A Quick Introduction to C Programming

presented by Ivan Beschastnikh
Sept 25, 2008

slides by Lewis Girod
CENS Systems Lab
http://lecs.cs.ucla.edu/~girod/talks/c-tutorial.ppt
or,

What I wish I had known about C during my first summer internship
High Level Question: Why is Software Hard?
High Level Question: Why is Software Hard?

Answer(s):

- **Complexity**: Every conditional (“if”) doubles number of paths through your code, every bit of state doubles possible states.
Answer(s):
• **Complexity**: Every conditional ("if") doubles the number of paths through your code, every bit of state doubles possible states.
Answer(s):
- **Complexity**: Every conditional ("if") doubles the number of paths through your code, every bit of state doubles possible states.
High Level Question: Why is Software Hard?

Answer(s):

- **Complexity**: Every conditional (“if”) doubles the number of paths through your code, every bit of state doubles possible states.

- **Mutability**: Software is easy to change. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug.
High Level Question: Why is Software Hard?

Answer(s):
• **Complexity**: Every conditional (“if”) doubles number of paths through your code, every bit of state doubles possible states.

• **Mutability**: Software is easy to change. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug.

• **Flexibility**: Programming problems can be solved in many different ways. Few hard constraints → plenty of “rope”.
High Level Question: Why is Software Hard?

Answer(s):

- **Complexity**: Every conditional (“if”) doubles number of paths through your code, every bit of state doubles possible states.

- **Mutability**: Software is easy to change.. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug.

- **Flexibility**: Programming problems can be solved in many different ways. Few hard constraints → plenty of “rope”.
Writing and Running Programs

1. Write text of program (source code) using an editor such as emacs, save as file e.g. my_program.c

2. Run the compiler to convert program from source to an “executable” or “binary”:
   $ gcc -Wall -g my_program.c -o my_program

3-N. Compiler gives errors and warnings; edit source file, fix it, and re-compile

N. Run it and see if it works 😊
   $ ./my_program
   Hello World
   $
C Syntax and Hello World

```c
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

- `#include` inserts another file. "`h`" files are called "header" files. They contain stuff needed to interface to libraries and code in other "`.c`" files.
- `printf("Hello World\n");` Print out a message. `\n` means "new line".
- `return 0;` Return '0' from this function.
- Blocks of code ("lexical scopes") are marked by `{ ... }`
- The `main()` function is always where your program starts running.
- This is a comment. The compiler ignores this.
- Can your program have more than one `.c` file?
- What do the `< >` mean?
A Quick Digression About the Compiler

Compilation occurs in two steps: “Preprocessing” and “Compiling”

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines are joined ( \)

The compiler then converts the resulting text into binary code the CPU can run directly.
A Function is a series of instructions to run. You pass Arguments to a function and it returns a Value.

“main()” is a Function. It’s only special because it always gets called first when you run your program.

Return type, or void

#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

Calling a Function: “printf()” is just another function, like main(). It’s defined for you in a “library”, a collection of functions you can call from your program.

Returning a value
What is “Memory”? 

Memory is like a big table of numbered slots where bytes can be stored.

The number of a slot is its **Address**. One byte **Value** can be stored in each slot.

Some “logical” data values span more than one slot, like the character string “Hello\n”

A **Type** names a logical meaning to a span of memory. Some simple types are:

- **char**: a single character (1 slot)
- **int**: signed 4 byte integer
- **float**: 4 byte floating point
- **char [10]**: an array of 10 characters
- **int [10]**: an array of 10 characters
- **float**: 4 byte floating point
- **int64_t**: signed 8 byte integer
- **not always…**
- **Signed?…**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
A **Variable** names a place in memory where you store a **Value** of a certain **Type**.

You first **Define** a variable by giving it a name and specifying the type, and optionally an initial value.

```
char x;
char y = 'e';
```

The compiler puts them somewhere in memory.
A **Variable** names a place in memory where you store a **Value** of a certain **Type**.

You first **Define** a variable by giving it a name and specifying the type, and optionally an initial value.

```plaintext
char x;
char y = 'e';
```

Initial value of `x` is undefined

Type is single character (char)

Initial value

**What names are legal?**

Extern? Static? Const?

The compiler puts them somewhere in memory.

**Symbol Table?**
Multi-byte Variables

Different types consume different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char).

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

An int consumes 4 bytes.

padding
Lexical Scoping

Every Variable is Defined within some scope. A Variable cannot be referenced by name (a.k.a. Symbol) from outside of that scope.

Lexical scopes are defined with curly braces { }.

The scope of Function Arguments is the complete body of the function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called “Global” Vars.

```c
void p(char x)
{
    /* p,x */
    char y;
    /* p,x,y */
    char z;
    /* p,x,y,z */
}
    /* p */
char z;
    /* p,z */

void q(char a)
{
    char b;
    /* p,z,q,a,b */
{
    char c;
    /* p,z,q,a,b,c */
}
char d;
/* p,z,q,a,b,d (not c) */

(Returns nothing)
```
Expressions combine Values using Operators, according to precedence.

\[
\begin{align*}
1 + 2 \times 2 & \rightarrow 1 + 4 \rightarrow 5 \\
(1 + 2) \times 2 & \rightarrow 3 \times 2 \rightarrow 6
\end{align*}
\]

Symbols are evaluated to their Values before being combined.

```c
int x=1;
int y=2;
x + y \times y \rightarrow x + 2 \times 2 \rightarrow x + 4 \rightarrow 1 + 4 \rightarrow 5
```

Comparison operators are used to compare values. In C, 0 means “false”, and any other value means “true”.

```c
int x=4;
\begin{align*}
(x < 5) & \rightarrow (4 < 5) \rightarrow <true> \\
(x < 4) & \rightarrow (4 < 4) \rightarrow 0 \\
((x < 5) || (x < 4)) & \rightarrow (<true> || (x < 4)) \rightarrow <true>
\end{align*}
```

Not evaluated because first clause was true
Comparison and Mathematical Operators

== equal to
< less than
<= less than or equal
> greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
! logical not

+ plus
- minus
* mult
/ divide
% modulo

& bitwise and
| bitwise or
^ bitwise xor
~ bitwise not
<< shift left
>> shift right

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit. For oft-confused cases, the compiler will give you a warning “Suggest parens around …” – do it!

Beware division:
- If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 whereas 5 / 10.0 → 0.5
- Division by 0 will cause a FPE

Don’t confuse & and &&..
1 & 2 → 0 whereas 1 && 2 → <true>
Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = y</td>
<td>assign y to x</td>
</tr>
<tr>
<td>x++</td>
<td>post-increment x</td>
</tr>
<tr>
<td>++x</td>
<td>pre-increment x</td>
</tr>
<tr>
<td>x--</td>
<td>post-decrement x</td>
</tr>
<tr>
<td>--x</td>
<td>pre-decrement x</td>
</tr>
</tbody>
</table>

**Note the difference between ++x and x++:**

```c
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```c
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don’t confuse = and ==! The compiler will warn “suggest parens”.

```c
int x=5;
if (x==6)   /* false */
{
    /* ... */
} /* x is still 5 */
```

```c
int x=5;
if (x=6)   /* always true */
{
    /* x is now 6 */
} /* ... */
```


```c
#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }

    /* “recursive” case */
    return x * pow(x, exp - 1);
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```
A More Complex Program: pow

```c
#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp) {
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x*pow(x, exp - 1);
}

int main(int argc, char **argv) {
    float p;
    p = pow(10.0, 5);
    printf("p = \%f
", p);
    return 0;
}
```

"if" statement

*/ if evaluated expression is not 0 */
if (expression) {
    /* then execute this block */
} else {
    /* otherwise execute this block */
}

Need braces?

X ? Y : Z

Short-circuit eval?
detecting brace errors
A More Complex Program: pow

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

float pow(float x, uint32_t exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x * pow(x, exp - 1);
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```

Tracing "pow()":
- What does pow(5,0) do?
- What about pow(5,1)?
- "Induction"

"if" statement

/* if evaluated expression is not 0 */
if (expression) {
    /* then execute this block */
} else {
    /* otherwise execute this block */
}
A More Complex Program: pow

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

float pow(float x, uint32_t exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x * pow(x, exp - 1);
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %.2f\n", p);
    return 0;
}
```

Challenge: write pow() so it requires log/exp iterations
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

```c
#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp) {
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* “recursive” case */
    return x*pow(x, exp - 1);
}

int main(int argc, char **argv) {
    float p;
    p = pow(5.0, 1);
    printf("p = %f\n", p);
    return 0;
}
```
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.
Recall lexical scoping. If a variable is valid "within the scope of a function", what happens when you call that function recursively? Is there more than one "exp"?

Yes. Each function call allocates a "stack frame" where Variables within that function’s scope will reside.

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

float pow(float x, uint32_t exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x*pow(x, exp - 1);
}

int main(int argc, char **argv)
{
    float p;
    p = pow(5.0, 1);
    printf("p = %f\n", p);
    return 0;
}
```
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>5.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>0</td>
</tr>
<tr>
<td>float x</td>
<td>5.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>1</td>
</tr>
<tr>
<td>int argc</td>
<td>1</td>
</tr>
<tr>
<td>char **argv</td>
<td>0x2342</td>
</tr>
<tr>
<td>float p</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Recall lexical scoping. If a variable is valid "within the scope of a function", what happens when you call that function recursively? Is there more than one "exp"?

Yes. Each function call allocates a "stack frame" where Variables within that function's scope will reside.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>5.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>1</td>
</tr>
<tr>
<td>int argc</td>
<td>1</td>
</tr>
<tr>
<td>char **argv</td>
<td>0x2342</td>
</tr>
<tr>
<td>float p</td>
<td>undefined</td>
</tr>
</tbody>
</table>

#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }

    /* "recursive" case */
    return x*pow(x, exp - 1);
}

int main(int argc, char **argv)
{
    float p;
    p = pow(5.0, 1);
    printf("p = %.2f\n", p);
    return 0;
}
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.

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

float pow(float x, uint32_t exp) {
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* “recursive” case */
    return x * pow(x, exp - 1);
}

int main(int argc, char **argv) {
    float p;
    p = pow(5.0, 1);
    printf("p = %f\n", p);
    return 0;
}
```
Iterative pow(): the “while” loop

Problem: “recursion” eats stack space (in C). Each loop must allocate space for arguments and local variables, because each new call creates a new “scope”.

Solution: “while” loop.

```c
float pow(float x, uint exp) {
    int i=0;
    float result=1.0;
    while (i < exp) {
        result = result * x;
        i += 1;
    }
    return result;
}

int main(int argc, char **argv) {
    float p;
    p = pow(10.0, 5);
    printf("p = %.f\n", p);
    return 0;
}
```
The “for” loop is just shorthand for this “while” loop structure.

float pow(float x, uint exp) {
    float result=1.0;
    int i;
    i=0;
    while (i < exp) {
        result = result * x;
        i++;
    }
    return result;
}

int main(int argc, char **argv) {
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
So far, all of our examples all of the data values we have used have been defined in our lexical scope.
So far, all of our examples all of the data values we have used have been defined in our lexical scope.
Referencing Data from Other Scopes

So far, all of our examples all of the data values we have used have been defined in our lexical scope.

```c
float pow(float x, uint exp) {
    float result = 1.0;
    int i;
    for (i = 0; (i < exp); i++) {
        result = result * x;
    }
    return result;
}

int main(int argc, char **argv) {
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```

Nothing in this scope

Uses any of these variables
Can a function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```
Can a function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```
void pow_assign(float x, uint exp) {
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{ 
    float p=2.0;
    pow_assign(p, 5);
}
void pow_assign(float x, uint exp) {
    float result = 1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{ float p=2.0;
pow_assign(p, 5); }
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{  
    float p=2.0;
    pow_assign(p, 5);
}
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{  float p=2.0;
    pow_assign(p, 5);
}

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>2.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>1.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
Remember the stack!

```c
void pow_assign(float x, uint exp)
{
    float result = 1.0;
    int i;
    for (i = 0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{  
    float p = 2.0;
    pow_assign(p, 5);
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>2.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
NO!

Remember the stack!

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{ float p=2.0;
  pow_assign(p, 5);
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>2.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{ 
    float p=2.0;
    pow_assign(p, 5);
}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>32.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{ float p=2.0;
  pow_assign(p, 5);
}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>32.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{ float p=2.0;
  pow_assign(p, 5);
}
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{  
    float p=2.0;
    pow_assign(p, 5);
}
void pow_assign(float x, uint exp) 
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{
    float p=2.0;
pow_assign(p, 5);
}
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

{  
    float p=2.0;  
    pow_assign(p, 5);
}
Recall our model for variables stored in memory.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
</tbody>
</table>
Recall our model for variables stored in memory

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

What if we had a way to find out the address of a symbol, and a way to reference that memory location by address?

\[
\text{address}_\text{of}(y) == 5
\]

\[
\text{memory}_\text{at}[5] == 101
\]
Recall our model for variables stored in memory

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

What if we had a way to find out the address of a symbol, and a way to reference that memory location by address?

```c
void f(address_of_char p)
{
    memory_at[p] = memory_at[p] - 32;
}
```
Recall our model for variables stored in memory

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
</tbody>
</table>

What if we had a way to find out the address of a symbol, and a way to reference that memory location by address?

```c
void f(address_of_char p)
{
    memory_at[p] = memory_at[p] - 32;
}

char y = 101;      /* y is 101 */
f(address_of(y));  /* i.e. f(5) */
/* y is now 101−32 = 69 */
```
This is exactly how “pointers” work.

```
void f(address_of_char p)
{
    memory_at[p] = memory_at[p] - 32;
}

char y = 101;    /* y is 101 */
f(address_of(y)); /* i.e. f(5) */
/* y is now 101–32 = 69 */
```

```
void f(char * p)
{
    *p = *p - 32;
}

char y = 101;    /* y is 101 */
f(&y);           /* i.e. f(5) */
/* y is now 101–32 = 69 */
```

A “pointer type”: pointer to char
This is exactly how “pointers” work.

"address of" or reference operator: &
“memory_at” or dereference operator: *

void f(address_of_char p)
{
    memory_at[p] = memory_at[p] - 32;
}

char y = 101; /* y is 101 */
f(address_of(y)); /* i.e. f(5) */
/* y is now 101-32 = 69 */

A “pointer type”: pointer to char

void f(char * p)
{
    *p = *p - 32;
}

char y = 101; /* y is 101 */
f(&y); /* i.e. f(5) */
/* y is now 101-32 = 69 */

Pointers are used in C for many other purposes:
• Passing large objects without copying them
• Accessing dynamically allocated memory
• Referring to functions
How should pointers be initialized?
A Valid pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

How should pointers be initialized?
A **Valid** pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:
- Program errors that set the pointer value to a strange number
- Use of a pointer that was at one time valid, but later became invalid
A **Valid** pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:
- Program errors that set the pointer value to a strange number
- Use of a pointer that was at one time valid, but later became invalid

Will ptr be valid or invalid?

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
}
```
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

Answer: Invalid!
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x = 0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

100 char * ptr ?

Grows
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x = 0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

| 101 | char x | 0 |
| 100 | char * ptr | ? |
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

100 | char * ptr |
101 | char x    |

Return 101

Grows
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x = 0;
    return &x;
}

{  
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```
char * get_pointer()
{
    char x=0;
    return &x;
}

{  
    char * ptr = get_pointer();
    *ptr = 12;  /* valid? */
    other_function();
}
```

But now, ptr points to a location that’s no longer in use, and will be reused the next time a function is called!
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```
char * get_pointer()
{
    char x=0;
    return &x;
}
{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!
We’ve seen a few types at this point: char, int, float, char *

Types are important because:
• They allow your program to impose logical structure on memory
• They help the compiler tell when you’re making a mistake

In the next slides we will discuss:
• How to create logical layouts of different types (structs)
• How to use arrays
• How to parse C type names (there is a logic to it!)
• How to create new types using typedef
Structures

struct: a way to compose existing types into a structure

```c
#include <sys/time.h>

/* declare the struct */
struct my_struct {
    int counter;
    float average;
    struct timeval timestamp;
    uint in_use:1;
    uint8_t data[0];
};

/* define an instance of my_struct */
struct my_struct x = {
    in_use: 1,
    timestamp: {
        tv_sec: 200
    }
};

x.counter = 1;

x.average = sum / (float)(x.counter);

struct my_struct * ptr = &x;
ptr->counter = 2;
(*ptr).counter = 3; /* equiv. */
```

- struct timeval is defined in this header
- structs define a layout of typed fields
- structs can contain other structs
- fields can specify specific bit widths
- A newly-defined structure is initialized using this syntax. All unset fields are 0.
- Fields are accessed using ‘.’ notation.
- A pointer to a struct. Fields are accessed using ‘->’ notation, or (*ptr).counter
Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```c
/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* accessing element 0 */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3);  /* x[3] */

/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for ( i=0; i<COUNT; i++ ) {
    /* process y[i] */
    printf("%c\n", y[i]);
}
```

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0..9)

<table>
<thead>
<tr>
<th>x</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char x [0]</td>
<td>100</td>
<td>'t'</td>
</tr>
<tr>
<td>char x [1]</td>
<td>101</td>
<td>'e'</td>
</tr>
<tr>
<td>char x [2]</td>
<td>102</td>
<td>'s'</td>
</tr>
<tr>
<td>char x [3]</td>
<td>103</td>
<td>'t'</td>
</tr>
<tr>
<td>char x [4]</td>
<td>104</td>
<td>'0'</td>
</tr>
</tbody>
</table>

What’s the difference between char x[] and char *x?

For loop that iterates from 0 to COUNT-1. Memorize it!
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

```c
int x;  /* int; */ typedef int T;
int *x; /* pointer to int; */ typedef int *T;
int x[10]; /* array of ints; */ typedef int T[10];
int *x[10]; /* array of pointers to int; */ typedef int *T[10];
int (*x)[10]; /* pointer to array of ints; */ typedef int (*T)[10];
```
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

```
int x;        /* int;                      */  typedef int T;
int *x;       /* pointer to int;           */  typedef int *T;
int x[10];    /* array of ints;            */  typedef int T[10];
int *x[10];   /* array of pointers to int; */  typedef int *T[10];
int (*x)[10]; /* pointer to array of ints; */  typedef int (*T)[10];
```

typedef defines a new type

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

```
int *x[10];
```

x is an array of pointers to int
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

```
int x;        /* int;                      */  typedef int T;
int *x;       /* pointer to int;           */  typedef int *T;
int x[10];    /* array of ints;            */  typedef int T[10];
int *x[10];   /* array of pointers to int; */  typedef int *T[10];
int (*x)[10]; /* pointer to array of ints; */  typedef int (*T)[10];
```

typedef defines a new type
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

```c
int x;        /* int;                      */  typedef int T;
int *x;       /* pointer to int;           */  typedef int *T;
int x[10];    /* array of ints;            */  typedef int T[10];
int *x[10];   /* array of pointers to int; */  typedef int *T[10];
int (*x)[10]; /* pointer to array of ints; */  typedef int (*T)[10];
```

typedef defines a new type

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.
The other confusing form is the function type. For example, qsort: (a sort function in the standard library)

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

The last argument is a comparison function.

```c
/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type)(const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

const means the function is not allowed to modify memory via this pointer.

void * is a pointer to memory of unknown type.

size_t is an unsigned int.
The other confusing form is the function type. For example, qsort: (a sort function in the standard library)

```c
void qsort(void *base, size_t nmemb, size_t size,
          int (*compar)(const void *, const void *));

/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type) (const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

- `void *` is a pointer to memory of unknown type.
- `size_t` is an unsigned int.
- `const` means the function is not allowed to modify memory via this pointer.
- The last argument is a comparison function.

For more details:
$ man qsort
So far all of our examples have allocated variables *statically* by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires *dynamic* allocation.
Dynamic Memory Allocation

So far all of our examples have allocated variables **statically** by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires **dynamic** allocation.

```c
int * alloc_ints(size_t requested_count)
{
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can’t allocate %d ints: %m\n", requested_count);
        return NULL;
    }
    /* now big_array[0] .. big_array[requested_count-1] are * valid and zeroed. */
    return big_array;
}
```

`sizet()` reports the size of a type in bytes

- `calloc()` allocates memory for N elements of size k
- Returns NULL if can’t alloc
- It’s OK to return this pointer. It will remain valid until it is freed with `free()`
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()’d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a “memory leak”.

Reference counting
Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()’d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a “memory leak”.

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()’d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a “memory leak”.

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic
Some Common Errors and Hints

sizeof() can take a variable reference in place of a type name. This guarantees the right allocation, but don’t accidentally allocate the sizeof() the pointer instead of the object!

```c
/* allocating a struct with malloc() */
struct my_struct *s = NULL;
s = (struct my_struct *)malloc(sizeof(*s)); /* NOT sizeof(s)!! */
if (s == NULL) {
    printf(stderr, “no memory!”);
    exit(1);
}
memset(s, 0, sizeof(*s));

/* another way to initialize an alloc’d structure: */
struct my_struct init = {
    counter: 1,
    average: 2.5,
    in_use: 1
};

/* memmove(dst, src, size) (note, arg order like assignment) */
memmove(s, &init, sizeof(init));

/* when you are done with it, free it! */
free(s);
s = NULL;
```

- malloc() allocates n bytes
- malloc() does not zero the memory, so you should memset() it to 0.
- memmove is preferred because it is safe for shifting buffers
- Always check for NULL. Even if you just exit(1).
Some Common Errors and Hints

/* allocating a struct with malloc() */
struct my_struct *s = NULL;
s = (struct my_struct *)malloc(sizeof(*s));  /* NOT sizeof(s)!! */
if (s == NULL) {
    printf(stderr, “no memory!”);
    exit(1);
}
memset(s, 0, sizeof(*s));

/* another way to initialize an alloc’d structure: */
struct my_struct init = {
    counter: 1,
    average: 2.5,
    in_use: 1
};

/* memmove(dst, src, size) (note, arg order like assignment) */
memmove(s, &init, sizeof(init));

/* when you are done with it, free it! */
free(s);
s = NULL;

sizeof() can take a variable reference in place of a type name. This guarantees the right allocation, but don’t accidentally allocate the sizeof() the *pointer* instead of the *object*!

malloc() allocates n bytes

Always check for NULL.. Even if you just exit(1).

malloc() does not zero the memory, so you should memset() it to 0.

memmove is preferred because it is safe for shifting buffers
sizeof() can take a variable reference in place of a type name. This guarantees the right allocation, but don’t accidentally allocate the sizeof() the pointer instead of the object!

```c
/* allocating a struct with malloc() */
struct my_struct *s = NULL;
s = (struct my_struct *)malloc(sizeof(*s)); /* NOT sizeof(s)!! */
if (s == NULL) {
    printf(stderr, “no memory!”);
    exit(1);
}
memset(s, 0, sizeof(*s));

/* another way to initialize an alloc’d structure: */
struct my_struct init = {
    counter: 1,
    average: 2.5,
    in_use: 1
};

/* memmove(dst, src, size) (note, arg order like assignment) */
memmove(s, &init, sizeof(init));

/* when you are done with it, free it! */
free(s);
s = NULL;
```

malloc() allocates n bytes

malloc() does not zero the memory, so you should memset() it to 0.

Always check for NULL.. Even if you just exit(1).

Why?

memmove is preferred because it is safe for shifting buffers

Why?
Some Common Errors and Hints

/* allocating a struct with malloc() */
struct my_struct *s = NULL;
s = (struct my_struct *)malloc(sizeof(*s));  /* NOT sizeof(s)!! */
if (s == NULL) {
    printf(stderr, “no memory!”);
    exit(1);
}
memset(s, 0, sizeof(*s));

/* another way to initialize an alloc’d structure: */
struct my_struct init = {
    counter: 1,
    average: 2.5,
    in_use: 1
};

/* memmove(dst, src, size) (note, arg order like assignment) */
memmove(s, &init, sizeof(init));

/* when you are done with it, free it! */
free(s);
s = NULL;

sizeof() can take a variable reference in place of a type name. This guarantees the right allocation, but don’t accidentally allocate the sizeof() the pointer instead of the object!

malloc() allocates n bytes

Always check for NULL.. Even if you just exit(1).

malloc() does not zero the memory, so you should memset() it to 0.

memmove is preferred because it is safe for shifting buffers

Use pointers as implied in-use flags!
Macros can be a useful way to customize your interface to C and make your code easier to read and less redundant. However, when possible, use a static inline function instead.

Macros and static inline functions must be included in any file that uses them, usually via a header file. Common uses for macros:

```
/* Macros are used to define constants */
#define FUDGE_FACTOR   45.6
#define MSEC_PER_SEC   1000
#define INPUT_FILENAME "my_input_file"

/* Macros are used to do constant arithmetic */
#define TIMER_VAL      (2*MSEC_PER_SEC)

/* Macros are used to capture information from the compiler */
define DBG(args...) \
    do { \
        fprintf(stderr, "%s:%s:%d: ", \
            __FUNCTION__, __FILE__, __LINENO__); \
        fprintf(stderr, args...); \ 
        fprintf(stderr, args...); \ 
    } while (0)

/* ex. DBG("error: %d", errno); */
```

What's the difference between a macro and a static inline function?

Float constants must have a decimal point, else they are type int

Put expressions in parens.

More on C constants?

Put expressions in parens.

More on C constants?

Put expressions in parens.
Sometimes macros can be used to improve code readability… but make sure what’s going on is obvious.

```c
/* often best to define these types of macro right where they are used */
#define CASE(str) if (strncasecmp(arg, str, strlen(str)) == 0)

void parse_command(char *arg)
{
    CASE("help") {
        /* print help */
    }
    CASE("quit") {
        exit(0);
    }
}

/* and un-define them after use */
#undef CASE
```

Macros can be used to generate static inline functions. This is like a C version of a C++ template. See emstar/libmisc/include/queue.h for an example of this technique.
Some schools of thought frown upon goto, but goto has its place. A good philosophy is, always write code in the most expressive and clear way possible. If that involves using goto, then goto is not bad.

An example is jumping to an error case from inside complex logic. The alternative is deeply nested and confusing “if” statements, which are hard to read, maintain, and verify. Often additional logic and state variables must be added, just to avoid goto.

```c
goto try_again;
```

```c
goto fail;
```
state_t *initialize()
{
    /* allocate state struct */
    state_t *s = g_new0(state_t, 1);
    if (s) {
        /* allocate sub-structure */
        s->sub = g_new0(sub_t, 1);
        if (s->sub) {
            /* open file */
            s->sub->fd =
                open("/dev/null", O_RDONLY);
            if (s->sub->fd >= 0) {
                /* success! */
            } else {
                free(s->sub);
                free(s);
                s = NULL;
            }
        } else {
            /* failed! */
            free(s->sub);
            free(s);
            s = NULL;
        }
    } else {
        /* failed! */
        free(s);
        s = NULL;
    }
    return s;
}
High Level Question: Why is Software Hard?
High Level Question: Why is Software Hard?

Answer(s):

- **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  - Solution: reuse code paths, avoid duplicate state variables

- **Mutability**: Software is easy to change.. Great for rapid fixes ☺.. And rapid breakage 😞.. always one character away from a bug
High Level Question: Why is Software Hard?

Answer(s):

- **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  - Solution: reuse code paths, avoid duplicate state variables

- **Mutability**: Software is easy to change. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug
  - Solution: tidy, readable code, easy to understand by inspection.
High Level Question: Why is Software Hard?

Answer(s):

- **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  - Solution: reuse code paths, avoid duplicate state variables

- **Mutability**: Software is easy to change.. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug
  - Solution: tidy, readable code, easy to understand by inspection. Avoid code duplication; physically the same → logically the same
High Level Question: Why is Software Hard?

Answer(s):

• Complexity: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  – Solution: reuse code paths, avoid duplicate state variables

• Mutability: Software is easy to change.. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug
  – Solution: tidy, readable code, easy to understand by inspection.
    Avoid code duplication; physically the same → logically the same

• Flexibility: Programming problems can be solved in many different ways. Few hard constraints → plenty of “rope”.
High Level Question: Why is Software Hard?

Answer(s):

• **Complexity**: Every conditional (“if”) doubles number of paths through your code, every bit of state doubles possible states
  – Solution: reuse code paths, avoid duplicate state variables

• **Mutability**: Software is easy to change.. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug
  – Solution: tidy, readable code, easy to understand by inspection.
    Avoid code duplication; physically the same → logically the same

• **Flexibility**: Programming problems can be solved in many different ways. Few hard constraints → plenty of “rope”.
  – Solution: discipline and idioms; don’t use all the rope
• **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  – Solution: reuse code paths, avoid duplicate state variables
Addressing Complexity

- **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  - Solution: reuse code paths, avoid duplicate state variables

```
On receive_packet:
    if queue full, drop packet
    else push packet, call run_queue

On transmit_complete:
    state=idle, call run_queue

Run_queue:
    if state==idle && !queue empty
        pop packet off queue
        start transmit, state = busy
```
• **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  - Solution: reuse code paths, avoid duplicate state variables

---

**reuse code paths**

- **On receive_packet:**
  - if queue full, drop packet
  - else push packet, call run_queue

- **On transmit_complete:**
  - state=idle, call run_queue

- **Run_queue:**
  - if state==idle && !queue empty
  - pop packet off queue
  - start transmit, state = busy

- On input, change our state as needed, and call Run_queue. In all cases, Run_queue handles taking the next step…
• **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  – Solution: reuse code paths, avoid duplicate state variables
Addressing Complexity

- **Complexity**: Every conditional ("if") doubles number of paths through your code, every bit of state doubles possible states
  - Solution: reuse code paths, avoid duplicate state variables

```c
int transmit_busy;
msg_t *packet_on_deck;

int start_transmit(msg_t *packet)
{
    if (transmit_busy) return -1;
    /* start transmit */
    packet_on_deck = packet;
    transmit_busy = 1;
    /* ... */
    return 0;
}
```

Why return -1?

```c
msg_t *packet_on_deck;
int start_transmit(msg_t *packet)
{
    if (packet_on_deck != NULL) return -1;
    /* start transmit */
    packet_on_deck = packet;
    /* ... */
    return 0;
}
```
Addressing Mutability

- **Mutability**: Software is easy to change. Great for rapid fixes 😊. And rapid breakage 😞. always one character away from a bug
  - Solution: tidy, readable code, easy to understand by inspection.
    Avoid code duplication; physically the same → logically the same
Addressing Mutability

- **Mutability**: Software is easy to change.. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug
  - Solution: tidy, readable code, easy to understand by inspection.
    - Avoid code duplication; physically the same → logically the same

Tidy code.. Indenting, good formatting, comments, meaningful variable and function names. Version control.. Learn how to use CVS
Addressing Mutability

- **Mutability**: Software is easy to change. Great for rapid fixes 😊.. And rapid breakage 😩.. always one character away from a bug
  - Solution: tidy, readable code, easy to understand by inspection.
  - Avoid code duplication; physically the same → logically the same

Tidy code.. Indenting, good formatting, comments, meaningful variable and function names. Version control.. Learn how to use CVS

Avoid duplication of anything that’s logically identical.

```c
struct pkt_hdr {
    int source;
    int dest;
    int length;
};
struct pkt {
    int source;
    int dest;
    int length;
    uint8_t payload[100];
};
```
Addressing Mutability

- **Mutability**: Software is easy to change.. Great for rapid fixes 😊.. And rapid breakage 😞.. always one character away from a bug
  - Solution: tidy, readable code, easy to understand by inspection. Avoid code duplication; physically the same → logically the same

Tidy code.. Indenting, good formatting, comments, meaningful variable and function names. Version control.. Learn how to use CVS

Avoid duplication of anything that’s logically identical.

```c
struct pkt_hdr {
    int source;
    int dest;
    int length;
};
struct pkt {
    struct pkt_hdr hdr;
    uint8_t payload[100];
};
```

Otherwise when one changes, you have to find and fix all the other places
Solutions to the pow() challenge question
Solutions to the pow() challenge question

Recursive

float pow(float x, uint exp)
{
    float result;

    /* base case */
    if (exp == 0)
        return 1.0;

    /* x^(2*a) == x^a * x^a */
    result = pow(x, exp >> 1);
    result = result * result;

    /* x^(2*a+1) == x^(2*a) * x */
    if (exp & 1)
        result = result * x;

    return result;
}
Solutions to the pow() challenge question

Recursive

```c
float pow(float x, uint exp)
{
    float result;

    /* base case */
    if (exp == 0)
        return 1.0;

    /* x^{2*a} == x^a * x^a */
    result = pow(x, exp >> 1);
    result = result * result;

    /* x^{2*a+1} == x^{2*a} * x */
    if (exp & 1)
        result = result * x;

    return result;
}
```

Iterative

```c
float pow(float x, uint exp)
{
    float result = 1.0;

    int bit;
    for (bit = sizeof(exp) * 8 - 1; bit >= 0; bit--)
        result *= result;
    if (exp & (1 << bit))
        result *= x;

    return result;
}
```
Solutions to the pow() challenge question

### Recursive

```c
float pow(float x, uint exp)
{
    float result;

    /* base case */
    if (exp == 0)
        return 1.0;

    /* x^(2*a) == x^a * x^a */
    result = pow(x, exp >> 1);
    result = result * result;

    /* x^(2*a+1) == x^(2*a) * x */
    if (exp & 1)
        result = result * x;

    return result;
}
```

### Iterative

```c
float pow(float x, uint exp)
{
    float result = 1.0;

    int bit;
    for (bit = sizeof(exp)*8-1; bit >= 0; bit--)
    {
        result *= result;
        if (exp & (1 << bit))
            result *= x;
    }

    return result;
}
```

Which is better? Why?