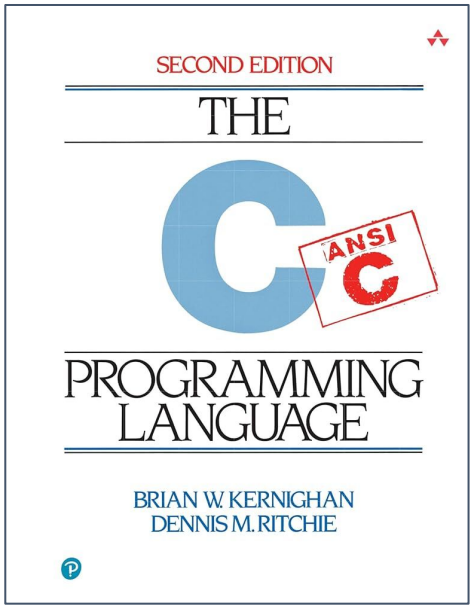
NEU CS 3650 Computer Systems

Instructor: Dr. Ziming Zhao

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Alden Jackson, Ben Weintraub, Gene Cooperman, Peter Desnoyers' lecture slides for the same course.

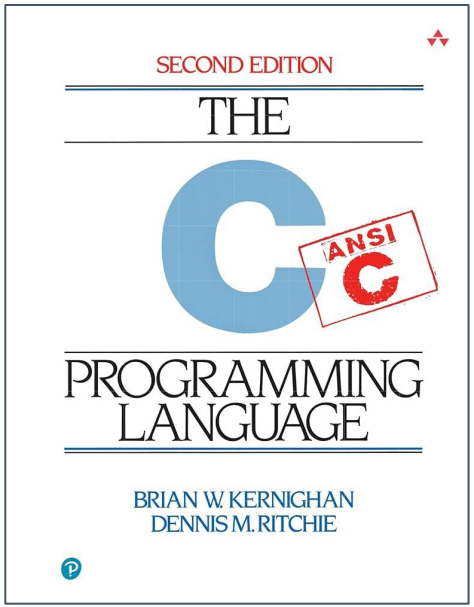
C Background

- 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



C Background

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



C hello world

```
#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
- Getting command line arguments
  - int argc: number of arguments (> 1)
  - char * argv[]: array of string
  - ./hello argument test 1
    - argc= 4,
    - argv[0] = “./executable” (always the path to binary file name)
    - argv[1] = “argument”
    - argv[2] = “test”
    - argv[3] = “1”
- printf
  - “Print”s according to the “f”ormat string
  - “\n” adds new line at the end of the string

The full main() signature

int main(int argc, char *argv[], char *envp[])

Parameters explained

- **argc** → *argument count*, the number of command-line arguments.
- **argv** → *argument vector*, an array of strings (char *) containing the arguments.
  - argv[0] is the program name.
  - argv[1] ... argv[argc-1] are the actual arguments.
- **envp** → *environment pointer*, an array of strings representing the environment variables.
  - Each string is of the form "KEY=VALUE".
  - The array is terminated by a NULL pointer.

Start a User Process

_start ###part of the program; entry point
→ calls __libc_start_main() ###libc
→ calls main() ###part of the program

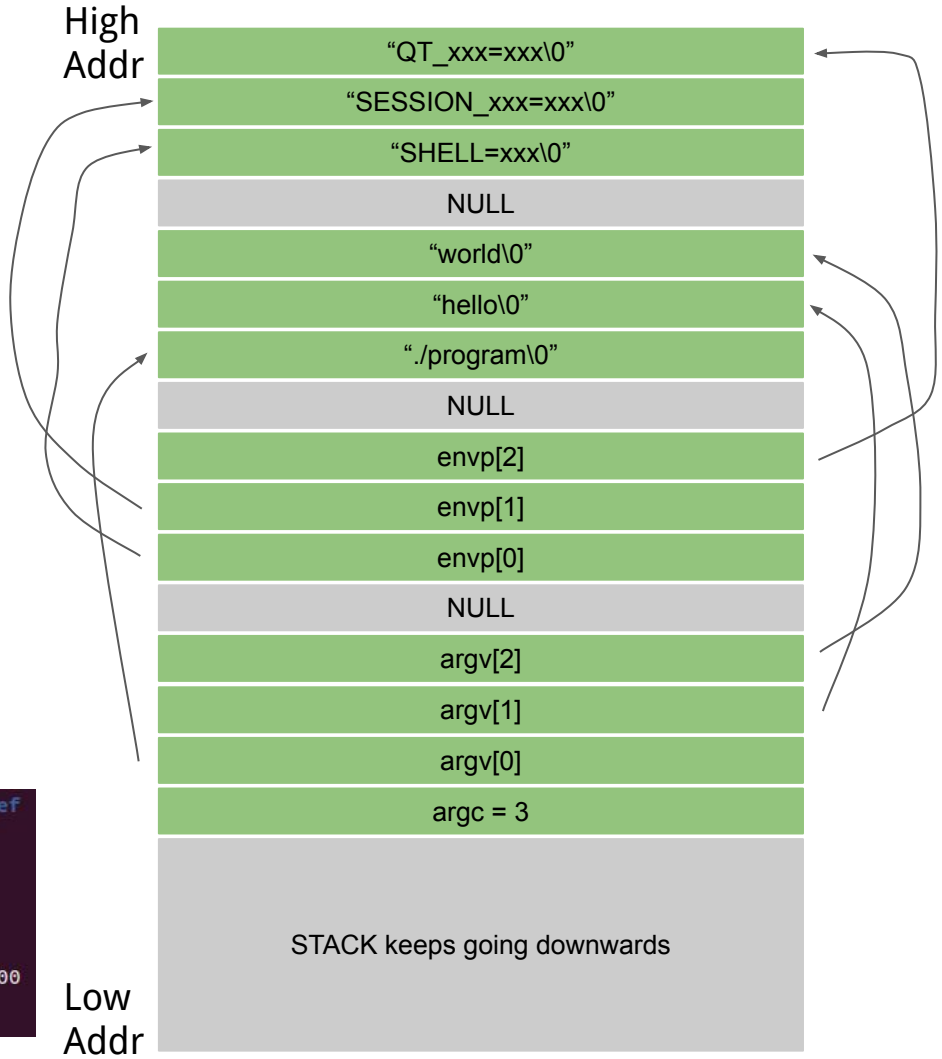
https://www.bottomupcs.com/starting_a_process.xhtml

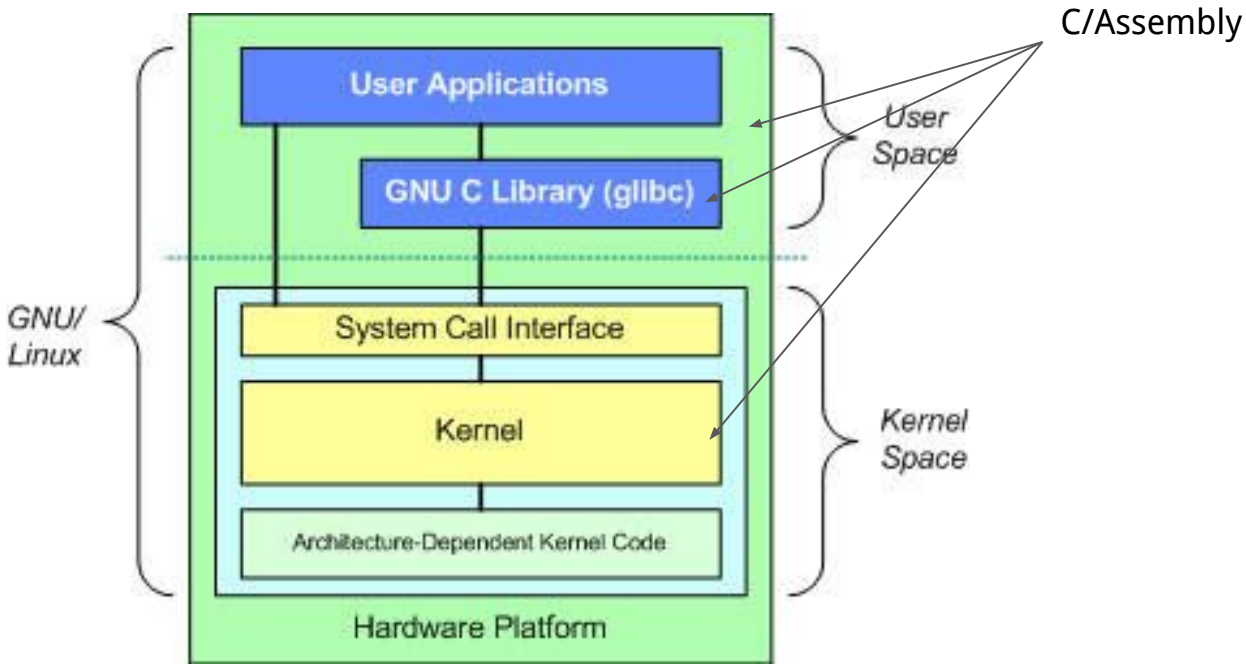
The Stack Layout before main()

The stack starts out storing (among some other things) the environment variables and the program arguments.

```
$ env
SHELL=/bin/bash
SESSION_MANAGER=local/ziming-XPS
QT_ACCESSIBILITY=1

$ ./stacklayout hello world
hello world
```

```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Def
ense for Binaries UB 2020/code/stacklayout$ ./stacklayout hello world
argc is at 0xffc444d0; its value is 3
argv[0] is at 0xffc462d0; its value is ./stacklayout
argv[1] is at 0xffc462de; its value is hello
argv[2] is at 0xffc462e4; its value is world
envp[0] is at 0xffc462ea; its value is SHELL=/bin/bash
envp[1] is at 0xffc462fa; its value is SESSION_MANAGER=local/ziming-XPS-13-9300
:@/tmp/.ICE-unix/2324,unix/ziming-XPS-13-9300:/tmp/.ICE-unix/2324
envp[2] is at 0xffc46364; its value is QT_ACCESSIBILITY=1
```




C/Assembly
User Applications
GNU C Library (glibc)
User Space
GNU/ Linux
System Call Interface
Kernel
Kernel Space
Architecture-Dependent Kernel Code
Hardware Platform

The kernel does not need to have a main()

In xv6, the kernel “main” is actually called **main** but lives in **main.c** inside the kernel source. Prototype: void main(void)

The Linux kernel does **not** have a user-level main() function — but it does have an equivalent **entry point** in C called start_kernel()

https://github.com/mit-pdos/xv6-public/blob/master/main.c

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)

Build-in/Primitive Data types

- Integer types
  - short: 16 bit integer
  - int: 32 bit integer
  - long int: 64 bit integer
  - char: 8 bit character (‘a’, ‘b’, ‘c’, ‘.’, ‘#’)
  - Each of the above (except plain char) can be modified with:
    - signed (default, can be positive or negative).
    - unsigned (only non-negative, larger positive range).
- Floating-Point Types
  - float: 32 bit floating point numbers
  - double: 64 bit floating point numbers (3.14, -123.456)
  - long double – extended precision (implementation-defined, often 80 or 128 bits).
- Void type
  - Used for functions that don’t return anything: void foo() { … }
  - Also used for generic pointers: void *p;

- No Boolean types: integer with 0 or 1 is used instead
- When in doubt about the size you can print sizeof(type)

Types - “Kind of strong, but not really”

```
#include <stdio.h>

int main() {
    double d = 3.14;
    int i = d;        // implicit conversion (truncates to 3)

    char c = 200;     // overflow, since char is usually -128 to 127

    int *p = (int *)0x1234; // arbitrary cast: allowed, but dangerous!

    printf("d = %f, i = %d, c = %d\n", d, i, c);
    return 0;
}
```

Derived/Special Types

- Pointers (int *p;) – memory addresses of values.
- Arrays (int a[10];) – contiguous sequences of elements.
- Structures (struct) and Unions (union) – user-defined groupings of data.
- Enumerations (enum) – symbolic constants mapped to integers.

Control flows: if

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

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

Control flows: if

```
int x = 5;

if (x > 10) {
    printf("x is greater than 10\n");
} else if (x > 0) {
    printf("x is positive but not greater than 10\n");
} else {
    printf("x is not positive\n");
}
```

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) {

}
```

Control flows: for

```
#include <stdio.h>

int main() {
    // Print numbers 1 through 5
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Operators

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

In classic C (before C99), there is **no dedicated bool type**. Any comparison (==, !=, <, >, <=, >=) produces an **int value**. The result is: 1 if the comparison is **true.** 0 if the comparison is **false.**

In classic C, 0 is false. Any nonzero value (positive or negative) is true.

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

Operators

```
#include <stdio.h>

int main() {
    int x = -5;

    if (x) {
        printf("x is true\n");
    } else {
        printf("x is false\n");
    }

    if (!x) {
        printf("!x is true\n");
    } else {
        printf("!x is false\n");
    }

    return 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
}
```

Continue and break

```
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            break;  // exits the loop completely
        }
        printf("%d\n", i);
    }

    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;  // skips this iteration and goes to next
        }
        printf("%d\n", i);
    }

    return 0;}
```

Control flows: switch

- Condition checks based on matching an expression (usually just a variable)

```
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
}
```

In **C**, the expression after case must evaluate to an **integer constant expression**.

Control flows: switch

```
int main() {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("Invalid day\n");
            break;
    }
```

Control flows: switch

```
switch (ch) {
    case 'a':   // allowed (character constant is an int)
        printf("You pressed a\n");
        break;
    case 10:    // allowed (integer constant)
        printf("You pressed newline\n");
        break;
}
```

Control flows: switch

```
enum Color { RED = 1, GREEN = 2, BLUE = 3 };

switch (color) {
    case RED:
        printf("Red\n");
        break;
    case GREEN:
        printf("Green\n");
        break;
}
```

Control flows: switch

Floating-point constants (case 3.14:) → compile error

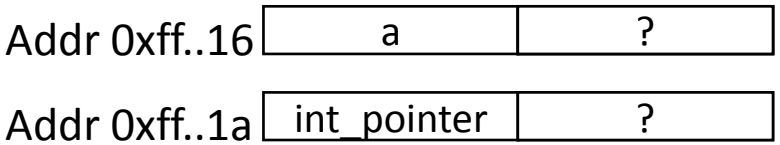
Strings (case "hello":) → compile error

Variables (int x = 5; case x:) → not allowed, must be a constant

Pointers

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

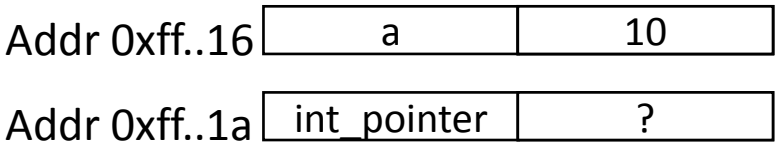
```
int main() {
        int a = 10;
        int * int_pointer; // currently points to an arbitrary location
        int_pointer = &a; // & returns the address of the variable

        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 1;
}
```



Pointers

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

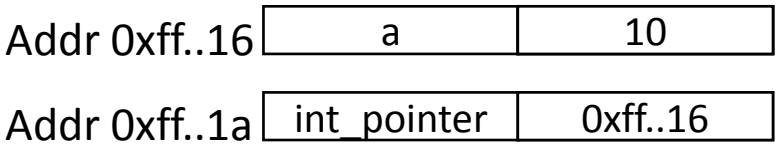
```
int main() {
        int a = 10;
        int * int_pointer; // currently points to an arbitrary location
        int_pointer = &a; // & returns the address of the variable

        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 1;
}
```



Pointers

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

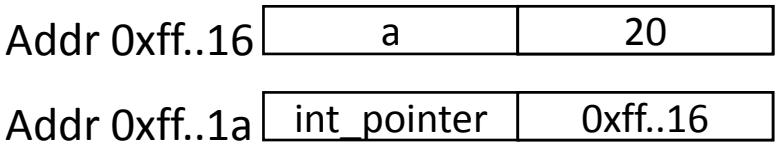
```
int main() {
        int a = 10;
        int * int_pointer; // currently points to an arbitrary location
        int_pointer = &a; // & returns the address of the variable

        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 1;
}
```



Pointers

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

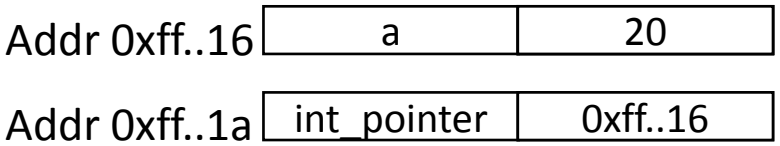
```
int main() {
        int a = 10;
        int * int_pointer; // currently points to an arbitrary location
        int_pointer = &a; // & returns the address of the variable

        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 1;
}
```



Pointers

- DataType * pointer
  - int *int_pointer;
  - 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 1;
}
```

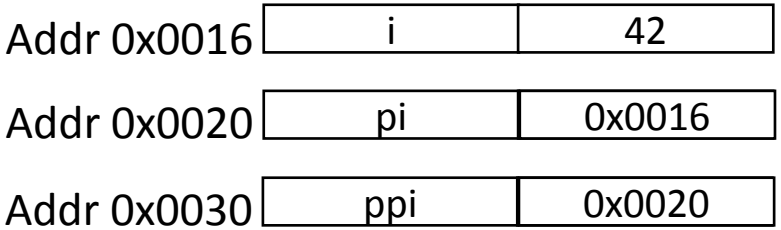


Sample Output:
0xff..16
10
0xff..16
20
20

Pointer of pointer

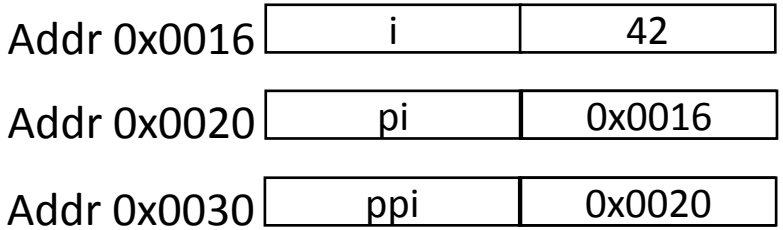
```
int i = 42;
int *pi = &i;
int **ppi = &pi;
printf("%d %d %d\n", i, *pi, **ppi);
```

What should be printed?



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

Function pointer

```
int add(int a, int b) {
    return a + b;}

int multiply(int a, int b) {
    return a * b;}

int main() {
    // Declare a function pointer
    int (*operation)(int, int);

    // Point it to the add function
    operation = add;
    printf("5 + 3 = %d\n", operation(5, 3));

    // Now point it to the multiply function
    operation = multiply;
    printf("5 * 3 = %d\n", operation(5, 3));

    return 0;
}
```

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 existing addresses
  - But sometimes you will access RELATIVE addresses

Arrays

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

- We will first discuss static arrays

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

- These will be stored contiguously in memory
- **nums points to the first element**

Arrays

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

```
float nums[4]; // create an array of 4
floats 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 - not a build-in type

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

- Relevant functions are 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 exactly the same object in memory
char msg1[6] = "Hello";
char msg2[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Complicated C declarations

```
int *a[10];
```

Complicated C declarations

```
int *a[10];
```

a is an array of 10 pointers to int.

Complicated C declarations

```
int (*a)[10];
```

Complicated C declarations

int (*a)[10];

a is a pointer to an array of 10 ints.

Step 1. Identify the variable. The identifier is a.

Step 2. Look around it. We see (*a)[10]. The parentheses matter: without them, it would mean something else. a is first dereferenced (*a), and then indexed as an array of 10. So that means: a is a pointer to an array of 10 …

Step 3. Look at the base type. The base type is int. So: a is a pointer to an array of 10 ints.

Complicated C declarations

```
int (*f[5])(int, int);
```

Complicated C declarations

```
int (*f[5])(int, int);
```

f is an array of 5 pointers to functions, each taking two ints and returning an int.

Complicated C declarations

```
int *(*f)(int, int);
```

Complicated C declarations

```
int *(*f)(int, int);
```

f is a pointer to a function that takes two ints and returns a pointer to int.

Dynamic memory allocations

- Memory can be allocated using the library function malloc
  - It is defined in stdlib.h
  - Takes the number of bytes we want
  - 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

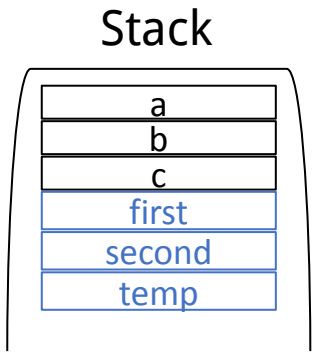
- We will mostly use malloc to allocate arrays and structs (below)

```
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;
}
```

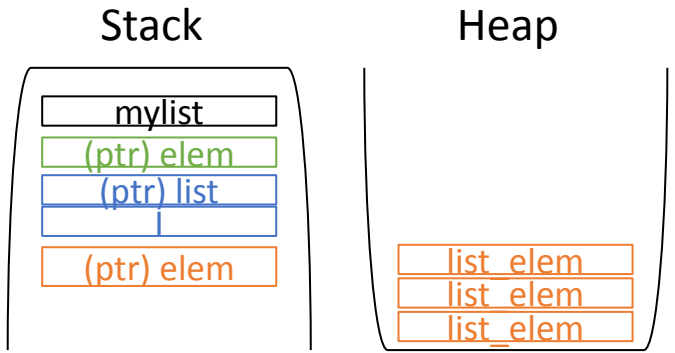


Pointers and memory management

- Stack vs heap
  - Heap memory is dynamically allocated and you should manage it
  - “malloc” allocates memory
  - “free” deallocates memory

```
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 will allocate 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 char 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 parameterized macro
  - E.g.,

    ```
    #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 2 * 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 2 * (10 + 2)
```

Preprocessor: #define

- Parametric (“function-like”) Macros
  - We can also define macros with arguments using #define
  - These look like function calls, but they get 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,
                                    // so 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
  - Example: #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

This set of directives allows conditional compilation. Basically, these are compile-time conditionals that hide or expose parts of the source file from or to the compiler

Example:

```
#ifdef UNIX
    #define PATH_SEPARATOR "/"
#elif defined(WINDOWS)    /* defined() is a C operator */
    #define PATH_SEPARATOR "\\"
#endif

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

- What if we include the same header twice?

mycode.h

```
struct my_struct {
        int x;
        int y;
};

int my_function(struct my_struct *my_arg);
```

mycode.c

```
#include “mycode.h”
#include “mycode.h”

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

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”
#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;

}
```

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

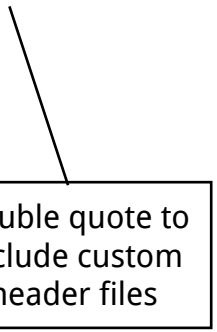
```
int my_min(int a, int b);
```

- 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;

}
```



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
```

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
```

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--; }
```

Inline Assembly

```c
#include <stdio.h>

int main() {
    int a = 5, b = 3, result;

    // Inline assembly (GCC-style)
    asm("addl %%ebx, %%eax"
        : "=a" (result)        // output in eax → result
        : "a" (a), "b" (b));   // inputs: a in eax, b in ebx

    printf("%d + %d = %d\n", a, b, result);
    return 0;
}
```

`asm(...)` or `__asm__` is a GCC extension for inline assembly.

Inside, you have three main parts:

```
"assembly"
: output operands
: input operands
```

The part after the first colon (`:`) lists **outputs**.

`"=a"` means: put the result into **EAX register** (`a` = `%eax`), and `=` means **write-only**.

`(result)` ties that register's content back to the C variable `result`.

Inputs come after the second colon.

`"a"(a)` → load the C variable `a` into **EAX** before the asm runs.
`"b"(b)` → load the C variable `b` into **EBX** before the asm runs.

`"a"` = use EAX
`"b"` = use EBX
`"c"` = ECX
`"d"` = EDX
`"r"` = any general register
`"m"` = memory
`=` write-only output, `+` read/write output.

Bitwise operations: AND &

```
#include <stdio.h>

int main() {
    int a = 6;   // binary:  0110
    int b = 3;   // binary:  0011
    int c = a & b;  // result: 0010 (decimal 2)

    printf("a & b = %d\n", c);
    return 0;
}
```

Bitwise operations: OR |

```
int a = 6;   // 0110
int b = 3;   // 0011
int c = a | b;  // 0111 = 7

printf("a | b = %d\n", c);
```

Bitwise operations: XOR ^

```
int a = 6;   // 0110
int b = 3;   // 0011
int c = a ^ b;  // 0101 = 5

printf("a ^ b = %d\n", c);
```

Bitwise operations: NOT ~

```
int a = 6;   // 0000...0110
int c = ~a;  // 1111...1001 (two's complement)

printf("~a = %d\n", c);
```

Bitwise operations: Left Shift <<

```
int a = 3;     // 0011
int c = a << 2; // 1100 = 12

printf("a << 2 = %d\n", c);
```

Bitwise operations: Right Shift >>

```
int a = 12;    // 1100
int c = a >> 2; // 0011 = 3

printf("a >> 2 = %d\n", c);
```

Bitwise use

```
// fcntl.h (x86 xv6)
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400

// usage:
open("out.txt", O_CREATE | O_WRONLY);
```

Bitwise use

```
// mmu.h (x86 xv6)
#define PTE_P   0x001   // Present
#define PTE_W   0x002   // Writeable
#define PTE_U   0x004   // User
#define PTE_PWT 0x008   // Write-Through
#define PTE_PCD 0x010   // Cache-Disable
#define PTE_A   0x020   // Accessed
#define PTE_D   0x040   // Dirty
#define PTE_PS  0x080   // Page Size
#define PTE_G   0x100   // Global

// masks to split address vs flags
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)  // clear low 12 bits
#define PTE_FLAGS(pte)  ((uint)(pte) & 0xFFF)   // keep low 12 bits

// usage example:
pte_t pte = *pteptr;
if (pte & PTE_P) {          // test "present"
    pte &= ~PTE_W;          // clear write flag
    *pteptr = pte;
}
uint pa = PTE_ADDR(pte);     // physical page base
```

What is a Makefile?

- A file named Makefile or makefile automates building your project.
- Contains **rules** that say *how to build targets (files) from sources.*
- make uses timestamps to rebuild only what changed.

```
# Basic rule
target: dependencies
<TAB>shell command to build target
<TAB>shell command to build target
<TAB>shell command to build target
<Non-TAB>
```

Variables

Variables store text you can reuse.

Assign with =

```
CC = gcc
CFLAGS = -Wall -O2

hello: hello.c
    $(CC) $(CFLAGS) hello.c -o hello
```

Automatic/Predefined Variables

Var	Meaning
$@	the target file
$<	first prerequisite
$^	all prerequisites

```
main.o: main.c defs.h
    $(CC) $(CFLAGS) -c $< -o $@
```

Pattern Rules

Rules with wildcards to avoid repeating yourself. in Makefiles the **% sign is a wildcard**.

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

Makefile for A3

```
BIN=mystery array-max

all: $(BIN)

mystery: mystery-main.s libmystery.a
  gcc -g -z noexecstack -no-pie mystery-main.s -L. -lmystery -o mystery

array-max: array-max.s array-max-main.c
  gcc -g -z noexecstack -no-pie array-max.s array-max-main.c -o array-max

clean:
  rm -f $(BIN)
  rm -f *.o
```

Errors you can get: Compiler Errors

These are caught by the compiler during translation from C to object code.

Causes:

- **Syntax errors** (typos, missing ;, wrong keywords)
- **Type errors** (mismatched types, undeclared variables, wrong function signatures)

```
#include <stdio.h>

int main() {
    int x = "hello";   // ❌ assigning string to int
    printf("%d\n", y); // ❌ 'y' undeclared
    return 0;
}
```

```
error: invalid initializer
error: 'y' undeclared
```

Errors you can get: Linker Errors

These happen after successful compilation, when the linker tries to combine object files into an executable.

Causes:

- Missing function definitions (declared but not defined).
- Multiple conflicting definitions.
- Forgetting to link a required library.

```
#include <stdio.h>

void greet();   // declared but never defined

int main() {
    greet();    // ❌ linker won't find it
    return 0;
}
```

undefined reference to 'greet'

Errors you can get: Runtime Errors

The program compiles and links, but fails **while running**.

Causes:

- Division by zero.
- Invalid memory access (segmentation fault).
- Infinite loops.
- Stack overflow.

```
#include <stdio.h>

int main() {
    int a = 5, b = 0;
    printf("%d\n", a / b);  // ❌ division by zero (runtime error)
    return 0;
}
```

Floating point exception (core dumped)

Errors you can get: Runtime Errors

The code runs but gives **wrong results**. These are the hardest to catch.

```
#include <stdio.h>

int main() {
    int fahrenheit = 100;
    int celsius = 5/9 * (fahrenheit - 32); // ❌ integer division bug
    printf("%d\n", celsius);  // prints 0 instead of 37
    return 0;
}
```

Compiles fine, runs fine, but logic is wrong (`5/9 = 0` in integer division).

How to debug C/assembly?

GDB Cheat Sheet

Start gdb using:
gdb <binary>
Pass initial commands for gdb through a file
gdb <binary> –x <initfile>

To start the program and breakpoint at main()
start <argv>

To start the program and breakpoint at _start
starti <argv>

To run the program without breakpoint
r <argv>
Use another progrom’s output as stdin in GDB:
r <<< $(python2 -c "print '\x12\x34'*5")

GDB Cheat Sheet

Set breakpoint at address:
b *0x80000000

Set breakpoint at beginning of a function:
b main

….
b <filename:line number>
b <line number>

Disassemble 10 instructions from an address:
x/10i 0x80000000

Exam 15 dword (w) from an address; show hex (x):
x/15wx 0x80000000

Exam 3 qword (g) from an address; show hex (x):
x/3gx 0x80000000

GDB Cheat Sheet

To show breakpoints
info b

To remove breakpoints
clear <function name>
clear *<instruction address>
clear <filename:line number>
clear <line number>

GDB Cheat Sheet

Use “examine” or “x” command
x/32xw <memory location> to see memory contents at memory location, showing 32 hexadecimal words
x/5s <memory location> to show 5 strings (null terminated) at a particular memory location
x/10i <memory location> to show 10 instructions at particular memory location

See registers
info reg

Step an instruction
si

How to debug C/assembly?

codespace A3 as an example