CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

```
void swap_values(int left, int right) {
    int temp = left;
    left = right;
    right = temp;        }

void swap_referents(int * left, int * right) {
    int temp = *left;
    *left = *right;
    *right = temp;       }

void swap_references(int * left, int * right) {
    int * temp = left;
    left = right;
    right = temp;        }
```

Which swap()s work?

```
void main(void) {
    int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6;
    swap_values(a, b);
    swap_referents(&c, &d);
    swap_references(&e, &f);
    printf("(%d %d), (%d %d), (%d %d)\n", a, b, c, d, e, f);
```

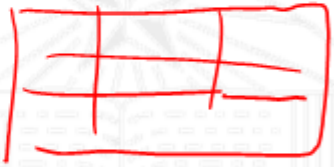
Announcements

- Lecture postings will now be day of (or day after) ✓

- A few Lab 3 changes
  - MatrixSubmatrix() ✓
  - MatrixPrint() ✓
  - Loops required ✓
  - Re-download lab doc

300 3.1415940

-1.8



Roadmap

- Pointers Review ✓
- Arrays Review ✓
- Unit Testing ✓
- Operators (?)
- #defines and literals (?)

Pointers Review

Pointers

How to Create a Pointer Variable

Syntax

```
type *ptrName;
```

- In the context of a declaration, the * merely indicates that the variable is a pointer
- *type* is the type of data the pointer may point to
- Pointer usually described as "*a pointer to type*"

Example

```
int *iPtr;          // Create a pointer to int
int *iPtr, x;       // Create a pointer to int and an int
float *fPtr1, *fPtr2;  // Create 2 float pointers
```

Pointers

Initialization

- To set a pointer to point to another variable, we use the **&** operator (address of), and the pointer variable is used <u>without</u> the dereference operator *****:

```
p = &x;
```

- This assigns the address of the variable **x** to the pointer **p** (**p** now points to **x**)
- Note: **p** must be declared to point to the type of **x** (e.g. `int x; int *p;`)

*p++;

Pointers

Dereferencing

- When accessing the data pointed to by a pointer, we use the pointer <u>with</u> the dereference operator *****:

```
y = *p;
```

- This assigns to the variable **y**, the value of what **p** is pointing to (**x** from the last slide)
- Using ***p**, is the same as using the variable it points to (e.g. **x**)

*p = 4;

Pointers

Dereferencing example

Example

```
int x = 6, *p; // int and a pointer to int

p = &x;        // Assign p the address of x
*p = 5;        // Same as x = 5;
printf("%d",x);
```

- **&x** is a constant memory value
  - It represents the address of **x**
  - The address of **x** will never change
- **p** is a variable pointer to int
  - It can be assigned the address of any int
  - It may be assigned a new address any time

Pointers

Dereferencing example

Example

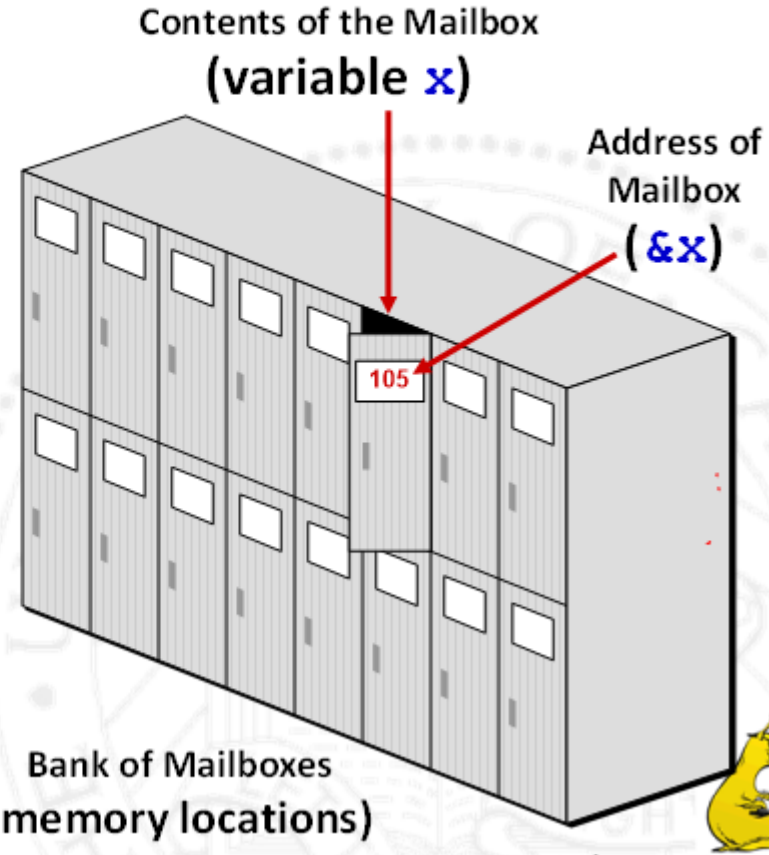
```
int x = 6, *p; // int and a pointer to int

p = &x;        // Assign p the address of x
*p = 5;        // Same as x = 5;
```

- ***p** represents the data pointed to by **p**
  - *p may be used anywhere you would use x
  - * is the dereference operator, also called the indirection operator
  - In the pointer declaration, the only significance of * is to indicate that the variable is a pointer rather than an ordinary variable

Pointers

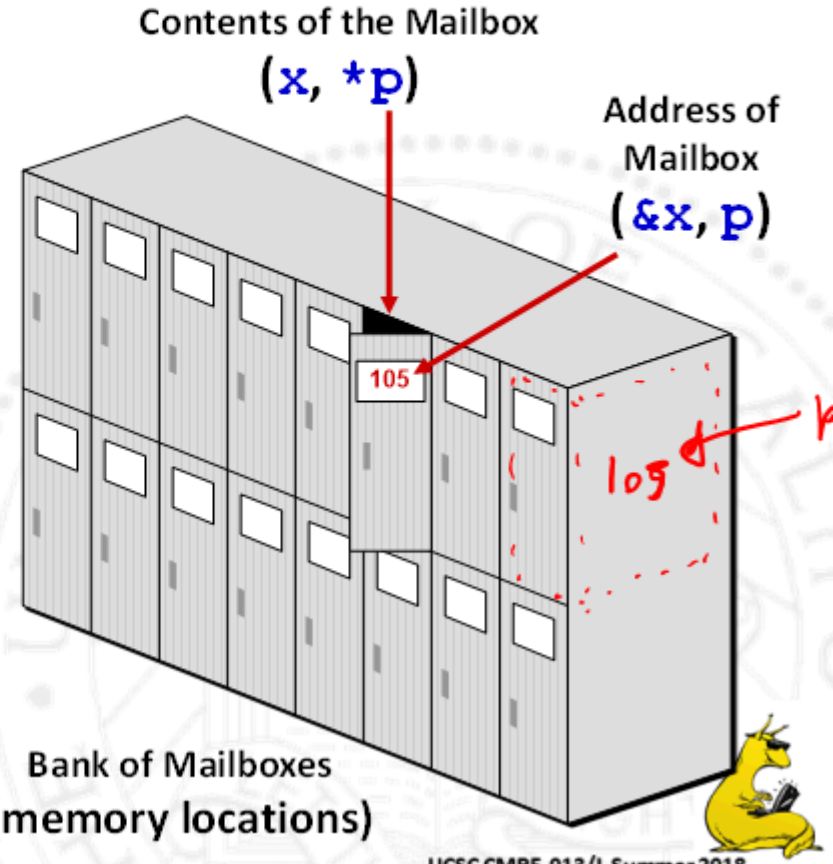
Another view



Pointers

Another view

```
p = &x;
```



Pointers

Another view

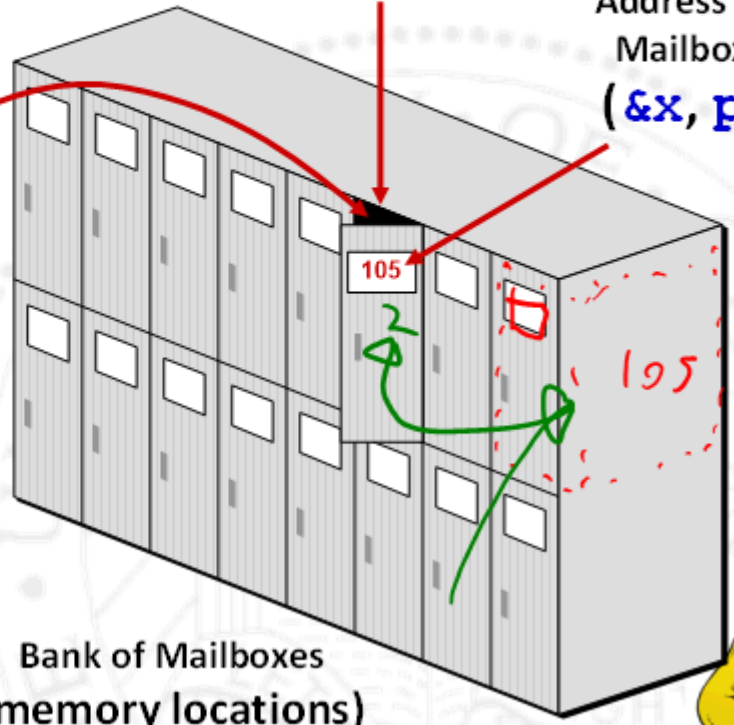
Contents of the Mailbox

(x, *p)

Address of Mailbox

(&x, p)

```
*p = 2;
```



Bank of Mailboxes

(memory locations)

Printing Pointers

```
short int variable = 0xBEEF;
short int * pointer = &variable;

printf("variable                = 0x%x\n", variable);
printf("pointer                 = 0x%x\n", pointer);
printf("dereferenced pointer = 0x%x\n", *pointer);
printf("referenced variable  = 0x%x\n", &variable);

printf("sizeof(variable)        = %d\n", sizeof (variable));
printf("sizeof(pointer)         = %d\n", sizeof (pointer));

printf("referenced pointer     = 0x%x\n", &pointer);
printf("dereferenced variable = %x\n", *variable);
```

Which swap()s work?

```
void swap_values(int left, int right) {
    int temp = left;
    left = right;
    right = temp;        }

void swap_referents(int * left, int * right) {
    int temp = *left;
    (*left) = *right;
    *right = temp;       }

void swap_references(int * left, int * right) {
    int * temp = left;
    left = right;
    right = temp;        }

void main(void) {
    int a = 0xDEAD, b = 0xBEEF, c = 0xCAFE,
            d = 0xBABE, e = 0xBA55, f = 0xBEA7;
    swap_values(a, b);
    swap_referents(&c, &d);
    swap_references(&e, &f);
    printf("(%x %x), (%x %x), (%x %x)\n", a, b, c, d, e, f);
}
```

No swap

swap!

bc

left right

ab

left right

main

swap

No swap

ef

left right

BEEF DEAD

Main - A closer look

```
void main(void) {
    int a = 0xDEAD, b = 0xBEEF, c = 0xCAFE,
            d = 0xBABE, e = 0xBA55, f = 0xBEA7;
    swap_values(a, b);
    swap_referents(&c, &d);
    swap_references(&e, &f);
    printf("(%x %x), (%x %x), (%x %x)\n", a, b, c, d, e, f);
}
```

0000_7F60	00100000	00100000	00000000	00000000
0000_7F70	00000000	00000000	00000000	00000000
0000_7F80	00000000	00153322	38017702	00000000
0000_7F90	38177702	00000000	A000FFA0	9D0021B4
0000_7FA0	00000000	00000000	00000000	00000000
0000_7FB0	00000000	00000000	00000000	00000000
0000_7FC0	0000DEAD	0000BEEF	0000CAFE	0000BABE
0000_7FD0	0000BA55	0000BEA7	00000000	9D002C0C
0000_7FE0	00000000	00000000	00000000	00000000
0000_7FF0	00000000	BFC001C8	00000000	00000000

main

Swap Values - A closer look

```
void swap_values(int left, int right) {
    int temp = left;
    left = right;
    right = temp;        }
```

Swap Referents - A closer look

```
void swap_referents(int * left, int * right) {
    int temp = *left;
    *left = *right;
    *right = temp;        }
```

LEFT

RIGHT

swap referents

min

0000_7F90	0000DEAD	00000000	A0007FA0	A0007FA0
0000_7FA0	A0007FC8	A0007FCC	00000000	00000000
0000_7FB0	00000000	00000000	00000000	00000000
0000_7FC0	0000DEAD	0000BEEF	0000CAFE	0000BABE
0000_7FD0	0000BA55	0000BEA7	00000000	9D002C0C
0000_7FE0	00000000	00000000	00000000	00000000
0000_7FF0	00000000	BFC001C8	00000000	00000000

Swap References - A closer look

```
void swap_references(int * left, int * right) {
    int * temp = left;
    left = right;
    right = temp;        }
```

0000_7F90	0000CAFE	00000000	A0007FA0	A0007FA0
0000_7FA0	A0007FD0	A0007FD4	00000000	00000000
0000_7FB0	00000000	00000000	00000000	00000000
0000_7FC0	0000DEAD	0000BEEF	0000BABE	0000CAFE
0000_7FD0	0000BA55	0000BEA7	00000000	9D002C0C
0000_7FE0	00000000	00000000	00000000	00000000
0000_7FF0	00000000	BFC001C8	00000000	00000000

Array Indexing

What is an array?

- Many values
- All adjacent in memory
- All same type
- Use pointers to access them

Creating Arrays

Arrays may be initialized with a list when declared:

Syntax

```
type arrayName[size] = {item1, …, itemn};
```

- The items must all match the `type` of the array

Example

```
int a[5] = {10, 20, 30, 40, 50};

char b[5] = {'a', 'b', 'c', 'd', 'e'};
```

char b[];

" b[] = { ' . . - - - . } ;

" b[5];

Accessing Arrays

Arrays are accessed like variables, but with an index:

Syntax

```
arrayName[index]
```

- *index* may be a variable or a constant
- The first element in the array has an index of 0
- C does not provide any bounds checking

Example

```
int i, a[10];    // An array that can hold 10 integers

for(i = 0; i < 10; i++) {
    a[i] = 0;    // Initialize all array elements to 0
}
a[4] = 42;       // Set fifth element to 42
```

Exploring 1D Arrays

```
int magicMatrix[8] = {0x70, 0x71, 0x72, 0x73,
                      0x74, 0x75, 0x76, 0x77};

printf("magicMatrix    = %x\n", magicMatrix);
printf("magicMatrix[0] = %x\n", magicMatrix[0]);
printf("magicMatrix[3] = %x\n", magicMatrix[3]);
printf("3[magicMatrix] = %x\n", 3[magicMatrix]);

printf("&magicMatrix[0] = %x\n", &magicMatrix[0]);
printf("&magicMatrix[3] = %x\n", &magicMatrix[3]);


printf("sizeof(magicMatrix)    = %d\n", sizeof(magicMatrix));
printf("sizeof(magicMatrix[0])= %d\n", sizeof(magicMatrix[0]));
printf("N = %d\n", sizeof(magicMatrix)/sizeof(magicMatrix[0]));
```

#define ARRAYSIZE(x) ...

magic matrix[4]

☆ (magic matrix + 4)

☆ (4 + magic matrix)

4[magic matrix]

Creating Multi-dimensional Arrays

Add additional dimensions to an array declaration:

Syntax

```
type arrayName[size1]...[sizen];
```

- Arrays may have any number of dimensions
- Three dimensions tend to be the largest used in common practice

Example

```
int a[10][10];          // 10x10 array for 100 integers

float b[10][10][10];    // 10x10x10 array for 1000 floats
```

Creating Multi-dimensional Arrays

Arrays may be initialized with lists within a list:

Syntax

```
type arrayName[size0]…[sizen] =
                    {{item,…,item},
                          ⋮
                     {item,…,item}};
```

Example

```
char a[3][3] = {{'X', 'O', 'X'},
                {'O', 'O', 'X'},
                {'X', 'X', 'O'}};

int b[2][2][2] = {{{0, 1},{2, 3}},{{4, 5},{6, 7}}};
```

Arrays

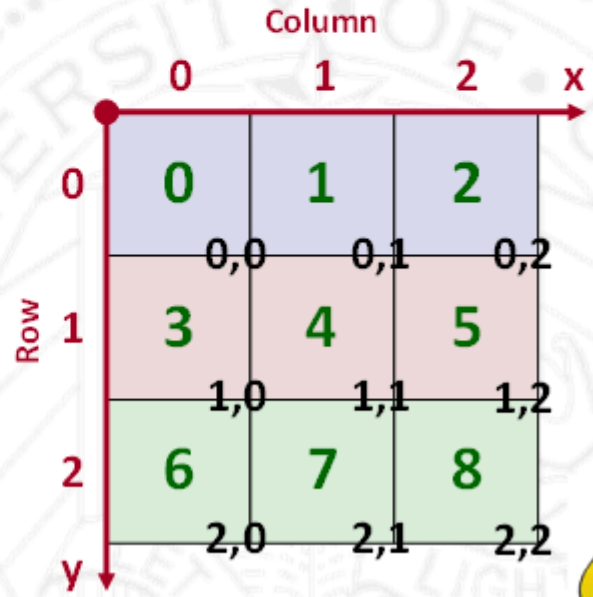
Visualizing 2-Dimensional Arrays

```
int a[3][3] = {{0, 1, 2},
               {3, 4, 5},
               {6, 7, 8}};
```

Row, Column

a[y][x]

Row 0	`a[0][0] = 0;`
	`a[0][1] = 1;`
	`a[0][2] = 2;`
Row 1	`a[1][0] = 3;`
	`a[1][1] = 4;`
	`a[1][2] = 5;`
Row 2	`a[2][0] = 6;`
	`a[2][1] = 7;`
	`a[2][2] = 8;`



magic matrix [2][1]

* (magic matrix + 2 * row len + 1)

Exploring 2-D Arrays

```
int magicMatrix[3][4] = {
    {0x100, 0x101, 0x102, 0x103},
    {0x104, 0x105, 0x106, 0x107},
    {0x108, 0x109, 0x110, 0x111}};

printf("magicMatrix          = %x\n", magicMatrix);
printf("magicMatrix[0]       = %x\n", magicMatrix[0]);
printf("magicMatrix[0][0]    = %x\n", magicMatrix[0][0]);
printf("&magicMatrix[0][0] = %x\n", &magicMatrix[0][0]);

printf("magicMatrix[0][6]          = %x\n", magicMatrix[0][6]);

printf("sizeof(magicMatrix)        = %x\n", sizeof(magicMatrix));
printf("sizeof(magicMatrix[0][0])= %x\n", sizeof(magicMatrix[0][0]));
```

Arrays

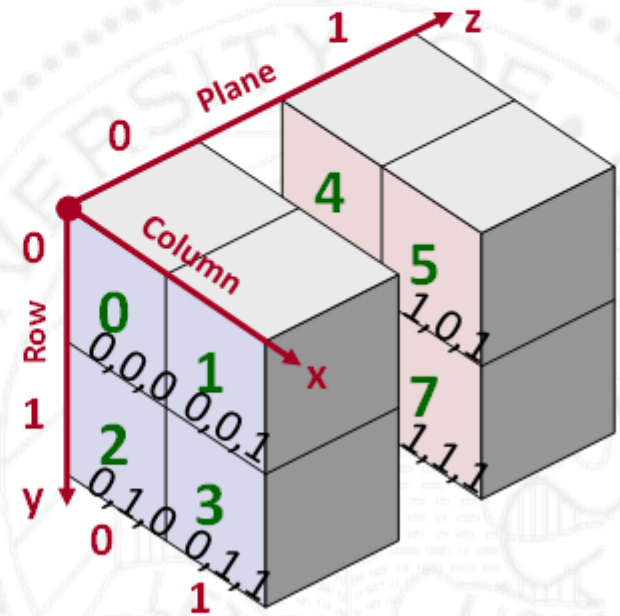
Visualizing 3-Dimensional Arrays

```
int a[2][2][2] = {{{0, 1},{2, 3}},
                  {{4, 5},{6, 7}}};
```

Plane, Row, Column

a[z][y][x]

Plane 0	a[0][0][0] = 0;
	a[0][0][1] = 1;
	a[0][1][0] = 2;
	a[0][1][1] = 3;
Plane 1	a[1][0][0] = 4;
	a[1][0][1] = 5;
	a[1][1][0] = 6;
	a[1][1][1] = 7;



Strings vs Arrays


```
void main(void)
{
    char hello_array[10] = "Hello\n";

    char world_array[10] = {'W','o','r','l','d',10,3,

    printf(hello_array);
    printf(world_array);
}
```

NULL
0
0x9

Incrementing String Pointers

```
void main(void)
{
    char hwString[15] = "Hello World!\n";
    printf(hwString);

    char * hwString_plus_4 = hwString + 4;
    printf(hwString_plus_4);

}
```

what does it print?

2 Ways to Declare Strings:

```
char * fixed_string = "I'll always be here";
char fixed_ptr[] = "You'll always look here for me";

void main(void){
    //we cannot change fixed_string's CONTENTS
    fixed_string[10] = fixed_string[10] - 0x30;
    //but we CAN change it's POINTER:
    fixed_string += 7;

    printf("fixed_string = %s\n", fixed_string);

    //we cannot change where fixed_ptr POINTS
    fixed_ptr = fixed_string;
    //but we CAN change what is there:
    fixed_ptr[11] = fixed_ptr[11] - 32;

    printf("fixed_ptr    = %s\n", fixed_ptr);
}
```

Collaboration: Make A checkerboard

+1	-1	+1
-1	+1	-1
+1	-1	+1

Unit testing

Unit testing

- Testing portions of code in isolation
- Normally testing is per function
- Requires input and expected output to be known a priori

Unit testing

Rationale

- Find problems early
  - Before integration
- Simplify testing by only testing small, segmented portions of code
- Test functionality that may not be exposed otherwise
- Find documentation errors

Unit testing

Preparing

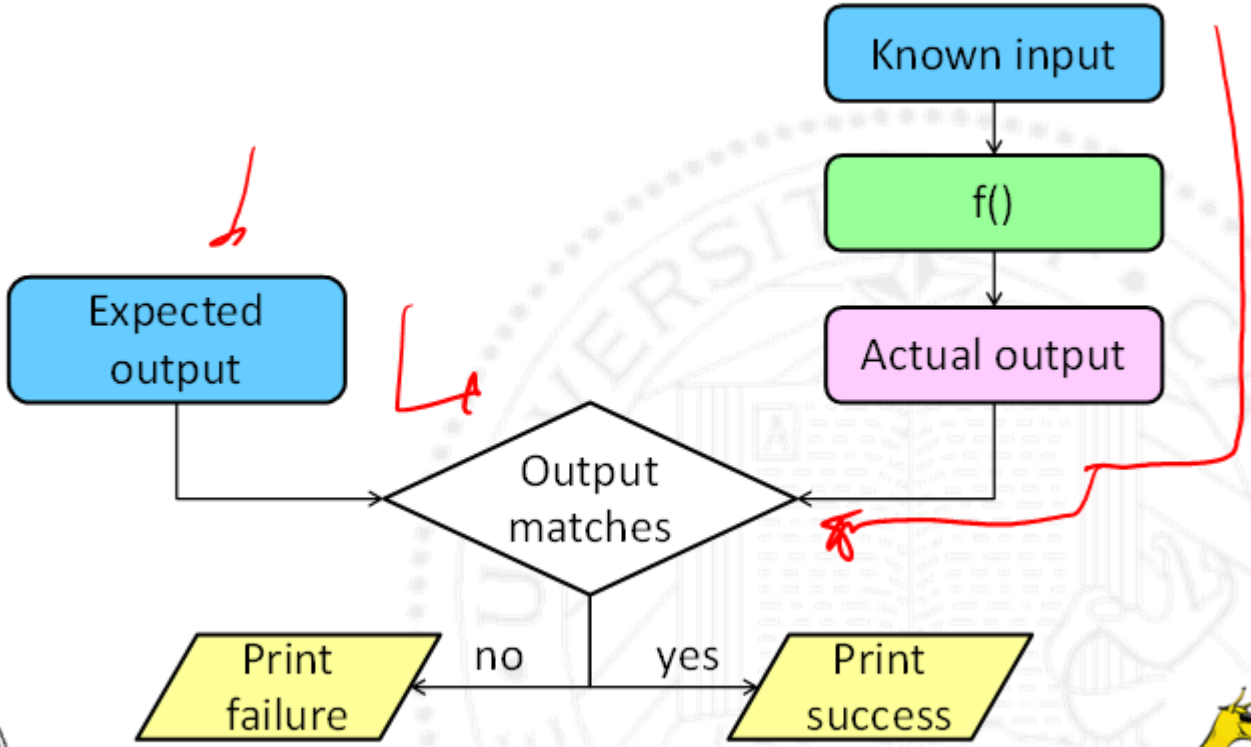
- The most important question:

  "How am I going to test this?"

- Break code into clean functions with:
  - Clear input
  - Clear output
  - No/minimal side effects

Unit testing

Testing architecture



	actually True	actually False
Test says True	✓	Type II error
Test says False	Type I Error	✓

Unit testing

Testing architecture

Example

```
// Declare test constants
testInput ← some input
testExpOutput ← precalculated output

// Calculate result
testActOutput ← function result

// Output test results
if testActOutput equals testExpOutput
  output "Test passed"
else
  output "Test failed!"
```

Unit testing

Trivial example

ExampleLib.c

```c
int AddFive(int x)
{
  return x + 5;
}
```

main.c

```c
#include "ExampleLib.h"

int main(void)
{
  // Declare test constants
  int test1Input = 0;
  int test1ExpOutput = 5;

  // Calculate result
  int test1ActOutput;
  test1ActOutput = AddFive(test1Input);

  // Output test results
  if (test1ActOutput == test1ExpOutput) {
    printf("Test1 passed.\n");
  } else {
    printf("Test1 failed!\n");
  }
}
```

Unit testing

Writing tests

- Write multiple tests
  - At least 1 for every group of inputs
  - Each edge case should have their own test
- Each test should check **one** part of the total functionality
  - One function or logical block of code at a time
- Try to break the code you're testing!

Unit testing

Testing framework

- Track how many tests passed/failed
  - Per function
- Track how many functions passed/failed
  - With all tests must pass for the function to pass
- Each test cleanly separated from other tests
  - Both in code and in logic
- Output results
  - Per function/per test results

Roadmap

- Pointers Review
- Arrays Review
- Unit Testing
- ~~Operators (?)~~
- ~~#defines and literals (?)~~

Operators

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Operators

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Operators

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Operators

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Operators

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Operators

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Operators

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left