CS 3650 Computer Systems – Spring 2026

Introduction to C

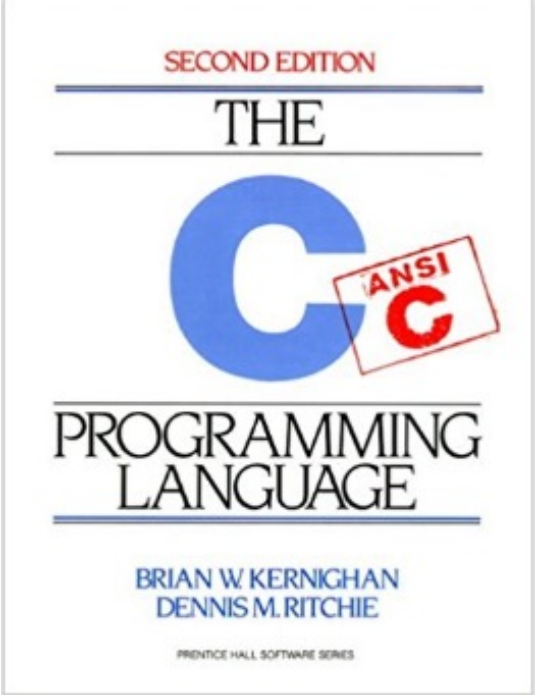
Week 4

Northeastern University

* Acknowledgements: created based on Mike Shah's and Ferdinand Vesely's lecture notes.

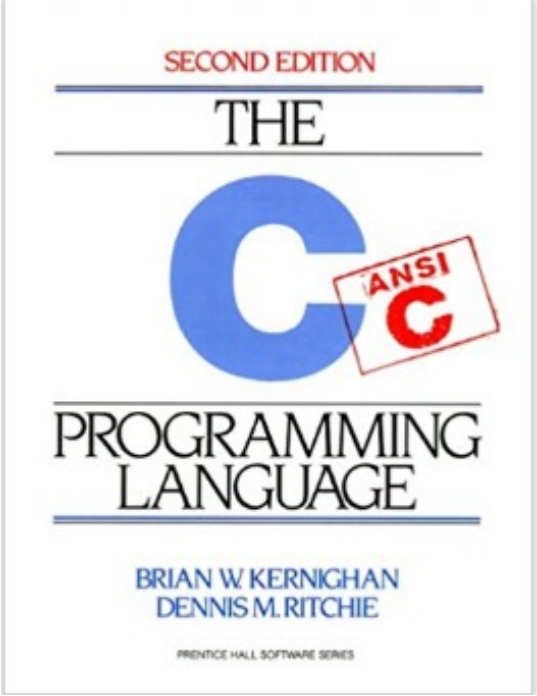
Background

- Programming language developed by Dennis Ritchie in 1972
- A successor language of Bell lab's programming language "B"
- C was intended to make programming Unix easier
- Early Unix versions in Assembly
- High-level, compared to assembly
- But still low-level conceptual model



Background

- Types - kind of “strong” but not really
- You manage memory
- You can even inline assembly



hello world in C

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("hello world!\n");
    return 1;}
// prints hello world
```

- **Compilation**: gcc hello.c -o hello
- #include <stdio.h>
  - imports the library for printf
- **Command line arguments**
  - **int argc**: number of arguments (> 1)
  - **char *argv[]:** array of string

  - **./hello argument test 1**
    - argc= 4,
    - argv[0] = "./hello" (includes the path to the binary and the file name)
    - argv[1] = "argument"
    - argv[2] = "test"
    - argv[3] = "1"

Other familiar features

- Blocks of scope are delimited by { and }
  - Variables are declared at the top of the block before calling other statements
  - Variable declared in the block is only visible in that block and any sub-blocks
  - Once the block ends, variable is not visible anymore
  - Blocks can be nested

- ; is used at the end of a statements

- Functions are declared pretty much like Java methods:
  - return_type function_name(type1 arg1, type2 arg2, ...)
    - E.g.,: int max(int first, int second)
  - Functions that don't return anything have a return type void
    - E.g.,: void print_many_ints(int first, int second, int third)

Sizes of data types (C compared to assembly)

When in doubt about the size: use sizeof(type)

C Declaration	Intel Data Type	Assembly-code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double Precision	l	8

Note: no Boolean type. 0 or 1

Control flows: if

```
if (condition) {
    // do stuff
}
```

```
if (condition) {
  // do stuff
} else {
  // do other stuff
}
```

Control flows: while

```
while (condition) {
  // do this while condition holds
}

do {
  // do this at least once and then
  // keep doing it again while condition holds
} while (condition); // don't forget the semicolon
```

Control flows: for

```
// 1. run the initializer expression
// 2. if condition holds go to 3, else go to 6
// 3. do stuff in body
// 4. run the updater expression
// 5. Go to 2
// 6. End

for (initializer; condition; updater) {

}
```

Operators

- Comparison operators: <, >, <=, >=, ==, !=
  - while (a <= b)
  - while (a != b)
  - for (i = 0; i < 10; i++)

- Logical operators: !, &&, ||
  - if(x > 0 && x <10)
  - while(x > 0 || y > 0)

Continue and break

- You can skip the rest of the current iteration of the innermost loop with continue
- You can break out of the innermost loop with break

```
while (x > 0) {
   if (x > 100) {
         break;
   }
   if (x > 10) {
         // do something 1
         continue;
   }
   // do something 2
}
```

Control flows: switch

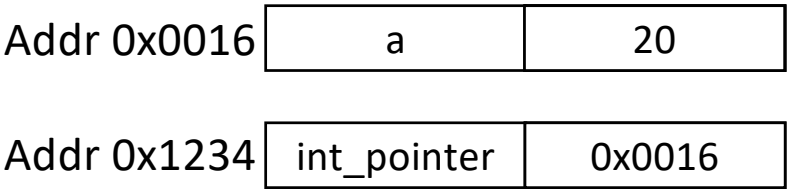
- Condition checks based on matching an expression

```
switch(expression) {
    case constant-expression:
            // do something
            break; // optional: if you don't break the next
                   // block will be executed unconditionally
    case constant-expression:
            // do something
            break;
    …
    default:
            // do something
}
```

Pointers

- DataType * pointer
  - int *int_pointer; // pointer to an int
  - double *element = NULL; // good practice to make initialize to NULL
- A pointer stores a memory address of a data instance

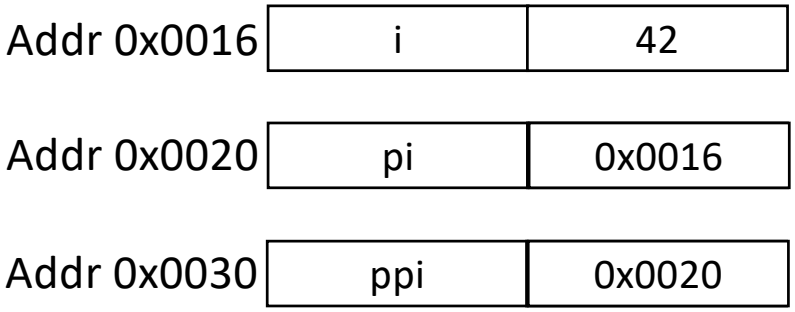
```
int main()
{
    int a = 10;
    int * int_pointer;
// currently points to an arbitrary location
    int_pointer = &a;
// & returns the address of the variable
    printf("%p\n", int_pointer);
// *pointer accesses the value
// stored in the memory address
    printf("%d\n", *int_pointer);
    *int_pointer = 20;
    printf("%p\n", int_pointer);
    printf("%d\n", *int_pointer);
    printf("%d\n", a);
    return 0;
}
```



Sample Output:
0x0016
10
0x0016
20
20

Pointer of pointer

```
int i = 42;
int *pi = &i;
int **ppi = &pi;
printf("%d %d %d\n", i, *pi, **ppi);
```



What should be printed?

42 42 42

ppi = pointer to (address of) pi
*ppi = pointer to (address of) i
**ppi = value of i

Reason why pointers are considered difficult

- Some program languages do not expose memory addresses
- Accessing an arbitrary address through pointers causes runtime errors
  - When you pass around pointer variables you will often see this
- Memory address is not a value that you directly use in a program
  - But it is often more convenient to have access to
- Little control over memory addresses (program assigns for you)
  - You will only directly assign NULL or copy an existing address e.g., a declared variable
  - Sometimes you will access a RELATIVE address

Arrays

- Arrays are just pointers with some fancy syntax
- There are static (size known at compile-time) and dynamic arrays

- First static arrays

```
float nums[4]; // create an array of 4 floats
```

- These will be stored contiguously in memory
- **nums points to the first element, nums[0]**

Arrays

- We can access them individually using indices, starting from 0

```
float nums[4]; // create an array of 4
nums[0] = 0.1;
nums[1] = 3.14;
nums[2] = 1.5;
nums[3] = 3214;
printf("2nd element: %f\n", nums[1]);
```

- Arrays can also be initialized:

```
float nums[4] = { 0.1, 3.14, 1.5, 3214 };
printf("2nd element: %f\n", nums[1]);
```

Arrays

- Pointer-based access

```
float nums[4] = { 0.1, 3.14, 1.5, 3214 };
printf("2nd element: %f\n", nums[1]);

printf("1st element: %f\n", *nums);
printf("2nd element: %f\n", *(nums+1));
printf("3rd element: %f\n", *(nums+2));
printf("4th element: %f\n", *(nums+3));
```

String

- In C (like in Assembly for us), strings are just arrays of characters, terminated by a 0 byte (also written '\0')

- Relevant functions are defined in **<string.h>**

- A string literal "Hello, world!" is just the corresponding array of characters with an extra char for \0

```
// msg1 and msg2 define identical objects in memory
char msg1[6] = "Hello";

char msg2[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Dynamic memory allocations

- Memory can be allocated using the library function malloc()
  - It is defined in <**stdlib.h**>
  - Its single argument is the number of bytes desired
  - Returns a pointer to the block of memory (if successful)
  - Allocated memory needs to be freed using free()

```
int *one_int = malloc(4);
*one_int = 42;
free(one_int);
```

Dynamic memory allocations

- We will mostly use malloc to allocate arrays and structs (below)

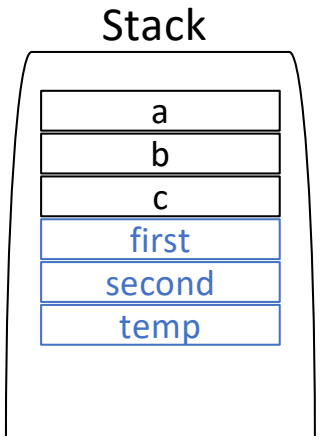
```
int *fifty_ints = malloc(50 * sizeof(int));
for (int i = 0; i < 50; ++i) {
        fifty_ints[i] = i * i;
   }
free(fifty_ints);
```

Pointers and memory management

- Stack vs heap
  - Stack memory is automatically managed (maintains variables in the scope)

```
int addsquare(int first, int second) {
        int temp = first + second;
        return (temp * temp);
}

int main() {
        int a = 1;
        int b = 2;
        int c = addsquare(a, b);
        printf("%d\n", c);
        return 1;
}
```

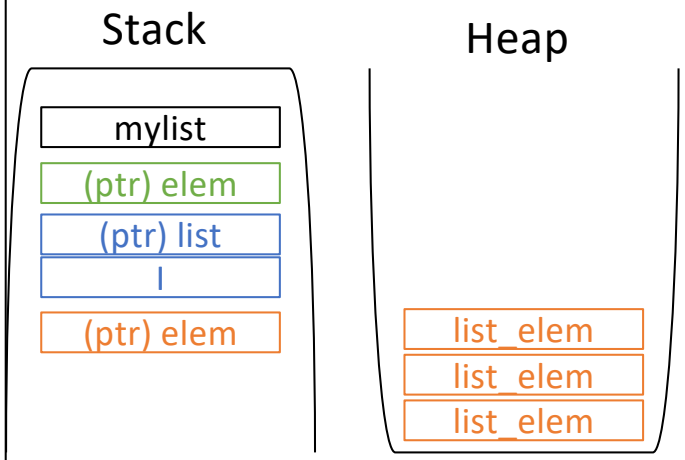


Pointers and memory management

- Stack vs heap
  - Heap memory is dynamically allocated and you must manage it

```
void add_elements(struct list *list) {
 int I;
 for (I = 0; I < 3; I ++) {
    struct list_elem *elem = malloc(sizeof(
           struct list_elem));
    list_push_back(list, elem);
    }
}
int main() {
    struct list my_list;
    list_init(&my_list);
    add_elements(&my_list);
    while (list_size(&my_list) > 0) {
        struct_list_elem *elem = list_pop_front(&my_list);
        free(elem);
           }
    return 1;
}
```

If you forget to delete, memory space will be wasted and in the long run, you can run out of memory space (memory leak)



Structs

- Structs are the most useful user-defined data types in C
- Think of them as Java classes, but everything is public
  - Structs do not have methods
- A struct stores multiple values of different types together
- It is defined using the **struct** keyword

```
struct address {
  unsigned int house_no;
  char street[32];
  char city[24];
  char state[3];
  unsigned int zip;
}; // don't forget the semicolon.

struct address home, work;  // this allocates two
                            // structs on the stack
```

Structs

- To access a field, we use "."

```
work.house_no = 360;
strcpy(work.street, "Huntington Ave"); // see man 3 strcpy
strcpy(work.city, "Boston");
strcpy(work.state, "MA");
work.zip = 02115;
```

- Structs can, of course, be nested:

```
struct person {
            char first[32];
            char last[32];
            struct address home;

};
```

They can be passed to and returned from a function:

```
struct address get_address(struct person p) { ... }
```

Typedef

Writing out struct every time can be tiring

```
struct address my_home;
struct person myself;
struct address get_home_addr(struct person arg);
```

- C allows us to introduce type synonyms using typedef:

```
typedef struct person person_t; // now we can use person_t
                                // to mean struct person
```

- typedef can be used with any type to make code more readable:

```
typedef unsigned short age_t;
```

Pointers to structs

- Of course, we can have pointers to structs:

```
struct person *p; // OR person_t *p;
```

- We can use the address-of operator & to get the address of a struct:

```
struct address *current = &work;
```

- We can also allocate memory for structs dynamically, using malloc and sizeof:

```
struct person *ferd = malloc(sizeof(struct person));

person_t *ferd = malloc(sizeof(person_t));
```

- We can also create arrays of structs:

```
person_t class[80];
person_t *friends = malloc(5 * sizeof(person_t));
// ...
for (int i = 0; i < 5; ++i) {
  if (strcmp(friends[i].home.street, "Huntington Ave") == 0) {
    printf("%s lives close!\n", friends[i].first);
  }

}
```

Pointers to structs

- Often, pointers are used to pass a struct to a function
  - This avoids copying the contents into the function's stack frame

- When accessing fields via a pointer, we use **->**

```
int lives_in_boston(person_t *p) {
  return strcmp(p->home.city, "Boston") != 0;
  // equivalent to
  // return strcmp((*p).home.city, "Boston") != 0;
}
```

Preprocessor

- The C preprocessor (CPP) is a separate phase run at the very beginning of the compilation process

- It is basically just a text processing engine that modifies the source text based on preprocessor directives

- The main job of CPP is to:
  - **Include** the requested header files
  - **Define "global constants"** – IMPORTANT: these are just textual macros, that is, pieces of C code that will get spliced wherever the constant name is mentioned
  - **Choose** which parts of code to include for compilation based on various conditions

Preprocessor: **#define**

- This directive is used to define a textual macro
- The macro can be a constant macro or a parametrized macro

```
#define COUNT 100
#define COURSE "Computer Systems"
```

  - This will define the macros **COUNT** and **COURSE**;
  - Everywhere else where **COUNT** is mentioned, it will be replaced with 100, and **COURSE** will be replaced with "Computer Systems"

Preprocessor: **#define**

- Note, that the expression is simply substituted for the macro
- It does not get evaluated at the definition site
- Hence there is a subtlety that one has to keep in mind: Consider,

```
#define X  10 + 2
int a = X;      // expands to 10 + 2
int b = 3 * X; // expands to 3 * 10 + 2
               //this might not be what we expect
```

- The solution is to always put an expression in parentheses:

```
#define X (10 + 2)
int b = 3 * X; // expands to 3 * (10 + 2)
```

Preprocessor: **#define**

- Parametric ("function-like") Macros
  - We can also define macros with arguments using #define
  - These look like function calls, but they are expanded at compile-time
  - Example,

```
#define max(a, b)    (a > b ? a : b)
printf("%d\n", max(3, 4));
```

- The argument to a macro does not get evaluated before being used in the macro, so we have a similar problem as above:

```
#define dbl(x)  (2 * x)
printf("%d\n", dbl(10 + 1));  // expands to 2 * 10 + 1,
                              // thus prints 21, not 22!
```

- So any argument use in a macro body should be enclosed in ():

```
#define max(a, b) ((a) > (b) ? (a) : (b))
#define dbl(x) (2 * (x))
```

Preprocessor: **#define**

- Another caveat: consider the following:

```
#define foomacro(x) ((x) + (x))
int foofun(int x) { return x + x; }
```

- Although both seem to be computing the same result, they will behave differently if the expression passed in has side-effects:

```
int x = 10;
printf("%d\n", foomacro(++x)); // will likely print 23
x = 10;
printf("%d\n", foofun(++x)); // prints 22
```

- Why?

- Note: a good modern C compiler will usually warn you about this

Preprocessor: **#include**

- The #include directive performs a textual inclusion of the given file

- Generally, only ever use this for headers - .h files

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

  - **DO NOT INCLUDE C FILES**

- Headers contain
  - Declarations and definitions of functions
  - Macros
  - Sometimes also global variables

Preprocessor: **#if/#ifdef/#ifndef/#elif/#else**

- These are compile-time conditionals that hide or expose parts of the source file from or to the compiler
- Example:

```
#ifdef UNIX
       PATH_SEPARATOR "/"
#elif
       defined WINDOWS PATH_SEPARATOR "\\"
#endif
```

- Other example:

```
for (int i = 0; i < length; i++) {
   sum += array[i];
   #if DEBUG_LEVEL >= 1
   printf("array[%d] = %d, sum = %d\n", i, array[i], sum);
   #endif

}
```

Header files

- Commonly include
  - Function declarations

    int max(int a, int b);

    int min(int a, int b);
  - Structs
  - Macros

mycode.h

```
#ifndef __MYCODE_H__
#define __MYCODE_H__
struct my_struct {
        int x;
        int y;
};
int my_function(
       struct my_struct *my_arg);

#endif
```

mycode.c

```
#include “mycode.h”

int my_function(
        struct my_struct *my_arg)
{
        int z;
        // do something
        return z;
}
```

Separate Compilation

- my_max.h

```
int my_max(int a, int b);
```

- my_max.c

```
int my_max(int a, int b) { return ((a > b) ? a : b); }
```

- my_min.h

```
int my_min(int a, int b);
```

- my_min.c

```
int my_min(int a, int b) { return ((a < b) ? a : b); }
```

- main.c

```
#include “my_max.h”
#include “my_min.h”
int main(void) {
        int x = 1;
        int y = 2;
        int z = 3;
        my_min(x, y);
        my_max(y, z);
        return 0;
}
```

Double quote to include custom header files

```
gcc –c my_max.c –o my_max.o
gcc –c my_min.c –o my_min.o
gcc –c main.c –o main.o
gcc my_max.o my_min.o main.o -o my_prog
```

```
gcc –c my_max.c my_min.c main.c
gcc my_max.o my_min.o main.o –o my_prog
```

```
gcc my_max.c my_min.c main.c –o my_prog
```

Using functions and variables from different files

- my_max.c

  int my_max(int a, int b) { return ((a > b) ? a : b); }

- my_min.c

  int my_min(int a, int b) { return ((a < b) ? a : b); }

- main.c

```
extern int my_max(int a, int b);
extern int my_min(int a, int b);
int main(void) {
   int x = 1;
   int y = 2;
   int z = 3;
   my_min(x, y);
   my_max(y, z);
   return 0;
}
```

```
gcc –c my_max.c –o my_max.o
gcc –c my_min.c –o my_min.o
gcc –c main.c –o main.o
gcc my_max.o my_min.o main.o -o my_prog
```

```
gcc –c my_max.c my_min.c main.c
gcc my_max.o my_min.o main.o –o my_prog
```

```
gcc my_max.c my_min.c main.c –o my_prog
```

Global variables

- Global variables can be declared outside of functions
- They can be accessed by anywhere in the program
- Pros
  - Convenient because all functions can access
- Cons
  - Can accidentally change
  - Abusing global variables can easily introduce bugs

main.c

```
int global_var = 100;
void print_global_var() {
    printf("%d\n", global_var);
}
int main(void) {
// do something
    return 0;
}
```

inc_dec.c

```
extern int global_var;
void inc_global_var() { global_var++; }
void dec_global_var() { global_var--; }
```