CS143A
Principles of Operating Systems
A Brief C Crash Course

Instructor: Prof. Ardalan Amiri Sani
TA: Ping-Xiang Chen (Shawn)

Acknowledgement

The slides are based on the previous discussions from Dr. Claudio A. Parra.

Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

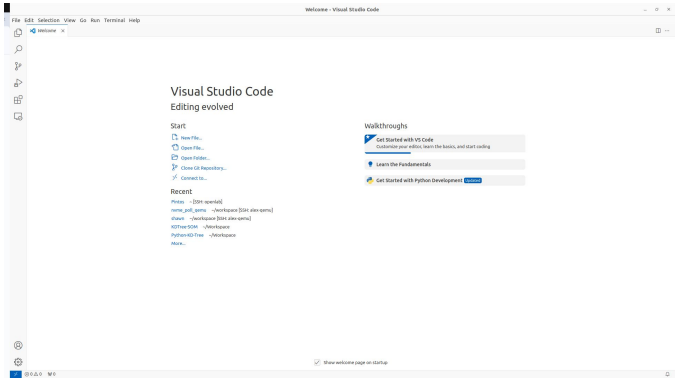
Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

Get Some Editor

- Go and get an editor.
- Get familiar with it.
- Learn its tricks.
- Get comfortable using it in a terminal.

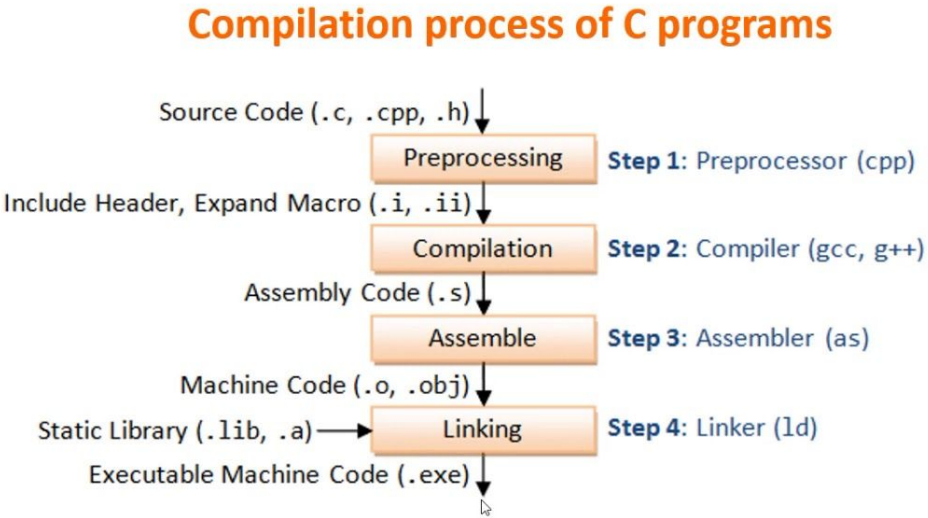
Vim



Microsoft Visual Studio Code

Compilation Process

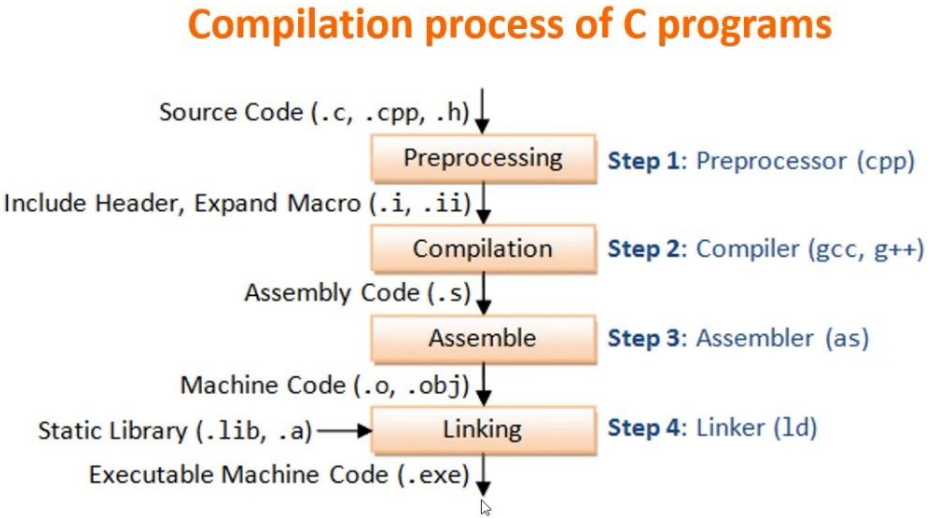
- Prepossessing
  - Remove comments.
  - Expands Macros. (#define)
  - Expand Included files. (#include)
- Compilation
  - Generates text files with assembly language.
  - Specific to the target machine.



NerdyElectronics, 15.1 - Compilation Process of a C Program - Theory - Master C and Embedded C Programming
https://www.youtube.com/watch?v=LOWQg8BNQJ4

Compilation Process

- Assembly
  - Convert the assembly into machine code.
  - This is 0s and 1s.
  - Also known as “Object code”
- Linking
  - Merges all the object codes from multiple modules into a single binary.
  - If we are using libraries, those libraries get linked (referenced or copied).



NerdyElectronics, 15.1 - Compilation Process of a C Program - Theory - Master C and Embedded C Programming
https://www.youtube.com/watch?v=LOWQg8BNQJ4

Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

Basic Types

- *char*, *short*, *int*, *long*, *size_t* store integers.
- *float*, *double* store numbers with fractional parts.
  - You don’t need this to work in an OS.
- *xxxx** are pointers, they store addresses of memory.
- These definitions are machine dependent.

Basic Types

```c
#include <stdio.h>
#include <stdint.h>
#include <float.h>
#include <limits.h>

int main(int argc, char *argv[]) {
  printf("|%10s|%7d bits|%22s|%22s|\n", "",CHAR_BIT,"","");
  printf("|%10s|%12s|%22s|%22s|\n", "type","bytes","min","max");
  printf("|----------+------------+----------------------+----------------------|\n");
  printf("|%10s|%12ld|%22d|%22d|\n", "char",sizeof(char),CHAR_MIN,CHAR_MAX);
  printf("|%10s|%12ld|%22d|%22d|\n", "uchar",sizeof(unsigned char),0,UCHAR_MAX);
  printf("| | | | |\n");
  printf("|%10s|%12ld|%22d|%22d|\n", "short",sizeof(short),SHRT_MIN,SHRT_MAX);
  printf("|%10s|%12ld|%22d|%22d|\n", "ushort",sizeof(unsigned short),0,USHRT_MAX);
  printf("| | | | |\n");
  printf("|%10s|%12ld|%22d|%22d|\n", "int",sizeof(int),INT_MIN,INT_MAX);
  printf("|%10s|%12ld|%22d|%22u|\n", "uint",sizeof(unsigned int),0,UINT_MAX);
  printf("| | | | |\n");
  printf("|%10s|%12ld|%22ld|%22ld|\n", "long",sizeof(long),LONG_MIN,LONG_MAX);
  printf("|%10s|%12ld|%22d|%22lu|\n", "ulong",sizeof(unsigned long),0,ULONG_MAX);
  printf("| | | | |\n");
  printf("|%10s|%12ld|%22lld|%22lld|\n", "llong",sizeof(long long),LLONG_MIN,LLONG_MAX);
  printf("|%10s|%12ld|%22d|%22llu|\n", "ullong",sizeof(unsigned long long),0,ULLONG_MAX);
  printf("| | | | |\n");
  printf("|%10s|%12ld|%22d|%22lu|\n", "size_t",sizeof(size_t),0,SIZE_MAX);
  printf("|----------+------------+----------------------+----------------------|\n");
  printf("|%10s|%12ld|%8s+-%12g|%8s+-%12g|\n", "float",sizeof(float),"",FLT_MIN,"",FLT_MAX);
  printf("|%10s|%12ld|%8s+-%12g|%8s+-%12g|\n", "double",sizeof(double),"",DBL_MIN,"",DBL_MAX);
  printf("|----------+------------+----------------------+----------------------|\n");
  printf("|%10s|%12ld|%22s|%22s|\n", "void*",sizeof(void*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "char*",sizeof(char*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "short*",sizeof(short*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "int*",sizeof(int*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "long*",sizeof(long*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "long long*",sizeof(long long*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "size_t*",sizeof(size_t*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "float*",sizeof(float*),"- ","- ");
  printf("|%10s|%12ld|%22s|%22s|\n", "double*",sizeof(double*),"- ","- ");
  printf("+----------+------------+----------------------+----------------------+\n");
  return 0;
}
```

```
|          |     8 bits|                      |                      |
|      type|      bytes|                   min|                   max|
|----------+-----------+----------------------+----------------------|
|      char|          1|                  -128|                   127|
|     uchar|          1|                     0|                   255|
| | | | |
|     short|          2|                -32768|                 32767|
|    ushort|          2|                     0|                 65535|
| | | | |
|       int|          4|           -2147483648|            2147483647|
|      uint|          4|                     0|            4294967295|
| | | | |
|      long|          8|  -9223372036854775808|   9223372036854775807|
|     ulong|          8|                     0|  18446744073709551615|
| | | | |
|     llong|          8|  -9223372036854775808|   9223372036854775807|
|    ullong|          8|                     0|  18446744073709551615|
| | | | |
|    size_t|          8|                     0|  18446744073709551615|
|----------+-----------+----------------------+----------------------|
|     float|          4|        +- 1.17549e-38|        +- 3.40282e+38|
|    double|          8|        +-2.22507e-308|        +-1.79769e+308|
|----------+-----------+----------------------+----------------------|
|     void*|          8|                     - |                     - |
|     char*|          8|                     - |                     - |
|    short*|          8|                     - |                     - |
|      int*|          8|                     - |                     - |
|     long*|          8|                     - |                     - |
|long long*|          8|                     - |                     - |
|   size_t*|          8|                     - |                     - |
|    float*|          8|                     - |                     - |
|   double*|          8|                     - |                     - |
+----------+-----------+----------------------+----------------------+
```

Variables and Constants

- Variables and constants are the basic data objects in a program.
- Constants are read only. Variables are rewritable.
- Both have a data type associated to it. (integer, decimal, character...)

```c
#define MAXSIZE 1000 //int constant
#define THREE 3L //long constant
#define PI 3.1415 //double constant
#define HALF 0.5F //float constant

int main() {
    int lucky_number; //declare and define
    lucky_number = 42; //initialize (assign) later

    char initial = 'C';

    //we can use the constants
    double use_constants = PI;
    float use_dot_for_floats = 5.0;

    // invalid, we cannot assign constants.
    THREE = 4; // ERROR

    // case sensitive, these are different
    long DIFFERENT = 3984756768;
    long different = 8731408705;

    // variable already used
    float lucky_number = 42.51; // ERROR

    nope = 300; // ERROR, variable never declared!!
    char 4nope = 'X'; // ERROR, invalid variable name!!
    return 0;
}
```

Variables and Constants

- Declaration:
  - Introduction of a new data object name to the program.
- Definition:
  - Explanation of what is the size and shape of the declared data object.
- Assignment:
  - Act of binding a value to a name.
- Initialization:
  - First assignment of a value to the name.

```c
#define MAXSIZE 1000 //int constant
#define THREE 3L //long constant
#define PI 3.1415 //double constant
#define HALF 0.5F //float constant

int main() {
    int lucky_number; //declare and define
    lucky_number = 42; //initialize (assign) later

    char initial = 'C';

    //we can use the constants
    double use_constants = PI;
    float use_dot_for_floats = 5.0;

    // invalid, we cannot assign constants.
    THREE = 4; // ERROR

    // case sensitive, these are different
    long DIFFERENT = 3984756768;
    long different = 8731408705;

    // variable already used
    float lucky_number = 42.51; // ERROR

    nope = 300; // ERROR, variable never declared!!
    char 4nope = 'X'; // ERROR, invalid variable name!!
    return 0;
}
```

Enumeration

- Useful to assign meaningful names to integral constants.
- Thus, the code is cleaner and easier to maintain/understand.
- Often used in the kernel of an OS.
- Values start from 0 unless values are specified explicitly.
- For not explicit specification, the values continue in progression.

```c
#include <stdio.h>
enum course_status { FAIL, PASS, INCOMPLETE, DROP };
enum score { BAD = 1, AVERAGE, GREAT };
//AVERAGE is 2, GREAT is 3

int main(void) {
  enum course_status pass_course = PASS;
  enum score how_was_it;
  how_was_it = GREAT;
  printf("Course Status? %d.\n", pass_course);
  printf("How was the course? %d.\n", how_was_it);
  return 0;
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./enum
Course Status? 1.
How was the course? 3.
```

Operators and their Precedence

- Operator associativity is used when two operators of the same precedence appear in an expression.
- Associativity can be either from Left to Right or Right to Left.

```c
#include <stdio.h>

int main(void){
  int a = 3, b = 4, c = 5;
  a = b = c;
  printf("a=%d, b=%d, c=%d\n", a, b , c);
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./associativity
a=5, b=5, c=5
```

C Operator Precedence, https://en.cppreference.com/w/c/language/operator_precedence

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

a = -5

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

a = -4

b = -28

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

a = -3

c = -21

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

d = 1

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

e = 1

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

f = 0

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

- The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result.
- The use of the comma token as an operator is distinct from its use in function calls and definitions, variable declarations, enum declarations, and similar constructs, where it acts as a separator.

i = 1

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

j = 2

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

Operators and their Precedence

```c
#include<stdio.h>
int main(){
  int a,b,c,d,e,f,g,h,i,j,k;
  a = 3 - 4 * 2;
  b = ++a * 7;
  c = a++ * 7;
  d = 3 < 2 != 2;
  e = 1 || 0 && 1;
  f = g = h = 7 == 1;
  i = 1 , 2;
  j = (1 , 2);
  k = 7 > 8 ? 0 : 3 != 3 ? 15 : 17;

  printf("a:%d\nb:%d\nc:%d\nd:%d\n"
    "e:%d\nf:%d\ng:%d\nh:%d\n"
    "i:%d\nj:%d\nk:%d\n",
    a,b,c,d,e,f,g,h,i,j,k);
  return 0;
}
```

k = 17

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

Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

Statements

- A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, collect input, assigning a value to a variable, etc.
- A computer program is made up of a series of statements. Statements are delimited by a semicolon at the end.

```
a = 3 - 4 * 2;
```

Blocks

- A compound statement or Block is the way C groups multiple statements into a single statement. It consists of multiple statements and declarations within braces.

```
{
    statement;
    statement;
    statement;
    ...
}
```

Selection Statements

if (expression) statement

- statement is executed only iff expression is non-zero.

if (expression) 1st-statement else 2nd-statement

- Similarly, but now 2nd-statement is executed iff expression is zero.

switch (expression) statement

- expression is integer or character. The statement is usually compound and it contains case-labeled statements and optionally a default-labeled statement.

```
if(var > 0){
    var -= 1;
    f = foo(var);
}
```

```
if(i * n + j < n * n){
    if(i<j)
       bar(i * n + j);
    else
       i += 1;
}

else{
    i = 0;
    j = 0;
}
```

```
switch(my_char){
    case 'a':
       foo(var);
       break;
    case 'b':
       bar(var);
       break;
    default:
       printf("not known\n");
}
```

Iteration Statements

while (expression) statement

- statement is executed repeatedly only iff expression is non-zero.

do statement while (expression)

- Similarly, but now statement is executed at least once.

for (exp1 ; exp2 ; exp3) statement

- exp1 is executed once, before the “for” iteration. statement is executed repeatedly as long as exp2 is non-zero. exp3 is executed right after every execution of the statement.

```
while(var < MAX){
  var += 1;
  foo(var);
}
```

```
do{
  c = read_char();
  store_char(c);
} while(c != 'x')
```

```
for(int i = 0; i < length; ++i){
  if(is_prime(i)){
  store(i);
}
```

Jump Statements

break;

- Used within iteration statements and switch statements to pass control flow to the statement following the while, do-while, for, or switch.

continue;

- Used within iteration statements to transfer control flow to the place just before the end of the statement. In for loops, right before exp3

return expression;

- Used to return control to the caller of the current function. If it is accompanied by an expression, its value become available to the caller.

```
while(1){
    if(r < 0)
       break;
    else
       r -= 1;
}
```

```
int i = 0;
for(int pair = getN(); i < len; pairs = getN()){
    if(pair % 2 != 0)
       continue;
    else{
       process(pair);
       i += 1;
    }
}
```

Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

When to create functions

- Break problems into small parts. Reuse your code. Easy readability. Parameters and return values are always copied.

```
int main(int argc, char **argv) {
  //check validity of the arguments
  for(all a in arguments){
        if(wrong argument)
        print error and exit;
  }
  //get user input
  print "what operation to execute";
  op = user input;
  if(op is wrong operation)
        print error and exit;
    //execute the requested operation
    if(operation is X){
        allocate memory;
        some computation;
    }
    else if(operation is Y){
        allocate memory;
        another computation;
    }
    return 0;
}
```

```
ret_type name(args declaration){
        declarations and statements
}
```

```
void check_args(int argc, char **argv) {
  for(all a in argv){
        if(wrong argument)
        print error and exit;
  }
}
int input_operation(void) {
        print "what operation to execute";
        op = user input;
  if(op is wrong operation)
        print error and exit;
        return op;
}
void run_operation(int oper) {
  if(operation is X){
        allocate memory;
        some computation;
  }
  else if(operation is Y){
        allocate memory;
        another computation;
  }
}
int main(int argc, char **argv) {
  check_args(argc, argv);
  ask what operation to execute;
  op = input_operation();
  run_operation(op);
  return 0;
}
```

External and Internal Variables

- A program written in C consists of a set of external objects, which are either variables or functions.
- These objects can be across several source files (“.c” files).
- A variable is external or internal if it is defined outside or inside of any function. All functions are external.
- An external variable is accessible from any function in the file after their declaration.
- There must be only one DEFINITION of each external object.
- Internal variables are destroyed on function return. External variables are permanent.

```c
#include <stdio.h>

int extvar;

void fn1(void){
  int invar = 42;
  extvar = 3;
}

void fn2(void){
  int invar = 57;
  extvar = 5;
}

int main(void){
  int invar;
  extvar = 2;
  invar = 57;
  printf("ext:%d int:%d\n",extvar,invar);
  fn1();
  printf("ext:%d int:%d\n",extvar,invar);
  fn2();
  printf("ext:%d int:%d\n",extvar,invar);
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./external
ext:2 int:57
ext:3 int:57
ext:5 int:57
```

Declare, Define, Initialize

- Declare: telling the program a variable or function exists, and what is its shape.
- Define: setting aside memory for the variable.
- Initialize: put the first value on the variable.
- If you use a variable in several files, you must declare it for all files. But you must define it only in one place.

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./declare
ext:2 int:57
ext:2 int:57
ext:5 int:57
arr[3]:33.000000
```

```c
#include <stdio.h>
void fn1(void){ // declare + define
  int invar = 42;
  //extvar = 3; //error, not declared yet
}

// declare, telling the program the
// variables and functions exist.
void fn2(void);
extern int extvar;
extern double arr[];
// define variables,
int extvar;
double arr[4];

int main(void){
  int invar; // declaration + definition
  arr[3] = 33;
  extvar = 2; // initialization
  invar = 57; // initialization
  printf("ext:%d int:%d\n",extvar,invar);
  fn1();
  printf("ext:%d int:%d\n",extvar,invar);
  fn2();
  printf("ext:%d int:%d\n",extvar,invar);
  printf("arr[3]:%f\n",arr[3]);
}

// define functions
void fn2(void){
  int invar = 57;
  extvar = 5; // this is fine
}
```

Header Files and Static Objects

- Variables and functions are declared in the header.
- They are defined in source2.c
- main.c must #include the header.
- The header file acts as a “contract” between main and source2, defining how the variables and functions can be used (for main) and how they will be defined (for source2)
- If you want to make an object only visible for that source file, use the word static.

main.c

```c
#include <stdio.h>
#include "header_header.h"
int main(void){
  var = 4;
  var += fn1(15);
  //num_calls = 0; // error!!
  printf("var: %d\n",var);
  printf("fn2: %d\n",fn2());
}
```

source2.c

```c
int var;
static int num_calls = 0;

int fn1(int i){
  num_calls += 1;
  return i + 3;
}

int fn2(void){
  num_calls += 1;
  return 2 * var;
}
```

header_header.h

```c
extern int var;
extern int fn1(int i);
extern int fn2(void);
```

$ gcc -o main main.c source2.c -include header_header.h

```
shawn@shawn-mini-desktop:~/Workspace/CS143A/header$ ./main
var: 22
fn2: 44
```

Makefile

- Make is a build automation tool that builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program.
- For more information, please refer to the:
  - Makefile Tutorial

main.c

```c
#include <stdio.h>
#include "header_header.h"
int main(void){
  var = 4;
  var += fn1(15);
  //num_calls = 0; // error!!
  printf("var: %d\n",var);
  printf("fn2: %d\n",fn2());
}
```

header_header.h

```c
extern int var;
extern int fn1(int i);
extern int fn2(void);
```

source2.c

```c
int var;
static int num_calls = 0;

int fn1(int i){
  num_calls += 1;
  return i + 3;
}

int fn2(void){
  num_calls += 1;
  return 2 * var;
}
```

Makefile

```
files := main.c source2.c
headers := header_header.h
binary := main

all:
    gcc -o $(binary) $(files) -include $(headers)
clean:
    rm -f $(binary)
```

Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

Pointers: Addresses of Objects

- Memory is a very long array of bytes, each with an address. A pointer is a group of bytes (normally 8) containing the address of some other byte.
- Given an object in memory, the operator reference (&) retrieves its address.
- Given an address, the operator dereference (*) retrieves the object at that address.
- When * is used in a definition, it means “this is a pointer to that type”.
- A pointer is a variable that contains an address to an object.
- The object a pointer “points to”, may be another pointer.

```c
include <stdio.h>

int main(void){
  int var = 99;
  int *pv; //this is a pointer to int

  pv = &var; //store the address of var

  // print the address itself
  printf("pv : %p\n", pv);

  // obtain the object var using
  // a pointer to it
  printf("var: %d\n", *pv);
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./pointer
pv : 0x7ffc1953efcc
var: 99
```

Pointers in Functions

- When you pass a pointer variable to a function, just like with any other variable, you are copying it.
- But if you dereference it inside of the function, you access the original value that the caller has.
- Never return pointers to automatic variables. As the function ends, that memory is reclaimed.
- Instead, you can receive a pointer to something from the caller, or allocate memory from the heap.
- In the later case, note that at some point that allocated memory from the heap must be freed with free(pz).

```c
#include <stdio.h>
#include <stdlib.h>
char *bad_idea(void){
  char c = 'w';
  return &c;
}
void good_idea(char *c){
  *c = *c + 1; //next character
  return;
}

char *also_good_idea(void){
  char *c = malloc(sizeof(char));
  *c = 'z';
  return c;
}

int main(void){
  char *pw, x, *pz;
  x = 'x';
  pw = bad_idea();
  good_idea(&x);
  pz = also_good_idea();
  printf("pw: %p\n", pw);
  printf("w : %c\n", *pw); // DANGER
  printf("x : %c\n", x);
  printf("z : %c\n", *pz);
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ gcc -o pointer_bad pointer_bad.c
pointer_bad.c: In function ‘bad_idea’:
pointer_bad.c:5:16: warning: function returns address of local variable [-Wreturn-local-addr]
    5 |         return &c;
      |                ^~
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./pointer_bad
pw: (nil)
Segmentation fault (core dumped)
```

Pointers in Functions

- When you pass a pointer variable to a function, just like with any other variable, you are copying it.
- But if you dereference it inside of the function, you access the original value that the caller has.
- Never return pointers to automatic variables. As the function ends, that memory is reclaimed.
- Instead, you can receive a pointer to something from the caller, or allocate memory from the heap.
- In the later case, note that at some point that allocated memory from the heap must be freed with free(pz).

```c
#include <stdio.h>
#include <stdlib.h>
void good_idea(char *c){
    *c = *c + 1; //next character
    return;
}

char *also_good_idea(void){
    char *c = malloc(sizeof(char));
    *c = 'z';
    return c;
}

int main(void){
    char x, *pz;
    x = 'x';
    good_idea(&x);
    pz = also_good_idea();
    printf("x : %c\n", x);
    printf("z : %c\n", *pz);
    free(pz);
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./pointer_good
x : y
z : z
```

Pointers, Arrays, and Address Arithmetic

- Arrays and Pointers have a very strong relationship. Any operation that can be achieved with arrays, can be done with pointers.
- L5: pa points to the first element of the array.
- L7: *(pa+1) points to the next element in the array.
- L8: *(pa+n) points to the nth element in the array.
- L10: By definition, the value of an array name alone is the address of the first element of the array.
- Therefore, L5 may be written as in L10.

```
1 #include <stdio.h>
2 int main(void){
3   int arr[] = {101, 102, 103, 104, 105};
4   int *pa, *pa2;
5   pa = &arr[0];
6   printf("*pa : %d\n", *pa);
7   printf("*(pa+1) : %d\n", *(pa+1));
8   printf("%d == %d\n", *(pa+4), arr[4]);
9   printf("%p == %p\n", arr, pa);
10  pa2 = arr;
11  printf("%p == %p\n", arr, pa2);
12
13  printf("%d == %d\n", pa[3], *(arr+3));
14
15  pa += 1;
16  //arr += 1; // error
17
18  char *name = "Claudio"; // plus '\0'
19  printf("name[2] : %c\n", name[2]);
20  printf("Name : %s\n", name);
21 }
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./array
*pa : 101
*(pa+1) : 102
105 == 105
0x7ffd94073330 == 0x7ffd94073330
0x7ffd94073330 == 0x7ffd94073330
104 == 104
name[2] : a
Name : Claudio
```

Pointers, Arrays, and Address Arithmetic

- L13: Additionally, we can use indices with the pointer, or *(+n) with the array name.
- L15-16: There is one key difference: pointers are variables, they can be assigned. An array name is not.
- L18: Strings are just arrays of characters with the null character “\0” at the end. Then, name has 8 elements.
- L20: printf prints the whole array until it finds \0.
- When an array name is passed to a function, internally, it is a pointer variable.

```
1 #include <stdio.h>
2 int main(void){
3   int arr[] = {101, 102, 103, 104, 105};
4   int *pa, *pa2;
5   pa = &arr[0];
6   printf("*pa : %d\n", *pa);
7   printf("*(pa+1) : %d\n", *(pa+1));
8   printf("%d == %d\n", *(pa+4), arr[4]);
9   printf("%p == %p\n", arr, pa);
10  pa2 = arr;
11  printf("%p == %p\n", arr, pa2);
12
13  printf("%d == %d\n", pa[3], *(arr+3));
14
15  pa += 1;
16  //arr += 1; // error
17
18  char *name = "Claudio"; // plus '\0'
19  printf("name[2] : %c\n", name[2]);
20  printf("Name : %s\n", name);
21 }
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./array
*pa : 101
*(pa+1) : 102
105 == 105
0x7ffd94073330 == 0x7ffd94073330
0x7ffd94073330 == 0x7ffd94073330
104 == 104
name[2] : a
Name : Claudio
```

Pointers, Arrays, and Address Arithmetic

- Pointer Comparison: ==, !=
  - Two pointer values are equal if they point to the same location, or if they are both null.
- Assignment: = Same type of pointers.
- Offset: +, - Pointer and Integer
- Distance: - You can subtract two pointers to obtain the distance between them if they are part of the same array.
- null: ==NULL, =NULL Always.
- All other operations are illegal.

```c
#include <stdio.h>
int main(void){
  int arr[] = {101, 102, 103, 104, 105};
  int *pa, *pa2;
  pa = &arr[0];
  printf("*pa : %d\n", *pa);
  printf("*(pa+1) : %d\n", *(pa+1));
  printf("%d == %d\n", *(pa+4), arr[4]);
  printf("%p == %p\n", arr, pa);
  pa2 = arr;
  printf("%p == %p\n", arr, pa2);

  printf("%d == %d\n", pa[3], *(arr+3));

  pa += 1;
  //arr += 1; // error

  char *name = "Claudio"; // plus '\0'
  printf("name[2] : %c\n", name[2]);
  printf("Name : %s\n", name);
}
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./array
*pa : 101
*(pa+1) : 102
105 == 105
0x7ffd94073330 == 0x7ffd94073330
0x7ffd94073330 == 0x7ffd94073330
104 == 104
name[2] : a
Name : Claudio
```

Pointers to Functions

- L2: The second parameter is “a pointer to a function that receives one character”
- L4: Call to the function.
- L14: fun is the name of the function, and acts as a pointer.

```
1 #include <stdio.h>
2 void fn2(char my_char, int (*pfun)(char c)){
3   int next;
4   next = (*pfun)(my_char);
5   printf("Done: %d\n", next);
6 }
7
8 int fun(char c){
9   printf("Char: %c\n", c);
10   return (int) c + 1;
11 }
12
13 int main(void){
14   fn2('K', fun);
15 }
```

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./func_pointers
Char: K
Done: 76
```

Agenda

- Workflow
- Types, Operators and Expressions
- Control Flow
- Functions
- Pointers and Arrays
- Structures

Syntax

- L2: Struct declaration.
- L7,8,28: Elements of struct's addresses accessed with ->
- L6: You can pass a pointer to struct to functions.
- L10: You can return a struct, the whole struct being copied.
- L20: You can assign all members of a struct at definition time.
- L26: You can obtain pointers to structs.

```
shawn@shawn-mini-desktop:~/Workspace/CS143A$ ./structure
p1.x, p1.y: 23, 74
x,y: -1, -2
p1.x, p1.y: -10, -20
pp1->x, pp1->y: -10, -20
```

```
1 #include <stdio.h>
2 struct Point{
3   int x;
4   int y;
5 };
6 void init(struct Point *p){
7   p->x = -1;
8   p->y = -2;
9 }
10 struct Point init2(void){
11   struct Point p;
12   p.x = -10;
13   p.y = -20;
14   return p;
15 }
16 void print_struct(struct Point p){
17   printf("x,y: %d, %d\n", p.x, p.y);
18 }
19 int main(){
20   struct Point p1 = {23, 74}, *pp1;
21   printf("p1.x, p1.y: %d, %d\n", p1.x, p1.y);
22   init(&p1);
23   print_struct(p1);
24   p1 = init2();
25   printf("p1.x, p1.y: %d, %d\n", p1.x, p1.y);
26   pp1 = &p1;
27   printf("pp1->x, pp1->y: %d, %d\n",\
28   pp1->x, pp1->y);
29 }
```

Thank you. Any Questions?