Pointers and Lexical Scoping

CS449 Fall 2015
Pointers Review

• Pointer: Variable (or storage location) that stores the address of another location.
• Reference operator: e.g. “scanf(“%d”, &x);”
  – Address of “x” is passed in order to modify it
• Dereference operator: e.g. “*p = 0;”
  – Access the location pointed to by “p”
• Pointer to array vs. array of pointers
• Value of array variable is the address to the first element of the array
  – Given “int a[3];”, “a == &a[0];” by definition
  – Thus can be stored in a pointer: “int *p = a; p[0] = 0;”
  – However, “a = p;” results in a compile error since &a[0] is not an l-value (a storage location)
Size of Derived Types

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

int main()
{
    char buf[20];
    char *str = buf;
    strcpy(buf, "Hello");
    printf("str=%s, buf=%s\n", str, buf);
    printf("sizeof(str)=%u\nssizeof(buf)=%u\n
"Hello")=%u\n", sizeof(str),
    sizeof(buf), sizeof("Hello"));
    return 0;
}
```

```
>> ./a.out
str=Hello, buf=Hello
sizeof(str)=8
sizeof(buf)=20
sizeof("Hello")=6
```
Functions Review

• One way pointers are useful.. when passing pointer arguments to function
  – When modifying variables in caller’s environment
    • E.g. “void swap(int *a, int *b);”
    • In a sense, allows the “return” of multiple values
  – When it’s more efficient (to avoid copying)
    • E.g. “size_t strlen(const char* s);”
    • string “s” is not getting modified but pointer is passed instead of copying character array
    • What if we declare “size_t strlen(const char s[100]);”?  
    • Will compile but same meaning as above. Don’t bother using it. Arrays are always passed as pointers.
Const Qualifier

- What if you want to make sure a string passed as an argument is never modified?
- Declare string as const
  - e.g. “size_t strlen(const char* s);”
- Contract to caller that string pointed to by “s” will not be modified
- If contract is violated -> compile time error
  - With no contract -> runtime error (much harder to debug)
- Examples:
  - const float pi = 3.14;
  - const char *str = “Hello”; 
  - const char *const str = “Hello”; 
    - Contract: String pointed to by “str” is immutable AND “str” cannot point to any other string
Example Use of Const

#include <stdio.h>

void foo(const char *s) { s[0] = 'h'; }

int main()
{
    const char *str = "Hello";
    foo(str);
    return 0;
}
Example Use of Const

#include <stdio.h>

int main()
{
    const char *str = "Hello";
    char *str2 = str;
    str2[0] = 'h';
    return 0;
}

After this:

>> gcc ./main.c
./main.c: In function ‘main’:
./main.c:6: warning: initialization discards qualifiers from pointer target type

You get:

>> ./a.out
Segmentation fault (core dumped)

• Assigning “const char*” to “char*” can potentially lead to violation of contract (Thus the warning.)
Pointer Arithmetic

• Another convenience offered by pointers
• Assuming “int a[3];”, following are equivalent:
  – a[2];
    • Get element 2 of int array “a”
  – *(a + 2);
    • Get element 2 offsets away from “a” or “&a[0]”
  – *(int*)((size_t)a + sizeof(int) * 2)
    • Direct address calculation of above
Operations Permitted on Pointers

• Add constant offset (+, +=, ++)  
  – E.g. “p = p + 1;”, “p += 1;”, “++p;”

• Subtract constant offset (-, -=, --)

• Offset between two pointers  
  – E.g. “int offset = p1 – p2;”

• Comparison between two pointers. E.g.:  
  – “p1 > p2” (p1’s offset is larger than p2’s offset)  
  – “p1 == NULL” (if p1 is equal to NULL value)
Strcpy Using Pointer Arithmetic

char* strcpy(char *dest, const char *src) {
    char *p = dest;
    while(*p++ = *src++) ;
    return dest;
}

• Stops when *src == ‘\0’ (when the null character at the end of src is reached)
Another Pointer Arithmetic Example

#include <stdio.h>
int main()
{
    int a[2][3];
    int *p = a[0];
    int (*p2)[3] = a;
    printf("p=%p, &a[0][0]=%p\n", p, &a[0][0]);
    printf("p2=%p, &a[0]=%p\n", p2, &a[0]);
    printf("p+1=%p, &a[0][1]=%p\n",
            p+1, &a[0][1]);
    printf("p2+1=%p, &a[1]=%p\n",
            p2+1, &a[1]);
    return 0;
}
# Another Pointer Arithmetic Example

```c
#include <stdio.h>
int main()
{
    int a[2][3];
    int *p = a[0];
    int (*p2)[3] = a;
    printf("p=%p, &a[0][0]=%p\n", p, &a[0][0]);
    printf("p2=%p, &a[0]=%p\n", p2, &a[0]);
    printf("p+1=%p, &a[0][1]=%p\n", p+1, &a[0][1]);
    printf("p2+1=%p, &a[1]=%p\n", p2+1, &a[1]);
    return 0;
}
```

- "p" and "p2" points to the same address
  - p == &a[0][0]
  - p2 == &a[0]

- "p+1" and "p2+1" point to different addresses
  - p+1 == &a[0][1] (base + sizeof(int))
  - p2+1 == &a[1] (base + sizeof(int[3]))

- Why are pointers typed differently depending on base type?
  - To perform accurate pointer arithmetic
  - To know what code to generate on a dereference operation (what is the size of the data to be read?)
  - To enable type checking when an incompatible type is assigned
The void* Type

• Assigning to different pointer type results in compile error.
  – E.g. “int *p; char *p2 = p;” results in error
• Except when assigning to void* type
  – E.g. “int *p; void *p2 = p;” is perfectly fine
• Void pointer (void *)
  – Generic pointer representing any type
  – No casting needed when assigning to void* (vice versa)
  – Cannot be dereferenced / no pointer arithmetic
    • Size and type of variable pointed to not known
  – Used when the base type of a variable is unknown at compile time (typically will be known at runtime)
The NULL Value

• Equivalent to the numerical value “0”. (Just like ‘\0’ is equivalent to “0”)

• NULL value means pointer points to nothing

• Make it a habit to initialize all invalid pointers to NULL. Advantages:
  – Can easily compare to NULL to check if pointer is valid
  – If accessing invalid pointer by mistake
    • Will result in a (clean) segmentation fault
    • Instead of accessing and corrupting some random memory
Command Line Arguments

#include <stdio.h>
int main (int argc, char *argv[])
{
    int i;
    printf("command: %s.\n", argv[0]);
    for (i = 1; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}

• argc: total number of command line arguments (including command itself)
• argv: string array that contains the command line arguments

>> ./a.out foo bar
command: "./a.out".
argv[1] = foo
argv[2] = bar
Lexical Scopes

• Scope: the portion of source code in which a symbol is legal and meaningful
  – Symbol: name of variable, constant, or function
  – At compile time, compiler matches each symbol to its corresponding memory location using scoping rules

• C defines four types of scopes
  – Block scope: within curly braces (e.g. within for loop)
  – Function scope: within functions
  – Internal linkage scope: within a single C source file
  – External linkage scope: global across entire program

• Means of encapsulation and data-hiding
  – In order to maximize encapsulation, minimize usage of globals
Lexical Scope Example

```c
int global;
static int file;
int main()
{
    int function;
    {
        int block;
    }
}

<main.c>

extern int global;
void foo() { global = 10; }

<foo.c>
```

- "int block": Block Scope
  - Only visible within curly braces
- "int function": Function Scope
  - Only visible within "main()" function
- "static int file": Internal Linkage Scope
  - Only visible within "main.c" file
  - `static`: storage class specifier limiting the scope
- "int global": External Linkage Scope
  - Visible across entire program
  - In foo.c, declaring "extern int global" tells compiler "global" refers to a variable defined elsewhere
  - `extern`: storage class specifier declaring external linkage
  - Cannot declare a static global variable extern
- Also applies to functions except functions cannot be defined inside another function (only declared)
Shadowing

#include <stdio.h>
int n = 10;
void foo() {
    int n = 5;
    printf("Second: n=%d\n", n);
}
int main() {
    printf("First: n=%d\n", n);
    foo();
    printf("Third: n=%d\n", n);
}
Shadowing

#include <stdio.h>
int n = 10;
void foo() {
  int n = 5;
  printf("Second: n=%d\n", n);
}
int main()
{
  printf("First: n=%d\n", n);
  foo();
  printf("Third: n=%d\n", n);
}
Lifetime

• Lifetime: time from which a particular memory location is allocated until it is deallocated
  – Only applies to variables
  – Is a runtime property and describes behavior of program while executes (unlike scopes which is a compile time property)

• C defines three types of lifetimes
  – Automatic: automatically created and destroyed by code generated by compiler at scope begin and end
  – Static: allocated at program initialization and destroyed at program termination (static is an overloaded term)
  – Manual: manually created and destroyed by the programmer on the heap (will discuss this later)

• Allows efficient management of memory by compiler
  – Avoid static when possible to allow memory to be reclaimed

• Static variables are guaranteed to be initialized to 0
  – Done once by the Standard C Library at runtime before calling “main()”
Storage Classes

• Storage class: combination of variable scope and lifetime

<table>
<thead>
<tr>
<th></th>
<th>Block</th>
<th>Function</th>
<th>Internal Linkage</th>
<th>External Linkage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>local</td>
<td>local</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Static</td>
<td>static local</td>
<td>static local</td>
<td>static global</td>
<td>global</td>
</tr>
</tbody>
</table>

• local: Visible within curly braces and valid while executing code inside block or function
• static global: Visible within file and valid for entire duration
• global: Visible globally and valid for entire duration
• static local: (We haven’t seen this yet) Visible within curly braces and valid for entire duration
Example of Wrong Storage Class

#include <stdio.h>
int* foo() {
    int x = 5;
    return &x;
}
void bar() { int y = 10; }
int main()
{
    int *p = foo();
    printf("*p=%d\n", *p);
    bar();
    printf("*p=%d\n", *p);
    return 0;
}
#include <stdio.h>
int* foo() {
    int x = 5;
    return &x;
}
void bar() { int y = 10; }
int main()
{
    int *p = foo();
    printf("*p=%d\n", *p);
    bar();
    printf("*p=%d\n", *p);
    return 0;
}
#include <stdio.h>

int* foo() {
    static int x = 5;
    return &x;
}

void bar() { int y = 10; }

int main() {
    int *p = foo();
    printf("*p=%d\n", *p);
    bar();
    printf("*p=%d\n", *p);
    return 0;
}
Returning String Using Static Local

char *asctime(const struct tm *timeptr) {
    static char wday_name[7][3] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
    static char result[26];
    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n", wday_name[timeptr->tm_wday], mon_name[timeptr->tm_mon],
    timeptr->tm_mday, timeptr->tm_hour, timeptr->tm_min, timeptr-
    >tm_sec , 1900 + timeptr->tm_year);
    return result;
}
• “result” string still valid even after function returns
Keeping Track of Internal State
Using Static Local

void foo() {
    static unsigned int count = 0;
    printf("Foo called %d time(s).\n", ++count);
}

• “count” is a property of function foo(), and as such is properly encapsulated through scoping
• But persistence of “count” allows tracking of internal state across calls to foo
Keeping Track of Internal State Using Static Local

```c
int main() {
    char str[] = "Blue,White,Red";
    char *tok = strtok(str, ",");
    while(tok != NULL) {
        printf("token: %s\n", tok);
        tok = strtok(NULL, ",");
    }
    return 0;
}
```

• **strtok(char *str, const char *delim):** parses “str” into a sequence of tokens using “delim” as delimiter (returns a token at each call)
• First call to strtok: return first token by replacing first occurrence of “delim” in “str” with ‘\0’ null character
• Subsequent calls to strtok (with NULL argument): return next token by picking up where we left off in “str” and replacing next occurrence of “delim” with ‘\0’ null character
• Location of next token is kept across calls using a static local variable
Register Storage Class Specifier

• Third storage class specifier (besides “static” and “extern”)
  – e.g. “register int x;”
• Tells the compiler to allocate the variable in a CPU register rather than memory, if possible
• Speeds up access for frequently used variables
• Can only be used for local (automatic) variables, which have a finite lifetime
  – For static variables, cannot allocate location in a register for entire duration of program
• Mostly obsolete and ignored by compiler since register allocators do a better job nowadays
Volatile Storage Class Specifier

• Fourth storage class specifier (besides “static”, “extern”, and “register”)
  – e.g. “volatile int x;”
• Tells the compiler to never allocate the variable in a register and always in memory
• Used for hardware devices when memory mapped I/O is required
• (Often mistakenly) used for multithreaded programming to force communication through memory
  – Not recommended since volatile does not establish a happens-before relationship. We will talk more about happens-before in the synchronization section.