Course Review

Topics

- Machine representation
- Strings, Arrays, Structures, Unions
- Pointers, Pointer Arithmetic
- Function Pointers and Callbacks
- Memory Management
- Data Structures
- Generics

Very important

- Undefined Behavior
- Optimizations
- I/O

Important

- Signal Handling
- Rust
- Asserts, Error Handling
- Control Stack Layout

Less important

Data Types

The base data type in C

- `int` - used for integer numbers
- `float` - used for floating point numbers
- `double` - used for large floating point numbers
- `char` - used for characters
- `void` - used for functions without parameters or return value
- `enum` - used for enumerations

The composite types are

- pointers to other types
- functions with arguments types and a return type
- arrays of other types
- `structs` with fields of other types
- `unions` of several types

Representing Numbers

Infinite set of numbers, but finite representation

- Everything must be represented in terms of 1's and 0's
- Machine architecture and instruction set dictates size of operands for arithmetic operations

  Example: 32-bit operands implies being able to represent 2^{32} integers
- Unsigned vs. signed numbers

  unsigned: represent positive numbers from 0

  signed: represent both positive and negative numbers
- Floating-point numbers

  a representation for real numbers

Qualifiers, Modifiers & Storage

Type qualifiers

- `short` - decrease storage size
- `long` - increase storage size
- `signed` - request signed representation
- `unsigned` - request unsigned representation

Type modifiers

- `volatile` - value may change without being written to by the program
- `const` - value not expected to change

Storage class

- `static` - variable that are global to the program
- `extern` - variables that are declared in another file

Sizes

Type	Range (32-bits)	Size in bytes
`signed char`	–128 to +127	1
`unsigned char`	0 to +255	1
`signed short int`	–32768 to +32767	2
`unsigned short int`	0 to +65535	2
`signed int`	–2147483648 to +2147483647	4
`unsigned int`	0 to +4294967295	4
`signed long int`	–2147483648 to +2147483647	4 or 8
`unsigned long int`	0 to +4294967295	4 or 8
`signed long long int`	–9223372036854775808 to +9223372036854775807	8
`unsigned long long int`	0 to +18446744073709551615	8
`float`	1×10^{-37} to 1×10^{37}	4
`double`	1×10^{-308} to 1×10^{308}	8
`long double`	1×10^{-308} to 1×10^{308}	8, 12, or 16

Character representation

ASCII code (American Standard Code for Information Interchange): defines 128 character codes (from 0 to 127),

In addition to the 128 standard ASCII codes there are other 128 that are known as extended ASCII, and that are platform-dependent.

Examples:

- The code for ‘A’ is 65
- The code for ‘a’ is 97
- The code for ‘b’ is 98
- The code for ‘0’ is 48
- The code for ‘1’ is 49

Opening files

`FILE* fopen(const char* filename, const char* mode)`

- mode can be “r” (read), “w” (write), “a” (append)

returns NULL on error (e.g., improper permissions)

filename is a string that holds the name of the file on disk

`int fileno(FILE *stream)`

- returns the file descriptor associated with stream

```
char *mode = "r";
FILE* ifp = fopen("in.list", mode);
if(ifp==NULL){fprintf(stderr,"Failed");exit(1);}
FILE* ofp = fopen("out.list", "w");
if (ofp==NULL) {...}
```

Reading files

`fscanf` requires a `FILE*` for the file to be read

```
        fscanf(ifp, “<format string>”, inputs)
```

Returns the number of values read or EOF on an end of file

Example: Suppose ifp is a file pointer to a file containing

```
    foo 70
    bar 50
```

To read elements from this file, we might write

```
   fscanf(ifp, “%s  %d”, name, count)
```

Can check against EOF:

```
   while(fscanf(ifp,“%s %d”,name,count)!=EOF);
```

External and static variables

C differentiates variable *definitions* from *declarations*.

- A declaration introduces a name and its type to allow the compiler to compile code that references that name.
- A definition is a declaration that also generates storage for the identifier based on its type. This is needed to link and execute code.

A **declaration** is a like a *customs declaration*

it is not the thing itself, merely
a description of some baggage that
you say you have around somewhere

a **definition** is the special kind of declaration
that fixes the storage for that thing

Libraries

A library in C ...

- contains object files used together in the linking phase of a program
- is indexed, so it is easy to find function and variable symbols

Static libraries:

- collections of object files that are linked into the program

Examples:

- Unix: libXXX.a
- Windows: XXX.lib

Command line:

- `gcc -ltweetit.a main.c`

Basics

`char c;` *declares a variable of type character*

`char* pc;` *declares a variable of type pointer to character*

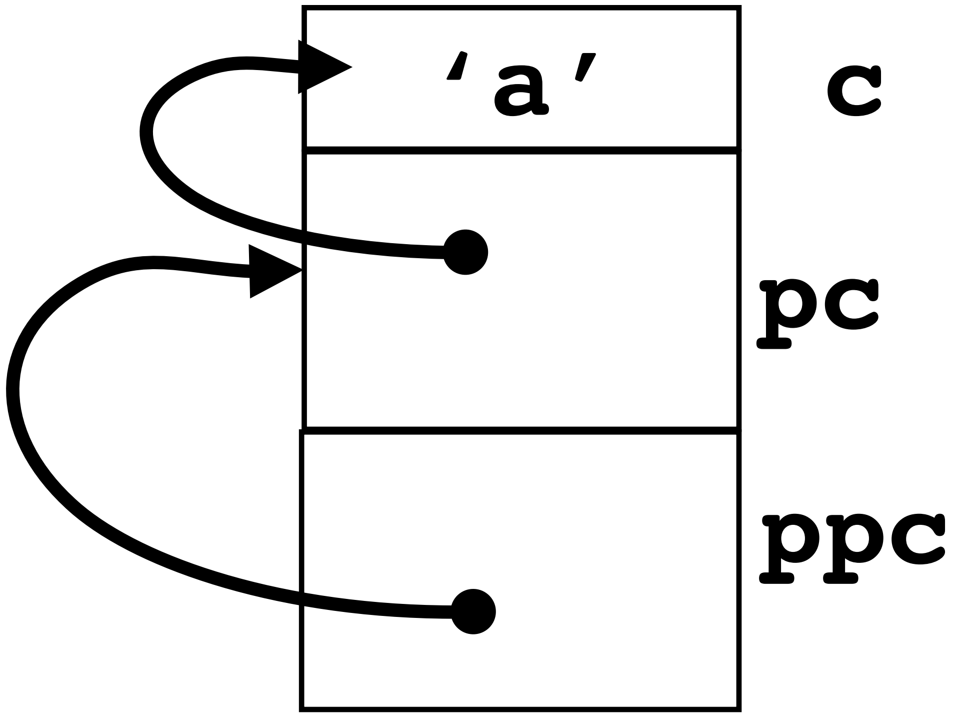
`char** ppc;` *declares a variable of type pointer to pointer to character*

`c = 'a';` *initialize a character variable*

`pc = &c;` *get the address of a variable*

`ppc = &pc;` *get the address of a variable*

```
c == *pc == **ppc
```



Arrays & pointers

```
int c = 0, in = 0;
char buf[2048]; char *p = buf;

while((c = getchar()) != EOF) {
    if(c=='<' || c=='&')  in = 1;
    if(in)  *p++=c;
    if(c=='>' || c==';') {
        in = 0; *p++ = '\0';
    }
}
```

- `buf` is an array of 2048 characters;
- `p` is pointer in the buffer
- boolean value `false` is 0, any non-0 is `true`

Arrays

```
char buf[2048]; int pos=0;
while((c = getchar()) != EOF) {
  ...
  if(in) buf[pos++] = c;
  if(c=='>'||c==';') {
    buf[pos++]='\0';
    pos=0;
  }
```

- the same program without pointers
- an alternative to pointers is to use an index in the array of chars
- strings must be `\0` terminated (or risk a buffer overflow...)

Memory layout

Data alignment: when the processor accesses memory, it usually reads a word at a time; on a 32-bit platform, reads entail reading 4 bytes at a time.

What if the data structure is not a multiple of 4?

- Padding: some unused bytes are inserted in the structure by the compiler

  Compiler will use padding to make sure field accesses are word-aligned. On a 32-bit machine, this means the address of every field is at a 4-byte boundary; on a 64-bit machine, the address of every field is at an 8-byte boundary.

  The sizeof() operator returns the amount of memory used by a structure, taking padding into account.

Endian-ness

The order of bytes in a word or multi- byte value

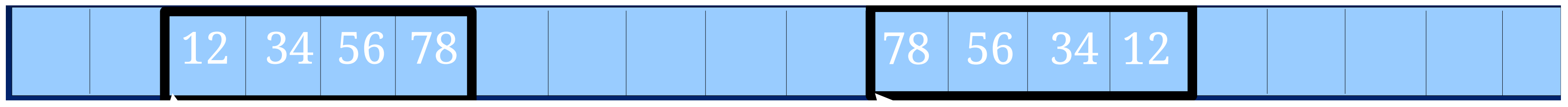
- Does not impact bit ordering for individual bytes!

Two ways:

- Big-endian: most significant byte first (lowest address)
- Little-endian: least significant byte first

Example:

Representing the integer 305419896 in memory
In hexadecimal: 0x 12 34 56 78
each pair of hex digits corresponds to a byte



Big endian

Little endian

Structures and functions

Structures can be initialized, copied as any other value

They can not be compared directly

- instead one must write code to compare members one by one
- Or compare the addresses of the structures *(usually not the right answer)*

Functions can return structure instances

- What is the cost in terms of memory allocation, copy, and performance?
- What's the difference between arrays and structures in this sense?

```
 struct pt { int x, y; };

struct pt mkpt(int x, int y) {
    struct pt t; t.x = x; t.y = y; return t;
}

struct pt p1 = mkpt(0, 0);
```

Recursive structures

What is the meaning of

```
    struct rec { int i;  struct rec r; }
```

A structure cannot refer itself directly.

The only way to create a recursive structure is to use pointers

```
 struct node {
    char * word;
    int count;
    struct node *left,*right;
}
```

Unions

```
typedef union {int units; float kgs;} amount;
```

Unions can hold different type of values at different times

Definition similar to a structure but

- storage is shared between members
- only one field present at a time
- programmers must keep track of what it is stored

Useful for defining values that range over different types

- Critically, the memory allocated for these types is shared

Memory layout

- All members have offset zero from the base
- Size is big enough to hold the widest member
- The alignment is appropriate for all the types in the union

Pointers and arrays

There is a strong relationship between pointers and arrays

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

A pointer (e.g. `p`) holds an address while the name of an array (e.g. `a`) denotes an address

Thus it is possible to convert arrays to pointers

```
    p = a;
```

Array operations have equivalent pointer operations

```
    a[5]      ==      *( p + 5 )
```

Note that `a=p` or `a++` are compile-time errors.

Pointers to arrays

```
char a[2][3];
```

Multi-dimensional array that stores two strings of 3 characters. (Not necessarily zero-terminated)

```
char a[2][3]={“ah”,”oh”};
```

Array initialized with 2 zero-terminated strings.

```
char *p = &a[1];
while( *p != ‘\0’ ) p++;
```

Iterate over the second string

Character arrays and pointers

What’s the difference between char s[] and char* s?

- As a declaration, none:

      int f(char* s)

      int f(char s[])

- As a definition:

  char s[] = “hello”
  - Allocates the string in modifiable memory, and defines s to be a pointer to the head of the string.
  - Can change the contents, but s will always point to the same place
    - Can’t write: s = p; an array name is not a variable (i.e., can’t be used as an l-value)

  char* s = “hello”
  - Allocates a pointer (freely modifiable)
  - Allocates a string (not modifiable) - typically allocated in the text segment of memory
  - s points to the beginning of the string, but modifications to the string (e.g., *s = ‘x’) is undefined
  - s can be reassigned to point to other strings

Parameter passing

Historically programming languages have offered:

- passed by value
- passed by reference
- passed by value-result

By-value semantics:

- Copy of param on function entry, initialized to value passed by caller
- Updates of param inside callee made only to copy
- Caller's value is not changed (updates to param not visible after return)

The Stack

Logically it's a last-in-first-out (LIFO) structure

Two operations: push and pop

Grows 'down' from high-addresses to low

Operations always happen at the top

Holds "activation frames", i.e. state of functions as they execute

- top-most (lowest) frame corresponds to the currently executing function

Stores local variables and address of the function that needs to be executed next

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl  8(%ebp), %edx
    movl  12(%ebp), %ecx
    movl  (%edx), %ebx
    movl  (%ecx), %eax
    movl  %eax, (%edx)
    movl  %ebx, (%ecx)
```

Recall Stack Layout

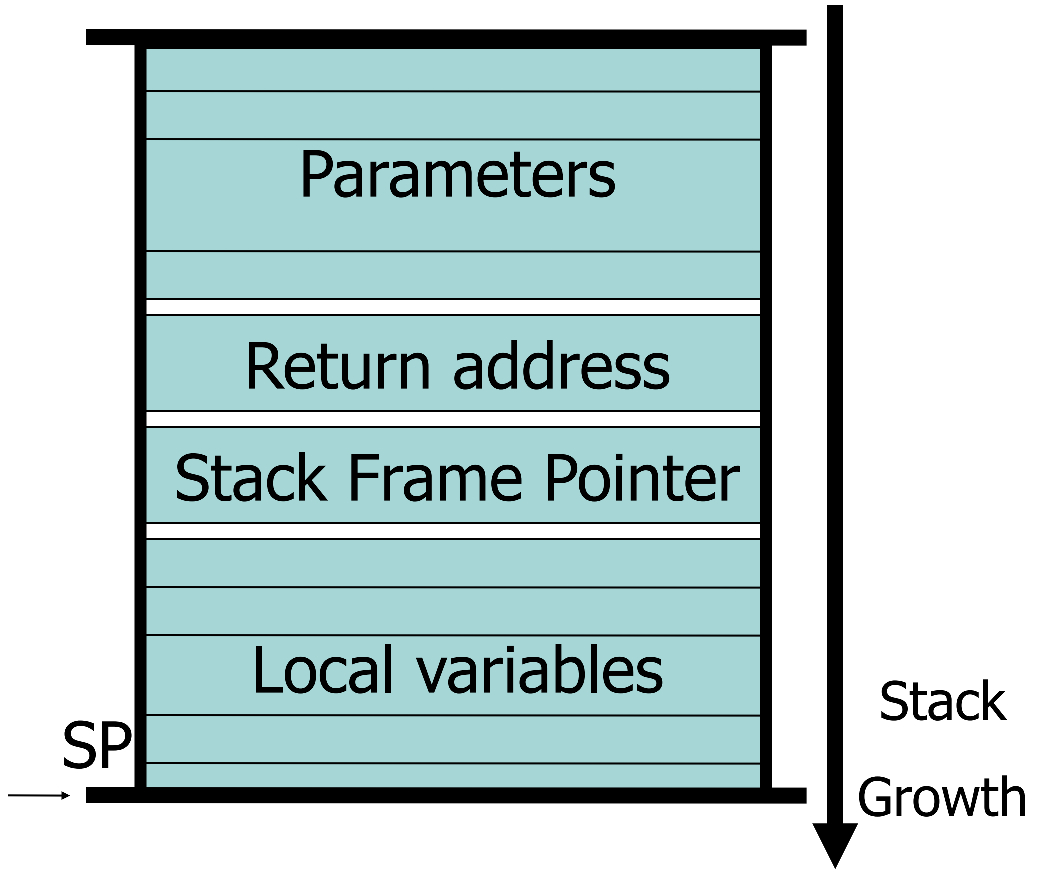
Parameters for the procedure

Save current PC onto stack (return address)

Save current SP value onto stack

Allocates stack space for local variables by decrementing SP by appropriate amount

Return value passed by register



Dynamic memory management

```
#include <stdlib.h>


void* calloc(size_t n, size_t s)
void* malloc(size_t s)
void  free(void* p)
void* realloc(void* p, size_t s)
```

Allocate and free dynamic memory

Operations with memory

#include <string.h>

```
void* memset(void *s, int c, size_t n)
void* memcpy(void *s, const void *s2, size_t n)
void* memove(void *s, const void *s2, size_t n)
```

Initializing and copying blocks of memory

Memory Allocation Problems

Memory leaks

- Alloc'd memory not freed appropriately
- If your program runs a long time, it will run out of memory or slow down the system
- Always add the free on all control flow paths after a malloc

```
void *ptr = malloc(size);
/*the buffer needs to double*/
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL)
   /*realloc failed, original address in ptr
    lost; a leak has occurred*/
   return 1;
```

Linked-List

- Represent the "next-node" field as a pointer
- Allocate new nodes using `malloc()`
- Remove nodes using `free()`

```
struct  lnode
{
   int        item;
   struct     lnode *next;
};
```

This describes the components of the struct; **no memory is allocated**

Now we could write

```
struct   lnode *np;
np = (struct lnode*)malloc(sizeof(struct lnode));
np->Item = 3;
np->Next = NULL;
free(np);
```

allocate memory for a pointer

allocate memory for a node

deallocate node's memory

Stack

```
typedef struct node {
  struct node *next;
  void *data;
} node;

typedef struct stack {
  int nelems;
  int elem_size_bytes;
  node *top;
} stack;
```

Each stack element stores a pointer to the next node in the stack and a pointer to the data the node stores.

A stack contains meta-data on the number of elements, the size of each element, and a pointer to the current 'top-of-stack'

- A stack is generic, but not heterogeneous

Operations:

- `stack *stack_create(int elem_size_bytes)`
- `void stack_push(stack *s, const void *data)`
- `void stack_pop(stack *s, void *addr)`

Fold

Now, lets define abstractions that compute over lists

```
int fold(int(*f)(int,int), List *l, int acc) {
    if (l == NULL)
        return acc;
    else {
        int x = l->val;
        return fold (f, l->next, (*f)(x,acc));
    }
}
```

a list of integers

an accumulator

A function pointer that operates over pairs of integers and returns an int

Each recursive call to fold operates on the current value and the current accumulator; the result becomes the new value of the accumulator in the next call

Map

```
List* map(int(*f)(int), List *l) {
   if (l == NULL) return l;
   List * l1 = malloc(sizeof(List));
   l1->val  = (*f)(l->val);
   l1->next = map( f, l->next);
   return l1;
}
```

A function pointer that points to a function which takes an integer argument and produces an integer result

Apply the function pointed to by f to the current list element

Recursively apply map to the rest of the list

void type

A pointer to a void type points to a value that has no type.

- This means there are no allowable operations on them.
- Must cast void pointers to concrete type in order to access its target value
- One use of void pointers is to pass “generic” parameters to a function

```
void f (void* data, int sz) {
    if (sz == sizeof(char)) {
      char* p = (char*)data; ++(*p);
    } else if (sz == sizeof(int)) {
      int* p = (int*)data; ++(*p);
    }
}
```

Generic Implementation

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                 int (*cmp fn)(void *, void *)) {
     while (true) {
         bool swapped = false;
         for (size_t i = 1; i < n; i++) {
             void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
             void *p_curr_elem = (char *)arr + i * elem_size_bytes;
             if (cmp_fn(p_prev_elem, p_curr_elem) > 0) {
                 swap(p_prev_elem, p_curr_elem, elem_size_bytes);
                 swapped = true;
             }
         }
         if (!swapped) {
             return;
         }
     }
 }
```

extract data based on size of each element

swap element based on element size

Comparison is now over bytes

Recall generic swap

```
void swap(void *x, void *y, size_t nbytes) {
  char tmp[nbytes];
  // store a copy of x in temporary storage
  memcpy(tmp, x, nbytes);
  // copy y to location of x
  memcpy(x, y, nbytes);
  // copy data in temporary storage to location of data2
  memcpy(y, tmp, nbytes);
```

Defining a signal handler

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

`signal` is a function pointer to a function that takes as arguments:

- a signal (represented as an int)
- a handler

and returns a function that takes an `int` and returns `void`

The handler is a function pointer to a function that takes an `int` and returns `void`.

Optimizations

- Inlining
- Constant Folding
- Common Subexpression Elimination
- Dead Code Elimination
- Strength Reduction
- Loop Unrolling
- Tail Recursion
- Code Motion
- Layout

Optimizations target one or more of:

- static instruction count
- dynamic instruction count
- execution time

Optimizing along one dimension doesn't necessarily result in optimization along another

Knowing the kinds of optimizations C performs can lead you to write better code, either by:

- performing them yourself
- writing code in ways that enable them

Optimizations

- Lots of wiggle room
- Three degrees of freedom provided to a compiler:
  - Implementation-dependent: compiler must document and choose a consistent behavior
  - Unspecified: multiple possibilities; compiler must choose one of those
  - Undefined: compiler is free to emit code that can produce arbitrary behavior

Undefined Behavior

– Examples:

- The value of a pointer to an object whose lifetime has ended:

```c
#include <stdio.h>

char *unary(unsigned short s) {
  char local[s+1];
  int i;
  for (i=0;i<s;i++) local[i]='1';
  local[s]='\0';
  return local;
}

int main(void) {
  printf("%s\n",unary(6)); //What does this print?
  return 0;
}
```

- Pointer conversion produces a value outside representable range:

```c
char* p = malloc(10);   short x = (short) p;
```

- Object that's modified multiple times between two sequence points

  Eg: int i = 0; `f(i++, i++)` – what arguments will f be called with?

Undefined Behavior

- Accessing an uninitialized variable

```
int main(int argc, char **argv) {
  int i;
  while (i < 10) {
    printf("%d\n", i);
    i++;
  }
}
```

- Accessing out-of-bounds memory

```
char *buf = malloc(10);
buf[10] = '\0';
```

- Null dereference or `wild` pointer

```
char *buf = malloc(10);
buf[0] = 'a'; // what if buf is NUL?
free(buf);
buf[0] = '\0'; // buf has been freed
```

Undefined Behavior

- Signed integer overflow

  – `INT_MAX + 1 != INT_MIN`

  - Knowing signed values cannot overflow (and still be defined) enables useful optimizations:

  E.g., consider: `for (i = 0, i < N, ++i) { … }`

  Because signed integer overflow is undefined, the compiler can assume that the loop runs exactly N+1 times

  If this wasn't the case, then it would be difficult to reason about code like:

  ```
  for (i = 0, i < INT_MAX, ++i) { … }
  ```

Summary

- The compiler only has to execute statements where the behavior is defined, and can optimize the rest away.
- In the C abstract machine, every operation that is being performed is either defined or undefined (as classified by the C standard).
- Application developers need to worry about every input to the program being defined, as the compiler is an "adversary" that can jump on undefined behavior and subvert the original "intention".

What to fear/check for...

Null pointer dereference
Use after free
Double free
Array indexing errors
Mismatched array new/delete
Potential stack/heap overrun
Return pointers to local variables
Logically inconsistent code
Uninitialized variables
Invalid use of negative values
Under allocations of dynamic data
Memory leaks
File handle leaks
Unhandled return codes

Rust Key Innovations: Ownership and Borrowing

```rust
fn consume(w: Vec<i32>) {
  drop(w); // deallocate vector
}
let v = vec![10, 11];
consume(v);
v.push(12); // Compiler error
```

Could have been automatically inserted by the compiler

Ownership

pass-by-value

track lifetimes

```rust
fn add_something(v: &mut Vec<i32>) {
  v.push(11);
}
let mut v = vec![10];
add_something(&mut v);
v.push(12); // Ok!
// v.push(12) is syntactic sugar for Vec::push(&mut v, 12)
```

Borrowing

pass-by-reference