CISC 372: Parallel Computing

C, part 1

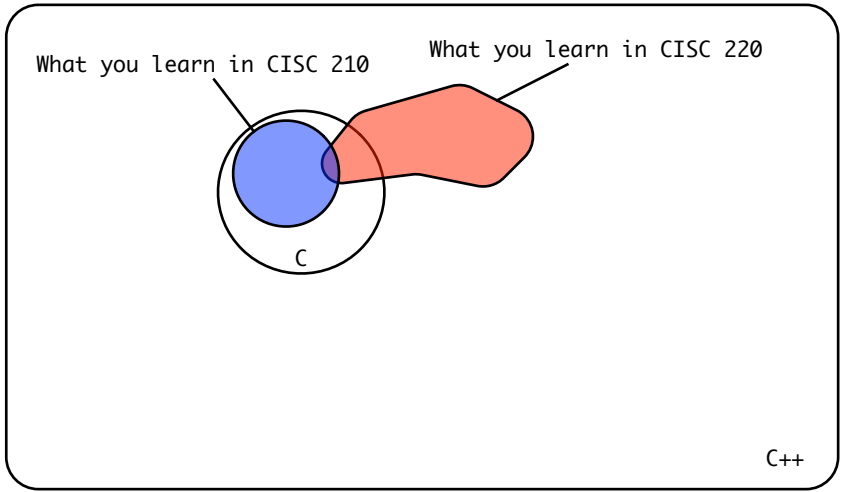
Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

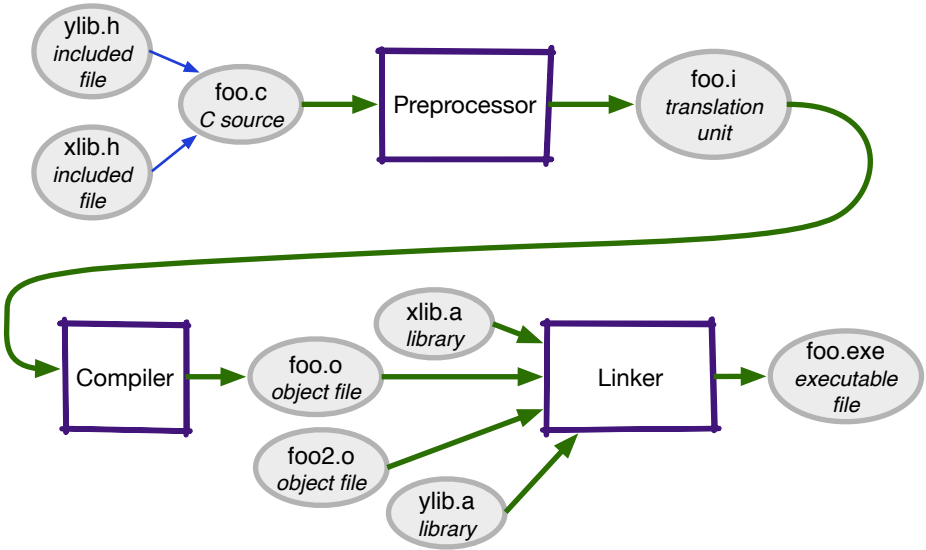
siegel@udel.edu

What is C?

- 1972, Dennis Ritchie, Bell Labs
- a programming language defined by an international standard
  - currently ISO/IEC 9899:2018 : Programming Language — C ("C18" or "C17")
  - spec in docs folder on public svn repo
- characteristics
  - general purpose
  - imperative
  - static types
  - structured programming
  - lexical scopes
  - recursion
  - "low-level" (memory is a sequence of bytes, pointers; "a portable assembly language")
- unlike modern languages, C has unspecified and undefined behavior
  - the standard leaves open many choices to the implementation
  - "unspecified": a finite number of implementation-specific choices
  - "undefined": anything can happen; should always be considered defects


What you learn in CISC 210
What you learn in CISC 220
C
C++

Translation of a C program



Typical command line syntax of a C compiler

- `cc -o foo.exec foo.c`
  - preprocess, compile, and link `foo.c` creating executable `foo.exec`
  - suitable for simple programs consisting of one translation unit
- `cc -E -o foo.i foo.c`
  - preprocess only, sending output to `foo.i`
  - useful for seeing what the preprocessor is doing; debugging
- `cc -c -o foo.o foo.c`
  - preprocess and compile only, creating object file `foo.o`
- `cc -o foo.exec foo1.o foo2.o foo3.o`
  - link object files `foo1.o`, `foo2.o`, `foo3.o` and libraries to form executable `foo.exec`

Preprocessor directives

- `#include "filename"` or `#include <filename>`
  - insert contents of filename here
- `#define X some text`
  - let X = "some text"
- `#define X`
  - let X be the empty string — but still defined
- `#ifdef X` — if X is defined, include ...

```
...
#endif
```

- `#ifdef X` — if X is defined, include ... else include ...

```
...
#else
...
#endif
```

- `#if defined(X) && Y>2`

Preprocessor example motivation: constants

The length in a C array declaration in file scope must be a constant expression ...

- `int a[100];` : good
- `const int n=100;` : may or may not work
  `int a[n];`
- `100` is definitely a constant expression
- is `n` a constant expression? that's up to the C implementation
- your goal should be to write portable code
  - will work for any conforming C compiler
  - compile with `-pedantic` to see if you rely on any non-portable features

```
basie:tmp siegel$ cc -pedantic -std=c11 tmp.c
tmp.c:2:5: warning: size of static array must be an integer constant
           expression [-Wpedantic]
int a[n];
      ^
1 warning generated.
```

Preprocessor example usage: constants

This always works:

```
#define N 100
int a[N];
int main() {
  for (int i=0; i<N; i++)
    a[i] = i;
}
```

After preprocessing, the code above becomes

```
int a[100];
int main() {
  for (int i=0; i<100; i++)
    a[i] = i;
}
```

Defining preprocessor macros on the command line

Compiling with the flag...

- `-DX`
  - equivalent to inserting `#define X` at the beginning of the file
- `-DX=blah`
  - equivalent to inserting `#define X blah` at the beginning of the file

Example:

```
// the preprocessor macro N must be defined (length of array a)
int a[N];
int main() {
  for (int i=0; i<N; i++)
    a[i] = i;
}
```

Compile:

```
cc -pedantic -DN=100 tmp.c
```

Example: boolean flags controlling compilation

```
// the preprocessor macro N must be defined (length of array a)
// define DEBUG to see debugging output
#include <stdio.h>
int a[N];
int main() {
#ifdef DEBUG
  printf("Entering for loop with N=%d\n", N);
  fflush(stdout);
#endif
  for (int i=0; i<N; i++)
    a[i] = i;
#ifdef DEBUG
  printf("Exiting for loop.\n");
  fflush(stdout);
#endif
}
```

To compile a "debugging version" of this program:

```
cc -pedantic -DDEBUG -DN=100 tmp.c
```

Preprocessor: Function-like macros

- the macros above are called object-like macros
- you can also `#define` function-like macros

```
#define MAX(x,y) ((x)>=(y) ? (x) : (y))
int main() {
  int m = MAX(N, 10);
}
```

expands to

```
int main() {
  int m = ((N)>=(10) ? (N) : (10));
}
```

- why the abundance of parentheses?

Function-like macros: beware the pitfalls!

```
#define ADD(x,y) x+y
#define MUL(x,y) x*y
```

What do these expand to?

- `ADD(1,2)*3` $\quad 1+2*3 = 7$
- `MUL(2,3+4)` $\quad 2*3+4 = 10$

Better:

```
#define ADD(x,y) ((x)+(y))
#define MUL(x,y) ((x)*(y))
```

Structure of a C program

After preprocessing, the program consists of a sequence of

- declarations
  - variables
    - `int x;`
  - types
    - `typedef double D;`
  - enumerations
    - `enum Color { RED=0, GREEN=1, BLUE=2 };`
  - function prototypes
    - `int sgn(double x);`
- function definitions
  - ```
    int sgn(double x) {
      if (x>0)
        return 1;
      else if (x<0)
        return -1;
      else return 0;
    }
    ```
    ```

Types

- C is statically typed
- every variable and expression has a type that is known at compile time
  - "statically"
  - before running the program
- you should be able to read a program and identify the type of any expression
- a type can be complete or incomplete
- every complete type has a size
  - the number of bytes required to store one element of that type
- `sizeof(`T`)` is an expression that returns the size of type T
  - this is a positive integer
- examples
  - `sizeof(int)`
    - often 4, sometimes 8
    - must be big enough so that `int` can hold at least -32767 .. 32767
  - `sizeof(float[10])`
    - size of an array of 10 `float`s
    - equals $10 *$ `sizeof(float)`

`char`: the smallest type

- `char` : size is always one byte
  - a byte has at least 8 bits
  - the smallest addressable unit of memory
- `signed char` : includes at least $-128..127$
  - `signed` means the type includes positive and negative integers (and 0)
- `unsigned char` : includes at least $0..255$
  - `unsigned` means the type includes only nonnegative integers
- `char` is either `signed char` or `unsigned char`
  - which one is unspecified

Other integer types

- (`short` | | `long`) (`signed` | `unsigned`) `int`
  - $3 * 2 = 6$ combinations, each a different type
  - `short signed int`, `short unsigned int`, `signed int`, ...
- abbreviations: `int` is optional, `signed` is the default
  - `short` = `short int` = `short signed int`
  - `long` = `long int` = `long signed int`
  - `int` = `signed int`
  - `unsigned short` = `unsigned short int`
  - `unsigned` = `unsigned int`
  - `unsigned long` = `unsigned long int`
- sizes
  - the C Standard specifies minimum ranges for each of these types
  - also `short` $\leq$ "medium" $\leq$ `long`
- `_Bool`
  - consists of exactly 0 and 1
  - a subtype of `int`

Floating types

- `float`
  - floating point
  - typically, 4 bytes = 32 bits
- `double`
  - double precision floating point
  - at least as precise as `float`
  - typically, 8 bytes = 64 bits

Simple declarations

For these basic types

- syntax: type-name variable-name ;
- can declare multiple variables of the same type
- an initializer is optional

Examples:

- `int x;`
- `double y;`
- `unsigned long z;`
- `int x, y;`
- `int x = 3;`
- `int x=3, y, z=-17;`

Array types: declaration

Declaration

- if $T(x)$ declares x to have type T
- then $T($`a[]`$)$ declares `a` to have type *array-of-T*
- and $T($`a[n]`$)$ declares `a` to have type *array-of-length-n-of-T*

Declaration examples

- `double a[]`
  - $T(x) =$ `double` x
    - declares x to have type `double`
  - $T($`a[]`$) =$ `double a[]`
    - declares `a` to have type *array-of-double*
    - incomplete array type
- `double a[n]`
  - $T(x) =$ `double` x
  - $T($`a[n]`$) =$ `double a[n]`
    - declares `a` to have type *array-of-length-n-of-double*
    - complete array type

Complete and incomplete types

- in some places, a complete array type is required
  - whenever space must be allocated for the array
  - an ordinary (not parameter) declaration of a local or global variable of array type
- in other places a complete or incomplete array type may be used
  - a parameter declaration of array type
  - as the base type in a pointer type
- the element type of an array type must be a complete type
  - in multi-dimensional arrays, only the first length can be unspecified
  - `int a[][10][20];`
    - incomplete type, but complete element type
    - OK sometimes (e.g., function parameter)
  - `int a[10][];`
    - incomplete element type – bad

Array example: simple 2d-array

```
#define N 2
#define M 3
int a[N][M];
int main() {
  // initialize...
  for (int i=0; i<N; i++)
    for (int j=0; j<M; j++)
      a[i][j] = i*M+j;
  // print...
  for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++)
      printf("%d ", a[i][j]);
    printf("\n");
  }
}
```

Compile and execute:

```
basie:tmp siegel$ cc tmp.c
basie:tmp siegel$ ./a.out
0 1 2
3 4 5
basie:tmp
```

Exercise: transpose a square matrix

```
#define N 3
int a[N][N];
int main() {
  // initialize...
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      a[i][j] = i*N+j;

  // INSERT: in-place transpose

  // print...
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++)
      printf("%d ", a[i][j]);
    printf("\n");
  }
}
```

```
basie:tmp siegel$ ./a.out
0 3 6
1 4 7
2 5 8
```

Pointers

- a pointer is the address of a memory location
- pointers are first-class objects in C
- there are pointer types
- a pointer can be assigned using =
- a pointer can be passed as an argument in a function call
- a pointer can be returned by a function
- there are operations which consume pointers and return pointers
- a pointer is just like any other kind of data

Pointer types

- declaration
  - if $T(x)$ declares x to have type T
  - then $T(*\texttt{p})$ declares `p` to have type *pointer-to-T*
- declaration examples
  - `double *p`
    - $T(x) =$ `double` x
    - $T(*\texttt{p}) =$ `double *p`
    - `p` has type *pointer-to-double*
  - `unsigned long int *p`
    - $T(x) =$ `unsigned long int` x
    - $T(*\texttt{p}) =$ `unsigned long int *p`
    - `p` has type *pointer-to-unsigned-long-int*

Pointer operations

There are two basic operations on pointers:

- address-of (`&`)
  - given a variable, returns the address of that variable
  - if `x` has type T then `&x` has type pointer-to-T
  - example
    - ```
      int x;
      int *p = &x; // address of x
      ```
- dereference (`*`)
  - given a pointer, returns the value stored at that address
  - if `p` has type pointer-to-T then `*p` has type T
  - example
    - ```
      int x = 5;
      int *p = &x;
      int y = 2 * (*p); // 10
      ```

Pointer operations, cont

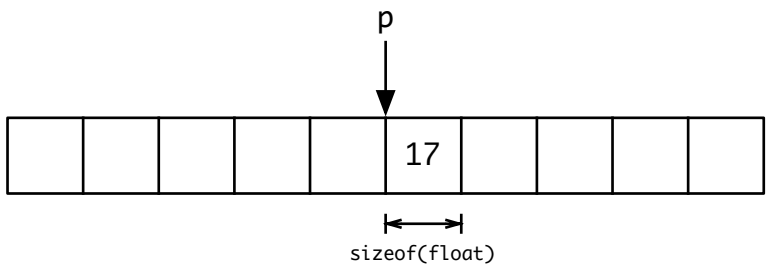
- `*p` can also be used on the left-hand side of an assignment

```
double x = 3.1415;
double *p = &x;
*p = 2.71828;
printf("%lf", x); // 2.71828
```

Pointers into arrays

- you can also take the address of array elements

```
float a[10];
float *p = &a[5];
*p = 17;
```



Pointer into 2d-array

```
int a[2][3];
int *p = &a[0][2];
*p = 13;
```