Pointers and Lexical Scoping

CS449 Fall 2016

Review: Pointers

- Pointer: Variable (or storage location) that stores the address of another location.
- Reference operator: e.g. "scanf("%d", **&**x);"
  - Pointer to "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
  - "char a[2][3]; char (*p)[3] = &a[1];" // pointer to array
  - "char *p[2] = {"yes", "no"};" // array of pointers
- Value of array: pointer to the first element of array
  - Given "int a[3];", "a == &a[0];"
  - Thus can be stored in a pointer: "int *p = a; p[0] = 0;"

Review: Size of Pointers and Arrays

```
#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\nsizeof(buf)=%u\n
sizeof(\"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
```

Pointer Arithmetic

- A small subset of arithmetic operators are allowed on pointers
- Assuming “int a[3];”, following are equivalent:
  - &a[2];
    - Get address of element 2 of int array “a”
  - a + 2;
    - Get address 2 offsets away from “a” (or “&a[0]”)
  - (size_t)a + sizeof(int) * 2
    - Direct address calculation of above

Arithmetic Ops Permitted on Pointers

- Pointer + Number
  - Result: address Number offsets away from Pointer
  - Also includes syntactic sugar: +=, ++
  - E.g. “p = p + 1;”, “p += 1;”, “++p;”
- Pointer – Number (Same as above)
- Pointer1 – Pointer2
  - Result: Offset between Pointer1 and Pointer2
  - Pointer1 and Pointer2 must be the same type
  - E.g. “int offset = p1 – p2;”
- Comparison between two pointers.
  - Result: Numerical comparison between the two addresses
  - E.g. “p1 == 0x1000” (if p1 is equal to address 0x1000)
- Food for thought: Why not allow other operations?
  - What does Pointer * Number or Pointer * Pointer mean?

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)

Types Allow Correct Pointer Arithmetic

```
#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;
}
```

```
>> ./a.out
p=0xbfdb6374, &a[0][0]=0xbfdb6374
p2=0xbfdb6374, &a[0]=0xbfdb6374
p+1=0xbfdb6378, &a[0][1]=0xbfdb6378
p2+1=0xbfdb6380, &a[1]=0xbfdb6380
```

- "p" and "p2" point 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]))

Types Allow Correct Pointer Arithmetic

```
#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;
}
```

```
>> ./a.out
p=0xbfdb6374, &a[0][0]=0xbfdb6374
p2=0xbfdb6374, &a[0]=0xbfdb6374
p+1=0xbfdb6378, &a[0][1]=0xbfdb6378
p2+1=0xbfdb6380, &a[1]=0xbfdb6380
```

- Why are pointers typed differently depending on base type?
  - For compiler to perform accurate pointer arithmetic (or index ops)
  - For compiler to know type of dereferenced value. E.g.:
    int n, *p = …; float *q = …;
    n = *p // No conversion needed
    n = *q // Float->int conversion

The void* Type

- Mixing different pointer types 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 that can point to any base type
  - No casting needed when assigning to void* (vice versa)
  - Used when base type of a variable is unknown until later
  - Cannot be dereferenced / no pointer arithmetic
    - Size and type of variable pointed to not known
    - Needs to be cast to specific pointer type before dereferencing

The NULL Value

- Equivalent to the numerical value “0”. (Just like ‘\0’ is equivalent to “0”)
- NULL value means pointer points to nothing
- Tip: initialize all invalid pointers to NULL, instead of having them contain random addresses. Advantages:
  - Can easily compare to NULL to check if pointer is valid
  - If accessing invalid pointer by mistake
    - Will always 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;
  for (i = 0; i < argc; i++) {
    printf("argv[%d] = %s\n", i, argv[i]);
  }
  return 0;
}
```

```
>> ./a.out foo bar
argv[0] = ./a.out
argv[1] = foo
argv[2] = bar
```

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

Const Type Qualifier

- **Type qualifiers**: Keywords that qualifies how a type can be used
  - E.g. “const”, “volatile”, “restrict”
- **Const** type qualifier: disallows modification of variables
  - const float pi = 3.14;
    - “pi” is constant
  - char * const str = “Hello”;
    - “str” is constant (cannot point to another string)
  - const char *str = “Hello”;
    - Location pointed to by “str” is constant (content of string cannot be modified)
  - size_t strlen(const char *s);
    - Content of string pointed to by “s” cannot be modified inside “strlen”
- If modified -> compile time error (just like any other type error)
  - Type declarations specify what values can be stored in variables
  - Type qualifiers specify how those variables can be used
  - A “contract” programmer makes and compiler enforces

Example Use of Const

```
#include <string.h>

int main()
{
  char *str = "Hello";
  strcpy(str, “World”);
  return 0;
}
```

If you run this you get this:

```
>> gcc ./main.c
>> ./a.out
Segmentation fault (core dumped)
```

- “str” points to a string constant (immutable memory)
- strcpy(char *dst, const char *src) modifies dst
- Writing to immutable memory segment results in segmentation fault and crash

Example Use of Const

```
#include <string.h>

int main()
{
 const char *str = "Hello";
 strcpy(str, “World”);
 return 0;
}
```

If you run this you get this:

```
>> gcc ./main.c
./test.c:6:10: warning: passing 'const char *' to parameter of type 'char *' discards qualifiers
>> ./a.out
Segmentation fault (core dumped)
```

- Contract: String pointed to by “str” cannot be modified
- Copying “str” to first argument of “strcpy(char *dst, const char *src)” can potentially lead to violation of contract
- Program still compiles (because compiler is not sure strcpy will actually modify “str”) but gives off a warning

Example Use of Const

```
#include <string.h>

int main()
{
 const char *str = "Hello";
 str[0] = “W”;
 return 0;
}
```

If you run this you get this:

```
>> gcc ./main.c
./test.c:6:10: error: read-only variable is not assignable
```

- Now compiler emits error instead warning since it is certain of the violation with str[0] = “W”;
- Executable binary is not generated

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
    ➔ This process is called **linkage**
- 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 scope

Lexical Scope Example

```
int global;
static int internal;
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 internal": 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 (since no static)
  - **extern**: storage class specifier declaring the variable is defined in another C source file
    - Does not define a new variable
  - In foo.c, declaring "extern int global" tells compiler "global" refers to a variable defined elsewhere
- Also applies to function declarations

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);
}
```

```
>> ./a.out
First: n=10
Second: n=5
Third: n=10
```

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**: when a variable in an inner scope "hides" a variable in an outer scope
- Function scope variable "int n = 5" shadows external linkage scope variable "int n = 10"
- Prevents local changes from inadvertently spilling over to global state
  - Whoever writes "foo()" does not need non-local knowledge of global variables with same name
  - Important for modular programming
- Do not over use shadowing
  - Reduces readability (have to think of scoping rules)
  - Prone to mistakes when renaming local variables (if you miss a few, it will still compile but refer to outer scope variables)

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 lexical or compile time property)
- C defines three types of lifetimes
  - **Automatic**: automatically created and destroyed at scope begin and end, by code generated by compiler
  - **Static**: created at program initialization and never destroyed
    (No relations to "static" storage class specifier for internal linkage scope)
  - **Manual**: manually created and destroyed by the programmer on the heap
    (Will discuss this later in lecture)
- Allows efficient management of memory by compiler
  - Avoid static lifetimes 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

	Block	Function	Internal Linkage	External Linkage
Automatic	local	local	N/A	N/A
Static	static local	static local	static global	global

- local: Visible only within curly braces (block or function) and valid only while executing code inside curly braces
- static global: Visible within file and valid for entire duration
- global: Visible globally and valid for entire duration
- automatic global? N/A since global variables need to be always valid
- static local: Visible only within curly braces but valid for entire duration
  - Is this storage class really useful?

Static Local Use 1: Returning Local Array

```
char *asctime(const struct tm *timeptr) {
  static char result [26];
  …
  sprintf(result, …);
  return result;
}
```

- Is memory reserved for char array “result” valid after function returns?
- Local strings in a function cannot be returned per se
  - Value of character array is just a pointer to the first element
  - Local character array gets deallocated on function return ➔ dangling pointer
- **Static** storage class specifier on “result” makes array **static local**
  - Allows array to live beyond function return
  - Static specifier has different meanings when used on global or local variables!

Static Local Use 2: Keeping Track of Internal State

```
void foo() {
  static int count = 0; // only initialized at program startup
  printf("Foo called %d time(s).\n", ++count);
}
```

- "count" accessed only in foo()
  -> should be **local** scope (for encapsulation)
- "count" must be kept track of across calls to foo()
  -> should have **static** lifetime
- "count" will be initialized to 0 at beginning of program and then retain its value across calls to foo()

Static Local Use 2: Keeping Track of Internal State

```
int main() {
 char str[20], *tok;
 strcpy(str, "Blue,White,Red");
 tok = strtok(str, ",");
 while(tok != NULL) {
  printf("token: %s\n", tok);
  tok = strtok(NULL, ",");
 }
 return 0;
}
```

```
>> ./a.out
token: Blue
token: Red
token: White
```

- char *strtok(char *str, const char *delim): C library function that parses “str” into a sequence of tokens using “delim” as delimiter
  - First call to strtok (with str argument): return first token for “str”
  - Subsequent calls to strtok (with NULL argument): return token that comes next
  - Need internal state to keep track of where we are in the string
- In strtok, current location is kept across calls using a static local pointer

Example of Wrong Storage Class

```c
#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;
}
```

```
>> gcc ./main.c
./main.c: In function ‘foo’:
./main.c:4: warning: function returns address of local variable
>> ./a.out
*p=5
*p=10
```

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;
}
```

- What happened?

1. When foo returns, it returns a pointer to the deallocated memory location for “x”
2. When “*p” is printed for the first time, it accesses the deallocated location but the location has not been reused yet, so it prints the correct value 5
3. When function bar() is called, variable “int y” is allocated to the same location as “int x” that has been deallocated, overwriting the value with 10
4. When *p is printed for the second time, it prints the overwritten value 10

- Why “int x” and “int y” end up in the same location will become clear when we talk about how memory is managed for local variables by the compiler

Fix Using Static Local Storage Class

```c
#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;
}
```

```
>> gcc ./main.c
>> ./a.out
*p=5
*p=5
```