CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

Which ones compile? Which ones run?

```
#define A 15
    A++;

    const int B=15;
    B++;
```

```
    int c=15;
#define C c
    C++;

    float D[1] = {15.0};
    D++;
```

Max Lichtenstein

Lab 3 debrief

Email 1:30

Thurs Morning @ six

Max Lichtenstein

Announcements

- Lab 4: Reverse Polish Notation calculator
  - Key concepts:
    - Structs
    - String Parsing / Tokenization
    - Stacks
- Piazza poll?

Roadmap

- Constants
  - And various ways to make them
- String Parsing
  - Tokens
  - Algebraic Expressions
  - Question for Piazza poll
- BREAK
  - Piazza poll
- Stacks
- RPN

Constants

4 kinds of constants:

```
//Literals:
if (foo == 1) return 0;
```

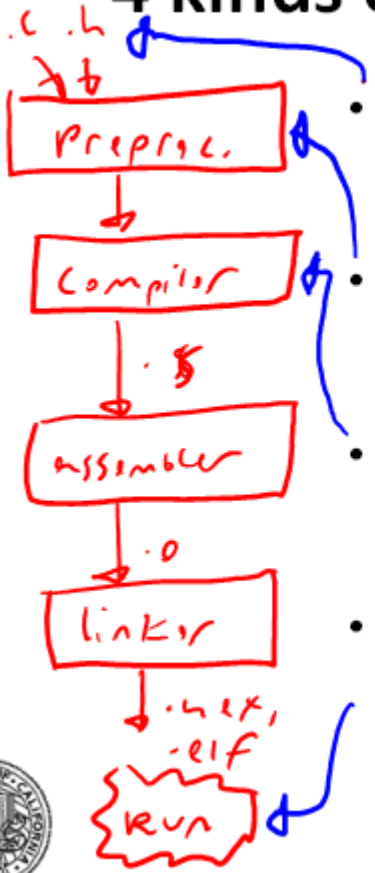
```
//Macros:
#define FALSE 0
#define TRUE 1

if (foo == TRUE) return FALSE;
```

```
//constant variables:
const int True = 1;
const int False = 0;
if (foo == True) return FALSE;
```

```
//enums:
enum {
    false, true, Maybe=3;
};
if (foo == true) return false;
```

Max Lichtenstein

4 kinds of constants: Summary



- Literals:
  - Can be magic numbers, avoid these
  - Ok *occasionally*
- Macros:
  - Really just literals, created during preprocessing
  - Fast (during runtime anyway)!
- Enums
  - Can be typed
  - Compiler optimizes them for you
- Consts:
  - Can be arbitrarily large
  - Live on stack
  - Aren't exacty all that constant

enum
day, week
CMON, TUES, WED, ...

How are they different?

- Where they are stored
- How long they live
- How constant are they, really?
- When do they get their values?

4 kinds of constants:

```
//Literals:
if (foo == 1) return 0;
```

```
//Macros:
#define FALSE 0
#define TRUE 1

if (foo == TRUE) return FALSE;
```

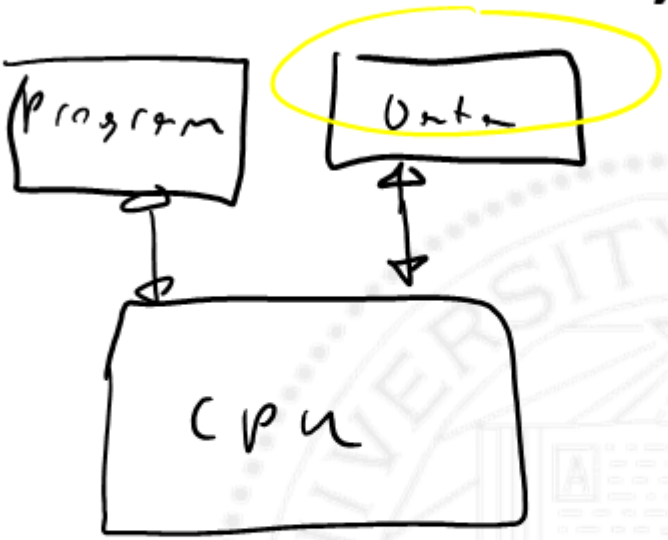
```
//constant variables:
const int True = 1;
const int False = 0;
if (foo == True) return FALSE;
```

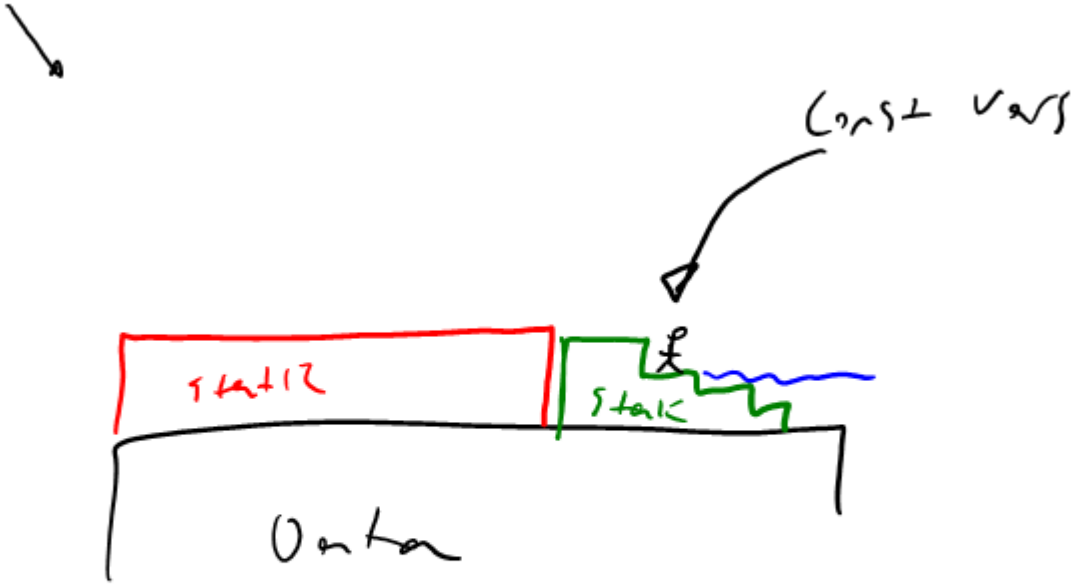
```
//enums:
enum {
    false, true
};
if (foo == true) return false;
```

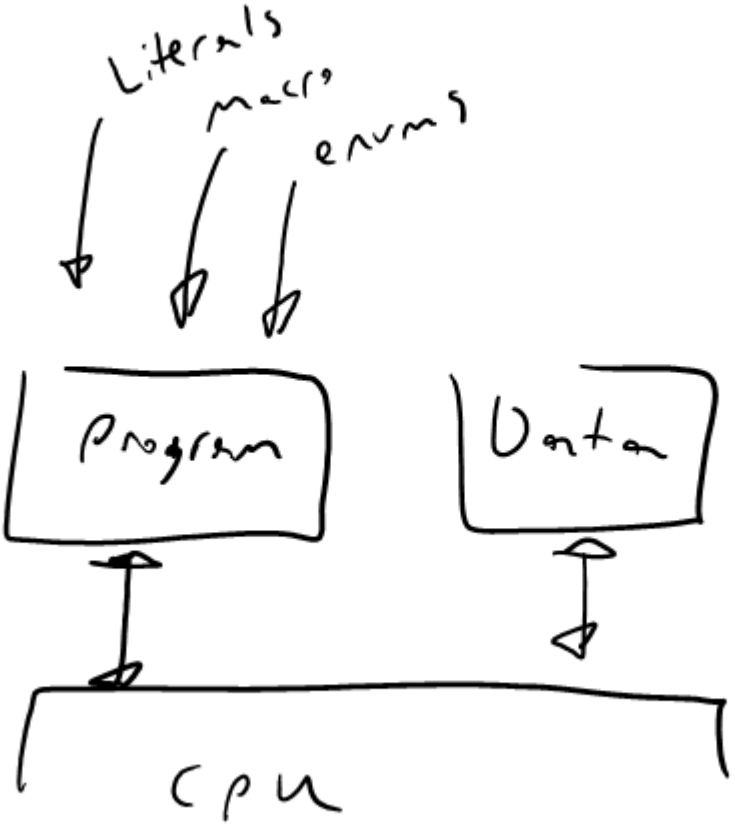
Max Lichtenstein

World of Memory

Harvard




Const vars
static
Stack
Data


Literals
macro
enums
Program
Data
CPU

Literals and Macros

```
//Literals:
if (foo == 1) return 0;

//Macros:
#define FALSE 0
#define TRUE 1

if (foo == TRUE) return FALSE;
```

Enums

```
//enums:
enum {
    false, true
};
if (foo == true) return false;
```

Const variables

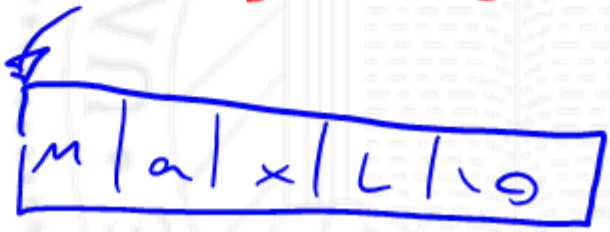
```
//constant variables:
const int True = 1;
const int False = 0;
if (foo == True) return FALSE;
```

Kind of constant:	Pros	Cons
Macro	Fast + Small, Works on any data type	Inflexible, often confusing
Enum	Typesafe-able, slightly optimized	Also inflexible
Const	Temporary, Not actually that constant	Takes up space, Slower

String Parsing

Array of char

char myName[] = "MaxL"



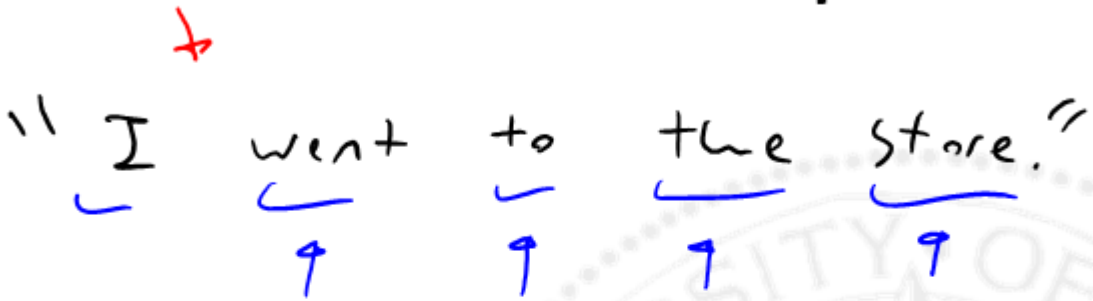
Max Lichtenstein

Tokenization

- Breaking up a long string into smaller strings
  - "tokens" separated by "delimiters"

Tokenization Example 1

"I went to the store."



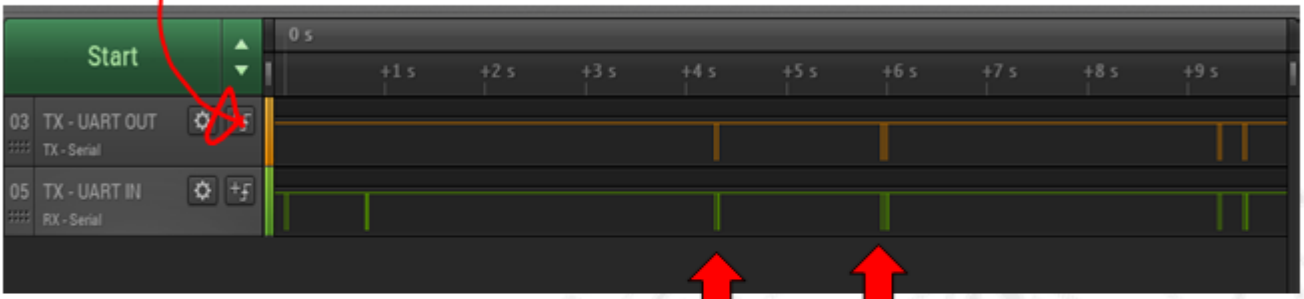
Tokenization Example 2

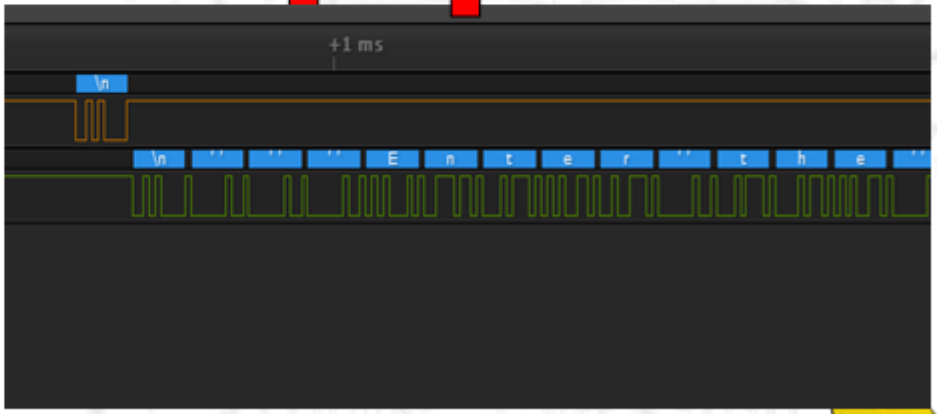
```
max@calamity MINGW64 /e/Dropbox/CE13/mnlichte (master)
$ git add -A
```

Tokenization Example 3

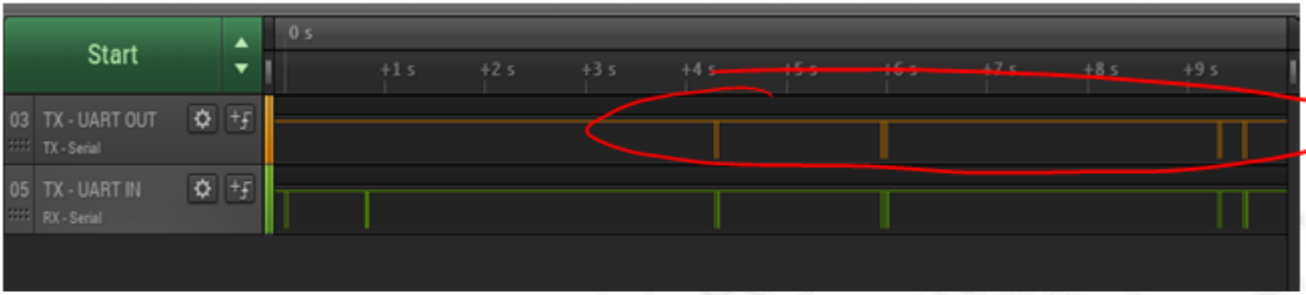
```
max@calamity MINGW64 /e/Dropbox/CE13/mnlichte (master)
$ git commit -m "I'm getting the hang of this!"
```

Tokenization Example 4





Tokenization Example 4



"+ \n 15.01 \n 30 \n"

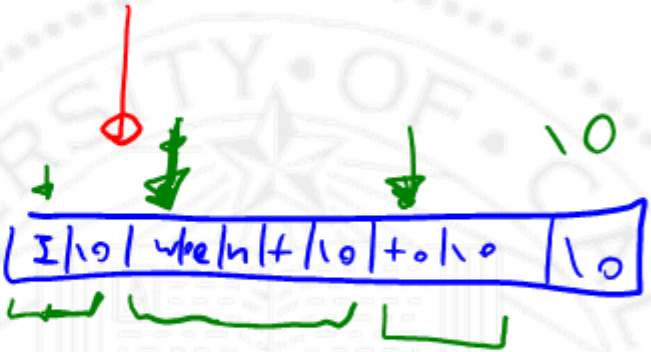
operator op1 op2

strtok()

- 2 kinds of calls: char sen[] = "I went to the store";
  - Initial

```
strtok(sen, " ");
```

  - Subsequent




```
char* next_token = strtok(NULL, " ");
```

```
next_token++;
```

Parsing Algebraic Expressions

- What about algebraic strings?

"(11 + 7) * (3 + 4)"

"(", "11", "+", "7", ...

String Parsing

```
int Evaluate(char algebra_string[]);
```

myString	Evaluate(myString)
1 + 1 =	2
1 + 1 + 1 =	3
1 + (2 + 3) =	6
1 + (2 * 3) + (4 * 5)	26
1 + (1 - (1 + 1) - 1) + 1 =	0
1 + (1 + 1 =	Error

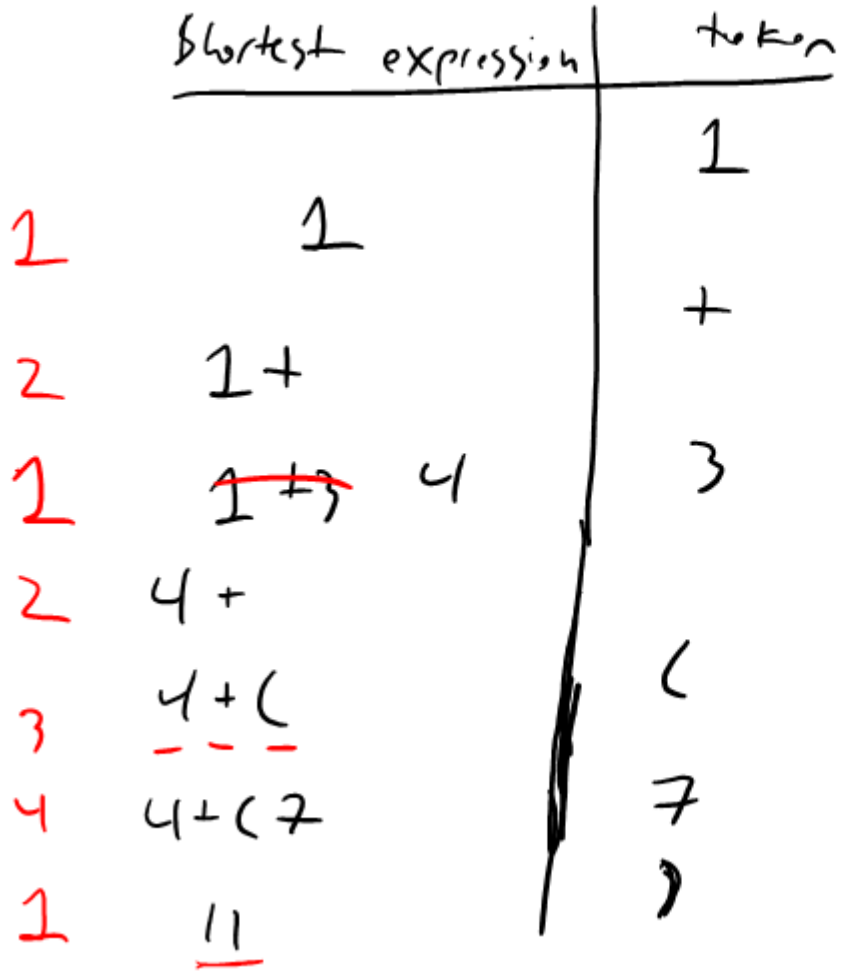
String Parsing

- Or alternatively, operate on one token at a time:

```
int EvaluateExpression(char next_token[]);
```

```
EvaluateExpression("1");
EvaluateExpression("+");
EvaluateExpression("23");
EvaluateExpression("+");
EvaluateExpression("(");
.
.
.
EvaluateExpression(")");
int answer = EvaluateExpression("=");
```

```
for(i = 0; i < n_tokens; i++){
    this_token = all_tokens[i];
    EvaluateExpression(this_token);
}
int answer = EvaluateExpression("=");
```


shortest expression
token
1
1
1
+
2
1+
1
1+3
4
3
2
4+
4+(
3
4+(7
4
11
1
)

Poll question:

We expect that

```
int EvaluateExpression(char next_token[]);
```

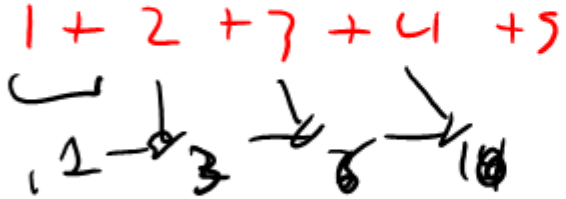
will need to work on strings that have a couple thousand tokens. How many variables do we need (or how much memory do we need)?

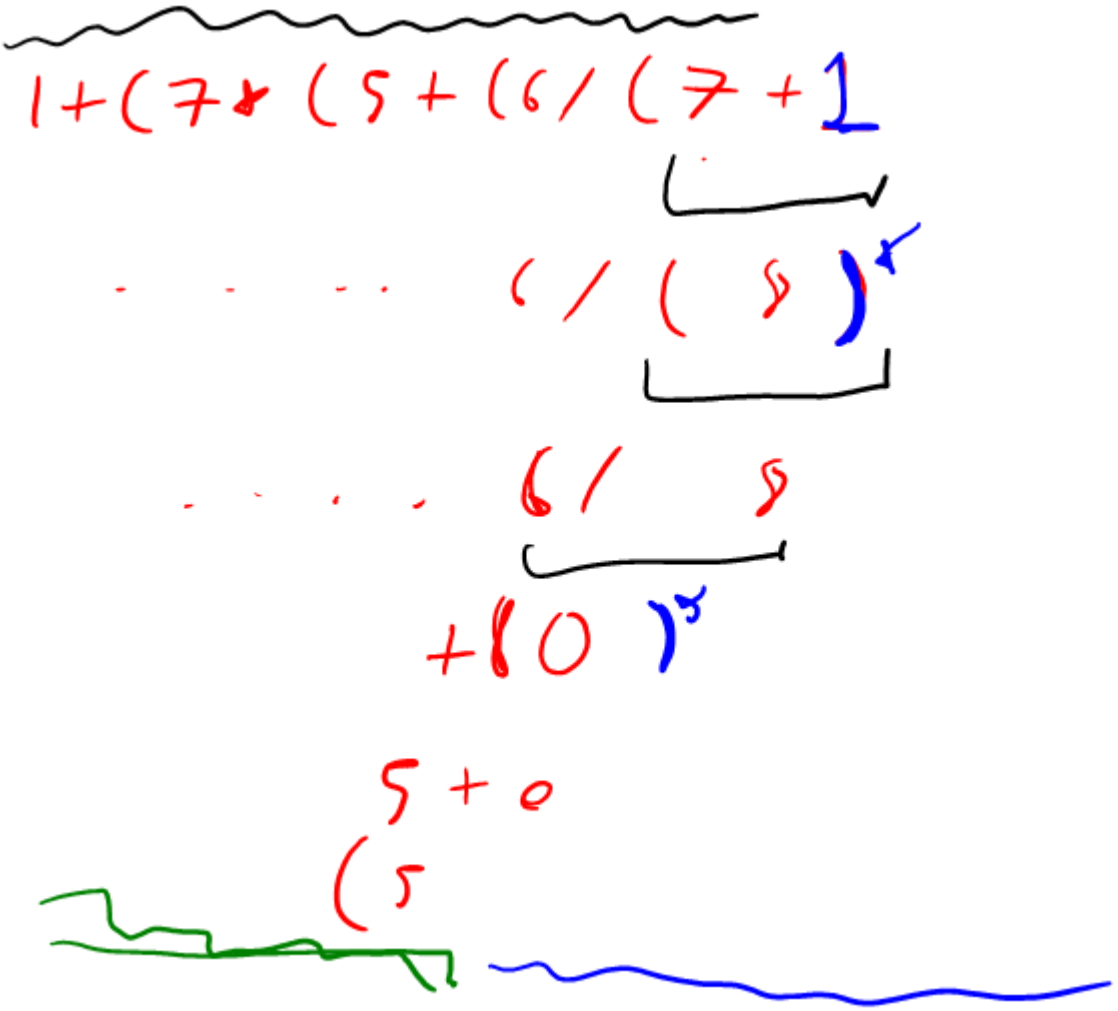
- A: Only one or two variables
- B: A few thousand variables
- C: Millions of variables

Max Lichtenstein

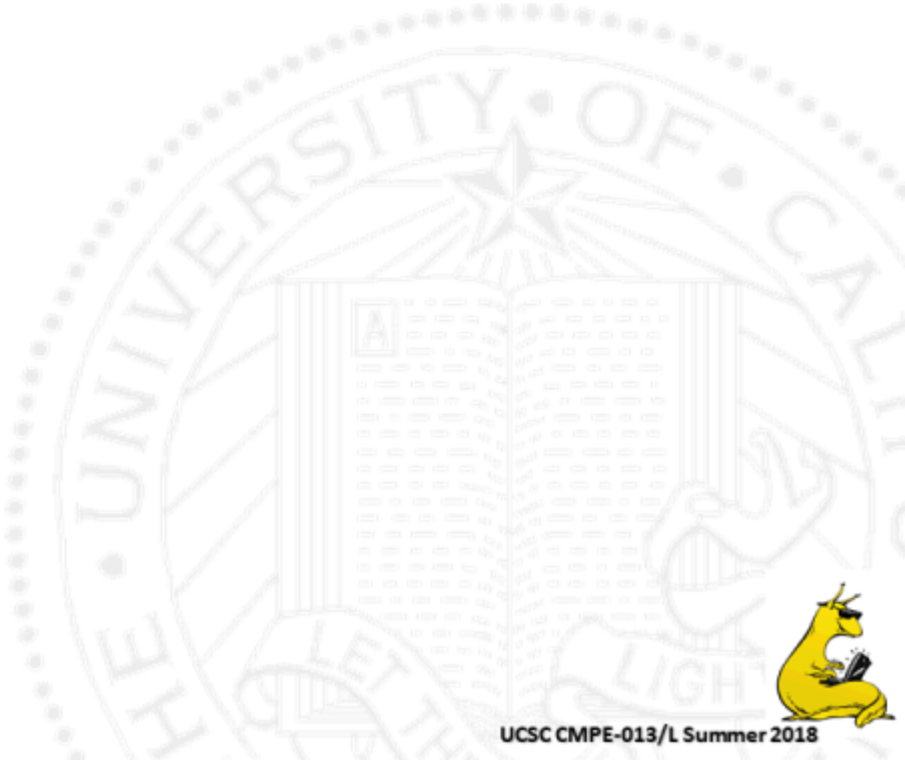

1 + (2 * 3)


1 + 6


1 + 2 + 7 + 4 + 5
1
2
3
6
10



BREAK



Poll results

Where do we keep all these variables?

previous_tokens[];

Stacks

stacks and The Stack

Stack Operations

Minimum, 2 operations:

- Push()
- Pop()

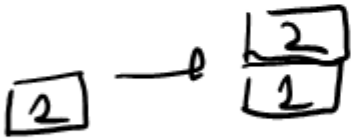
Also maybe:

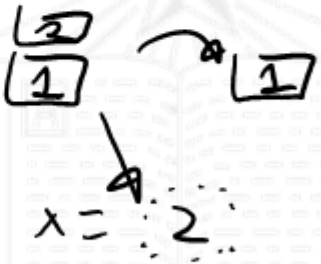
- Initialize()
- ~~Peek()~~

Stack operations

push(2)

int x = pop()



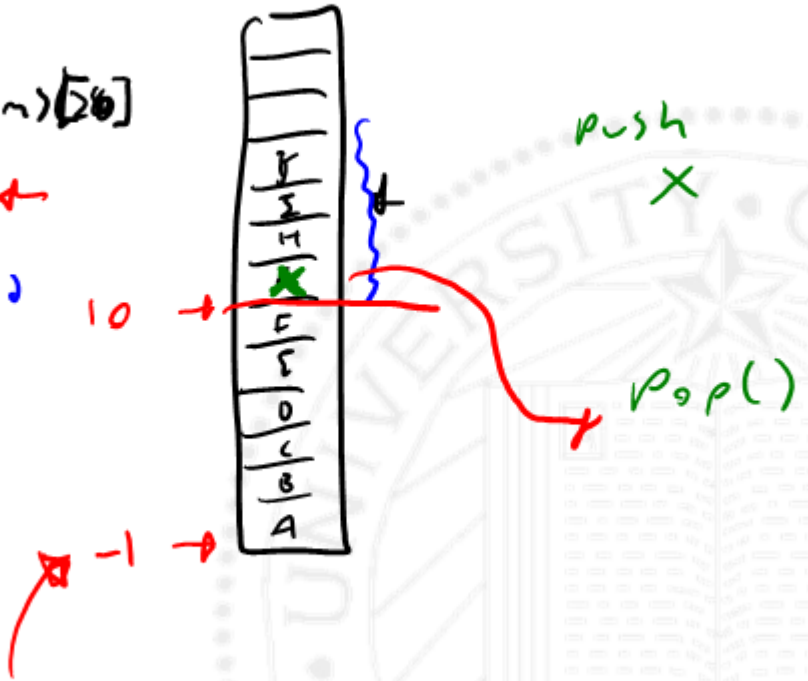


Stacks

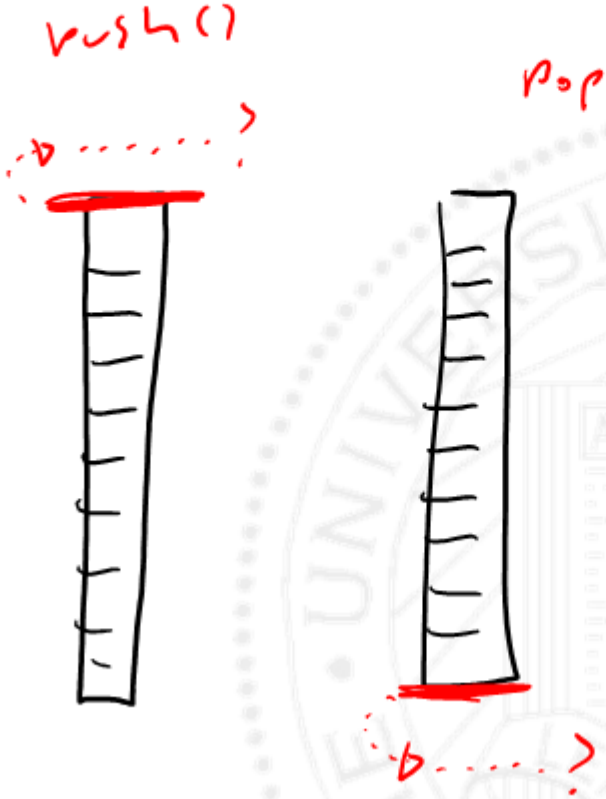
Stack in Algebra Evaluation

Simple Stack in C

```
double stack_items[26]
int index = -1
int initialized
```



Stack overflow and underflow



Which functions need a stack?

- IsPalindrome("Do geese see god") = True

do geese see god

- PrintTwice("Hello") = "HelloHello"

↑ ↑

- FindMax({12, 3, 16, 2}) = 16

RPN

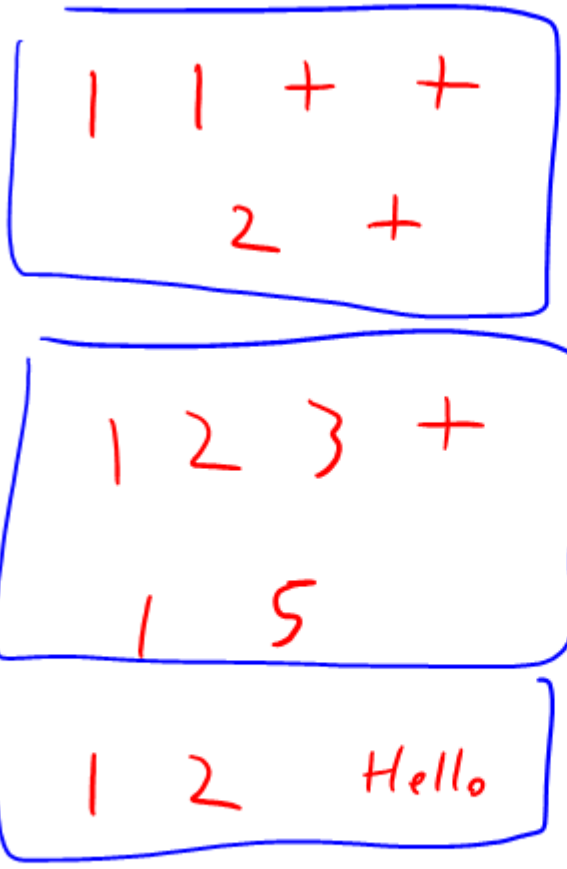
((1 22 +)(8 9 -)+)

1+22 → (23

8-9 → -1 +)

23 + (-1)

22



1 2 3 + +
1 5 +
6

1 (2 3 *) +
1 6 +
7

Structs

Structures

Definition

Structures are collections of variables grouped together under a common name. The variables within a structure are referred to as the structure's **members**, and may be accessed individually as needed.

- Structures:
  - May contain any number of members
  - Members may be of **any** data type
  - Allow a group of related variables to be treated as a single unit, even if different types
  - Ease the organization of complicated data

Structures

Declaring

Syntax

```
struct StructName {
   type1 memberName1;
   ...
   typen memberNamen;
};
```

Members are declared just like ordinary variables

Example

```
// Structure to handle complex numbers
struct Complex {
  float re;     // Real part
  float im;     // Imaginary part
};
```

Structures

Instantiating

Syntax

```
struct StructName {
   type1 memberName1;
   ...
   typen memberNamen;
} varName1,...,varNamen;
```

Example

```
// Structure to handle complex numbers
struct Complex {
  float re;
  float im;
} x, y;          // Declare x and y of type complex
```

Structures

Instantiating cont'd

Syntax

If *StructName* has already been defined:

```
struct StructName varName1, …, varNamen;
```

Example

```
struct Complex {
  float re;
  float im;
}
...
struct Complex x, y;  // Declare x and y of type complex
```

Structures

Accessing members

Syntax

structVariableName.memberName

Example

```
struct Complex {
  float re;
  float im;
} x, y;              // Declare x and y of type `struct complex`

int main(void)
{
  x.re = 1.25;       // Initialize real part of x
  x.im = 2.50;       // Initialize imaginary part of x
  y = x;             // Set struct y equal to struct x
  ...
```

Structures

Initialization

Syntax

If *StructName* has already been defined:

```
struct StructName varName = {const1,…,constn};
```

Example

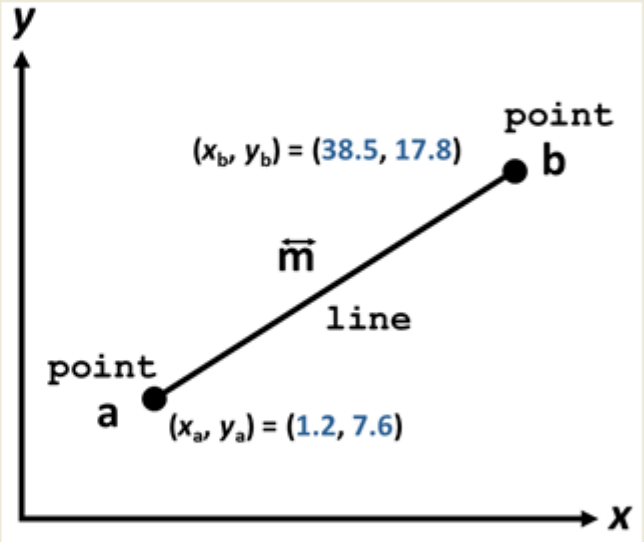
```
struct Complex {
  float re;
  float im;
};
...
struct Complex x = {1.25, 2.50};
```

Structures

Nesting Structures

Example

```
struct point {
  float x;
  float y;
};

struct line {
  struct point a;
  struct point b;
};
```




```
int main(void)
{
  struct line m = {{1.2, 7.6}, {38.5, 17.8}};
  ...
}
```

Structures

Nesting Structures

Example

```
struct point {
  float x;
  float y;
};

struct line {
  struct point a;
  struct point b;
};

int main(void)
{
  struct line m = {{1.2, 7.6}, {38.5, 17.8}};
  printf("Line (%f, %f) <-> (%f, %f)",
    m.a.x, m.a.y, m.b.x, m.b.y);
  ...
}
```

Structures

Arrays and Pointers with Strings

- Strings:
  - May be assigned directly to `char` array member only at declaration
  - May be assigned directly to a pointer to `char` member at any time

Example: Structure

```
struct Strings {
  char a[4];
  char *b;
} str = {"Bad", "Good"};
```

Example: Initializing Members

```
int main(void)
{
  struct Strings str;
  str.a[0] = 'B';
  str.a[1] = 'a';
  str.a[2] = 'd';
  str.a[3] = '\0';
  str.b = "Good";
```

Structures

Creating Arrays of Structures

Syntax

If *StructName* has already been defined:

```
struct StructName arrName[n];
```

Example

```
struct Complex {
  float re;
  float im;
};
...
struct Complex a[3];
```

Structures

Initializing Arrays of Structures at Declaration

Syntax

If *StructName* has already been defined:

```
struct StructName arrName[n] = {{list_1},…,{list_n}};
```

Example

```
struct Complex {
  float re;
  float im;
};
...
struct Complex a[3] = {{1.2, 2.5}, {3.9, 6.5}, {7.1, 8.4}};
```

Structures

Using Arrays of Structures

If *arrName* has already been defined:

Syntax

```
arrName[n].memberName
```

Example: Definitions

```
typedef struct {
  float re;
  float im;
} Complex;
...
struct Complex a[3];
```

Example: Usage

```
int main(void)
{
  a[0].re = 1.25;
  a[0].im = 2.50;
  ...
}
```

Structures

Creating a Pointer to a Structure

Syntax

If *StructName* has already been defined:

```
struct StructName *ptrName;
```

Example

```
struct Complex {
  float re;
  float im;
};
...
struct Complex *a;
```

Structures

How to Use a Pointer to Access Structure Members

If *ptrName* has already been defined:

Syntax

ptrName->memberName

Pointer must first be initialized to point to the address of the structure itself: *ptrName = &structVariable;*

Example: Definitions

```
struct Complex {
  float re;
  float im;
};
...
struct Complex x;
struct Complex *p;
```

Example: Usage

```
int main(void)
{
  p = &x;
  // Set x.re = 1.25 via p
  p->re = 1.25;
  // Set x.im = 2.50 via p
  p->im = 2.50;
}
```

Structures

How to Pass Structures to Functions

Example

```
struct Complex{
  float re;
  float im;
};

void Display(struct Complex x)
{
  printf("(%f + j%f)\n", x.re, x.im);
}

int main(void)
{
  struct Complex a = {1.2, 2.5};
  struct Complex b = {3.7, 4.0};

  Display(a);
  Display(b);
}
```

Structures

How to Pass Structures to Functions

Example

```
struct Complex {
  float re;
  float im;
};

void Display(struct Complex *x)
{
  printf("(%f + j%f)\n", x->re, x->im);
}

int main(void)
{
  struct Complex a = {1.2, 2.5};
  struct Complex b = {3.7, 4.0};

  Display(&a);
  Display(&b);
}
```

Structures

How to Pass Structures to Functions

Example

```c
typedef struct {
  float re;
  float im;
} Complex;

void Display(const struct Complex *x)
{
  printf("(%f + j%f)\n", x->re, x->im);
}

int main(void)
{
  struct Complex a = {1.2, 2.5};
  struct Complex b = {3.7, 4.0};

  Display(&a);
  Display(&b);
}
```