Pointers and Lexical Scoping

CS449 Fall 2017
Review: Pointers

- **Pointer**: Data type for an address
- **Reference operator**: e.g. “int *p = &x;”
  - Assign address of “x” to pointer p
- **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: Pointers, Arrays, and Strings

```
#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("strlen(str)=%u, strlen(buf)=%u, strlen("Hello")=%u\n", strlen(str), strlen(buf), strlen("Hello"));
    printf("sizeof(str)=%u, sizeof(buf)=%u, sizeof("Hello")=%u\n", sizeof(str), sizeof(buf), sizeof("Hello"));
    return 0;
}
```

```
>> ./a.out
str=Hello, buf=Hello
strlen(str)=5, strlen(buf)=5, strlen("Hello")=5
sizeof(str)=8, sizeof(buf)=20, sizeof("Hello")=6
```
Pointer Arithmetic

- A subset of arithmetic operators that work on pointers
- Example: meaning of “p + N”

```c
#include <stdio.h>
int main() {
    int *p = (int *) 0x1000;
    printf("p=%p
p+2=%p\n", p, p+2);
    return 0;
}
```

- `p + N` == Pointer to memory location N offsets away from p
  - Address calculation: `p + sizeof(int) * N`
- Why use “p + N” rather than “&p[N]”?  
  - They mean exactly the same thing  
  - Just like the [] operator, pointer arithmetic is meant to work with pointers to elements in an array  
  - But often more concise and intuitive to use pointer arithmetic

```
>> ./a.out
p=0x1000
p+2=0x1008
```
List of Pointer Arithmetic Operations

• \( p + N \)
  – Result: Pointer to location \( N \) offsets away from \( p \)
  – E.g. “\( p = p + 1; \)”, “\( p += 1; \)”, “\( ++p; \)”

• \( p - N \) (Same, except now offset is negative)

• \( p1 - p2 \)
  – Result: Integer offset between \( p1 \) and \( p2 \)
  – E.g. “\( \text{int offset} = p1 - p2; \)” (offset between the two array elements)
  – \( p1 \) and \( p2 \) must be of same pointer type

• \( p1 \text{ COMPARE } p2 \) (where COMPARE is one of ==, !=, <, >, <=, >=)
  – Result: Numerical comparison between \( p1 \) and \( p2 \)
  – E.g. “\( p1 < p2 \)” (whether \( p1 \) comes before \( p2 \) in array)

• Why not allow other operations (e.g. “\( p1 + p2 \)” or “\( p * N \)”)?
  – Would they compute any meaningful values?
Strcpy Using Pointer Arithmetic

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

- At each iteration of loop, the below happens (in order)
  1.  *p = *src         // Copy char at *src to location *p
  2.  The char is checked against 0 (or ‘\0’) to decide whether to stop
      (Remember: return value of = operator is the assigned value)
  3.  p++ and src++      // Advance p and src one char
      // Remember: post-increment happens last

- Stops when *src == ‘\0’ (when end of string is reached)
#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","p2" point to the same location
  – p == &a[0][0]
  – p2 == &a[0]

• "p+1","p2+1" point to different locations
  – p+1 == &a[0][1] (base + sizeof(int))
  – p2+1 == &a[1] (base + sizeof(int[3]))

• Types allow compiler to translate the meaning of "p+1" correctly
# Types Allow Correct Pointer Arithmetic

```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;
}
```

```
>> ./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.:
    ```c
    int n, *p = ...;
    float *q = ...;
    n = *p  // No conversion needed
    n = *q  // Float->int conversion
    ```
The void* Type

• Mixing different pointer types results in compiler warning
  – E.g. “int *p; char *p2 = p;” results in compiler warning
  – Generally, using int * to point to char location is a bad idea

• Except when assigning to and from void* type
  – E.g. “int *p; void *p2 = p;” is perfectly fine
  – E.g. “void *p; int *p2 = p;” is also perfectly fine

• Void pointer (void *)
  – Generic pointer that has no base type
  – Cannot be dereferenced / no pointer arithmetic
    • Size and type of location pointed to not known
    • All it can do is point to a memory location
  – Then what is it useful for?
    • To point to a piece of memory that you have not yet decide the use for
    • When you decide, you can assign it to a concrete pointer with a base type
    • Will later see uses when we talk about dynamic memory allocation
The NULL Value

• Equivalent to the numerical value 0 (Just like ‘\0’ is equivalent to 0)
• NULL value means pointer points to nothing
• Good practice to initialize all pointers to NULL, instead of leaving them dangling. Why?
  – Easy to check if a pointer is valid
    • If pointer != NULL, then pointer is valid
    • No easy way to check whether a pointer is dangling
  – If accessing NULL pointer by mistake
    • Will always result in an Immediate segmentation fault
    • Instead of accessing and corrupting some random memory (Results in wrong output or delayed crash – hard to debug)
#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;
}

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

>> ./a.out foo bar
argv[0] = ./a.out
argv[1] = foo
argv[2] = bar
Const Type Qualifier

- **Type qualifiers**: Keywords that qualifies how a variable can be used
  - E.g. “const”, “volatile”, etc.
- **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 compiler detects modification => compile error or warning
  - A “contract” programmer makes with herself that compiler enforces
  - Serves as a form of documentation on the use of variables
  - Unlike mere comments, compiler verifies documentation is correct!
Example Use of Const 1

```c
#include <string.h>

int main()
{
    char *str = "Hello";
    /* str is a STRING CONSTANT */
    str[0] = "Y";
    return 0;
}
```

• “str” points to a string constant (immutable memory)
• `str[0] = "Y";` modifies string pointed to by “str”
• No amount of all caps commenting will prevent someone else (or yourself) from writing this code

If you run this you get this:

```
> gcc ./main.c
> ./a.out
Segmentation fault (core dumped)
```
Example Use of Const 1

#include <string.h>

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

If you run this you get this:

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

- Contract: String pointed to by “str” cannot be modified
- Compiler emits error since str[0] = “Y”; is a violation
- Attaching const type qualifier guarantees that there are no errors in program with respect to modifying “str”
  - If the program compiles correctly
Example Use of Const 2

```c
#include <string.h>

int main()
{
    char *str = "Hello";
    /* str is a STRING CONSTANT */
    strcpy(str, "World");
    return 0;
}
```

If you run this you get this:

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

• This time string constant pointed to by “str” is modified by a `strcpy()` call

• Same issue as last example causes crash
Example Use of Const 2

```c
#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 for `strcpy(char *dst, const char *src)`:
  - String pointed to by “dst” can be modified but “src” cannot be
- Passing “str” to “dst” can potentially lead to a violation of contract
  - Now “dst” and “str” point to the same string
  - But “dst” does not have const type qualifier and may modify string
- Program still compiles but gives off a warning
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 use of a symbol to its corresponding symbol definition using scoping rules
    ➔ This process is called **linkage**

• C defines four types of scopes
  – Block scope: within code block in curly braces
  – 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

```c
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 int global”: Allows use of “global” in foo.c**
  - `extern`: storage class specifier for **declaring** the variable is defined in another C source file
    - **Does not define** a new variable
  - Definition: allocating something in memory
  - Declaration: telling compiler something exists
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
#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);
}
Lifejme

• **Lifetime**: duration of time where a variable is live
  – Time between the allocation and deallocation of a variable in memory
  – Is a runtime property (unlike scopes, a lexical or compile time property)

• C defines three types of lifetimes
  – **Automatic**: variables that are automatically created / destroyed at scope begin / end, by allocation / deallocation code generated by compiler
  – **Static**: variables that are live for entire duration of program
    (No relation to “static” storage class specifier for internal linkage scope)
  – **Manual**: memory manually created and destroyed by the programmer
    (Will discuss this later)

• Limited lifetimes allow conservation of memory
  – Automatic / manual lifetimes 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()”
  – Automatic variables are not initialized to 0
    (Automatic variables may need to be created / destroyed multiple times and initializing to 0 each time is too costly)
Storage Classes

• Storage class: combination of variable scope and lifetime

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

• local: Visible only within curly braces (block or function) and live only while executing code inside curly braces
• static global: Visible within file and always live
• global: Visible across files and always live
• automatic global? Impossible, since global variables need to be always live
• static local: Visible only within curly braces but always live
  – Is this storage class really useful?
Static Local Use 1: Returning Local Array

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

• Is char array “result” live after function returns?
• Local arrays in a function cannot be returned
  – Value of “result” is just a char pointer
  – Local 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
• Note: Static has different meanings when used on global or local variables!
  – Static local: refers to the lifetime of the variable (always live)
  – Static global: refers to the scope of the variable (internal linkage)
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()
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;
}

• char *strtok(char *str, const char *delim): C Standard 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: Lifetime of Variables

```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: Lifetime of Variables

```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;
}
```

- 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: Use 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
```