division is used.
    - To compute correctly, use a float constant:
      ```c
      float mean = (2 + 3 + 5) / 3;
      ```
      ```c
      float mean = (2 + 3 + 5) / 3.0;
      ```
      ```c
      float mean = (2 + 3 + 5) / ((float)3);
      ```

Operators

- Assignment: =
- Relational: >, >=, <, <=, ==, !=
- Logical: &&, ||, !
- Binary arithmetic: +, -, *, /, %
  - % is the modulus operator:
    - a%b is the remainder when a is divided by b
    - e.g. 8 % 3 == 2
- Shortcut assignment operators
  - +=, -=, *=, /=, %=, etc. e.g.
    ```c
    x += 2  // same as x = x + 2
    x *= 2  // same as x = x * 2
    x %= 5+3 // same as x = x % (5+3)
    ```

Increment/decrement operators

- There are two types of increment/decrement operators
  - ++x, --x: pre-increment, pre-decrement
    - add or subtract 1 from x, and return the new value
  - x++, x--: post-increment, post-decrement
    - add or subtract 1 from x, and return the original value
- Example
  ```c
  int a = 10, b, c, d;
  b = ++a;
  // a and b are now both 11
  c = a++;
  // a is now 12, c is 11
  ```
Increment/decrement operators

• These operators are often used in loops
  • Q: What is the difference between these code snippets?

```c
int j;
for(j=0; j<10; ++j) {
    // some code
};
```

```c
int j;
for(j=0; j<10; j++) {
    // some code
};
```

• Avoid these operators in complex expressions
  • Q: What does this program print?

```c
int a = 2;
printf("%d %d\n", --a, --a);
```

Three ways to increment...

• Three ways to increment/decrement a variable in C
  • `x = x + 1;`
  • `x += 1;`
  • `x++;`

• Which you use is a matter of style and efficiency
  • `x++` may be slightly more efficient than `x += 1`
  • `x += 1` may be slightly more efficient than `x = x + 1`

Order of evaluation

• Operator precedence and associativity rules define the order in which operators are evaluated
  • Some examples:
    • `5 + 3 / 2` ≡ `5 + (3/2)`
    • `1 - 1 - 1` ≡ `(1 - 1) - 1`
    • `3 < 5 + 2` ≡ `(3 < 5) + 2`

Avoid confusing expressions

• Use parentheses to make precedence clear
  • Q: What does this code do?

```c
void main() {
    int a = -2, b = -1, c = 0;
    if(a < b < c)
    
    if (a >= b >= c)
    
    printf("True\n");
    printf("False \n");
}
```

Math functions

• Warning: `^` is the XOR operator, not exponentiation!
  • e.g. In C, `2 ^ 3` != 8 (instead, `2 ^ 3` = 1)

• Many math functions available in math.h:
  • `pow(a, b)` : computes $a^b$
  • `exp(a)` : computes $e^a$
  • `log(a)` : natural logarithm
  • `cos, sin, tan`
  • `acos, asin, atan`
  • etc.