MPCS 51040 – C Programming

Lecture 4 – Pointers & Recursion

Dries Kimpe

April 17, 2017

Announcements

- Reminder: quiz next week.

Pointers

Pointer

A pointer is any variable which holds (as value) a memory address. Like all variables, pointer variables have a static compile-time data type and a runtime value.

```c
// Value = 0
// Type = integer
int myvariable = 0;

// Value = address of a
// Type = pointer to int
int * myvariable = &a;

// Change value of myvariable
*myvariable=10;
```

- Like all variables, pointer variables have a value (an address) and a type.
- Addresses are always the same size (for a given platform and compiler), so all pointer variables (*even of different type*) are the same size.
- Pointer variables with different types are compatible (can be assigned to one another)
  - However, this is dangerous and should be avoided!
    Note the warnings with `-Wall`!

`pointer-type.c`: address-of, dereference, pointers and struct/union, dangling pointer, nullpointer.

Pointers

Pointer

A pointer is any variable which holds (as value) a memory address. Like all variables, pointer variables have a static compile-time data type and a runtime value.

```c
// Value = 0
// Type = integer
int myvariable = 0;

// Value = address of a
// Type = pointer to int
int * myvariable = &a;

// Change value of myvariable
*myvariable=10;
```

- operator & returns the address of a variable (and the type of the expression will be pointer-to-type-of-variable)
- operator * (dereference) takes a pointer-to-sometype and returns an *lvalue* of type 'sometype'.
- operator -> is shorthand for * followed by . (member selector).

`pointer-type.c`: address-of, dereference, pointers and struct/union, dangling pointer, nullpointer.

Pointer Arithmetic

Operations on the pointer *datatype*

Pointers form a family of datatypes (i.e. **int** $*$, **int** $**$, **char** $*$, . . .)
All members of this family support a common set of operations (just like all integer-types support + and −.

```
// a now points to someint
int * a = &someint;

// b now points to the
// address FOLLOWING
// someint (no matter if there
// is an integer
// there or not)
int * b = a+1;
```

- Pointers support addition, subtraction, increment, decrement.
- Pointer arithmetic is in multiples of the type pointed to.
  - ++a increments the value (address) by **sizeof**(∗a), not by 1!
  - a−b (with a and b pointers) return the difference between the addresses of a and b in units of sizeof(*a)

Demonstrate pointer + pointer, pointer - pointer, pointer + int, pointer - int, pointer int, . . .

void pointers

```c
int b;
void * a = &b;
*a = 10; // illegal
int * c = a; // no cast!
double * p;
*c = 10; // OK
*(int *)a = 10; // OK

p = c; // warning
a = c; // no warning
p = a; // no warning
```

- **void** pointers cannot be dereferenced (**void** is 'no type')
- To dereference them, provide type information by
  - Assigning them to non-void pointer.
  - *casting* (i.e. override deduced type) to a typed pointer.

`void` pointers (i.e. `void *`) are compatible with all other pointers. If `p` is a pointer (any pointer type), and `q` is a `void *` pointer, then `q = p` and `p = q` are both valid and typically does not issue a warning... This is *NOT* true in C++, where `q = p` is accepted, but `p = q` is an error...

Pointers and Arrays

```c
char test[] = "test";
somefunction(test); // decay
test[0]='a';        // decay

// no decay; is somefunc(5);
somefunc(sizeof(test));

// decay into char *
somefunc(test);
```

- Arrays are not really a first-class data type in C
- Arrays almost always 'decay' into pointers; Exceptions:
  - Argument of operator &
  - Argument of operator **sizeof**
  - Argument of operator alignof (C11)
  - Use as string literal

In particular, they decay when used in an expression (other than above) or when passed to a function.

NOTE

Even the subscript operator ([]) is not what it seems:
a[2] => *(a+2)

Be careful with `sizeof`!

pointer-decay.c

Function Pointers

Pointers to functions are possible:

```c
int test(int i);
int (*alsotest)(int i) = test;

test(2);
alsotest(2);
```

- For functions, & and $*$ are optional. (clear from context)
- Pointer arithmetic is not allowed on function pointers.
  (some compilers allow as extension)
  - ++ will *not* move to the next line of code!

In normal usage, operations on function pointers are restricted to setting to the address of a function, assigning to another function pointer, setting to NULL or dereferencing (calling).

sorting: `qsort.c`

Pointers: recap

A pointer p is:

- A variable
- which has as **type** a pointer (to some type)
- which (like all other variables) has a *static* type
  (i.e. a pointer to a char will always be a pointer to a char)
- which holds as **value** a memory address
- which (like any variable) itself has a memory address
- which can be *dereferenced* (using $*$) to obtain an *lvalue* (i.e. something which can appear on the left of '$=$')
- when dereferenced, an lvalue of type T (for pointers of type T $*$)
- when dereferenced, an lvalue for which, &($*$p) $==$ p
- capable of arithmetic (unless function pointer): add, subtract integer, increment, decrement, subtract two pointers to same type.
  Beware: arithmetic in base types, not in bytes!

Decoding Declarations

Right-left rule

- $*$ → Pointer to
- [] → Array of
- () → function returning

Rule

1. Find identifier
2. Go right (until no more symbols or until parenthesis)
3. Go left (until no more symbols or until left parenthesis)

Example

```c
// array of int
int a[10];
// function returning pointer
int * f();
// pointer to array of array of ints
int (*a)[][];
```

```c
// function ret pointer to array of int
int (*f())[];
// typedef for function pointer
typedef int(*TypeAlias)(int);
// function returning pointer to function
// returning int
int (*f())();
```

What is: **void** (*signal (**int** sig, **void** (*func)(**int**)))(**int**);
(help? `http://cdecl.org`)

Function: overview

```c
// function prototype(declaration)
int testfunc ();

// function definition
int testfunc ()
{
    // declarations and
    // statements
}

// Function taking 1 argument
// returning nothing
void returnnothing (int arg);
```

- A function starts a new scope: variables and declarations are local to the function.
- The scope of a variable in a function is until the end of the function. Same for function parameters.
- The (default) lifetime of a variable in a function starts from the moment it is defined until the function returns (same for function parameters).
- Each function invocation has its own set of local variables (allows recursion).
- All functions share the same namespace (can't have two functions with the same name *in the same program*)

Functions

Parameter Passing

There are multiple ways in which a parameter can be passed to a function, and different programming languages use different methods. Mostly, there are two approaches:

Pass-by-value or pass-by-reference?

pass-by-value The value of the variable is copied to the function; The function operates on a *copy* of the variable.

pass-by-reference The variable in the function is an *alias* for the variable passed to the function. The function directly operates on the passed variable.

pass-by-object-reference (java, python) Pass-by-value but objects are references (which are passed by value).

In C, **ALL** function calls are **pass-by-value**.

Returning Values

```
// Grammar:
// return <expression> ;
//
int myfunc ()
{
    // return takes int
    return 10;
}

void myfunc2 ()
{
    return;
}
```

- Functions can return any datatype (including structs) but not arrays.
- The return value is *copied*; Be careful with returning large structures.
- **return** optionally returns a value and always returns to the caller immediately (further statements are skipped).
- **return** is optional in functions returning void.

Emulating Pass-By-Reference

```c
void pass_by_value(int p)
{
    p=0;
}
void example(int * p)
{
    *p=0;
}

// pass/call by value
int a = 1;
pass_by_value(a);
// a is STILL 1!
printf("%i\n", a);

// emulate pass by reference
// a==1
example(&a);
// prints 0
printf("%i\n", a);
```

Question

Since function parameters are always pass-by-value, how do we *modify* a variable from within the function?

Solution: pass a *pointer* to the variable.

- The pointer is passed by value (and thus copied)
- The function can dereference the pointer to modify the variable.

Scope and lifetime revisited

```c
// - file scope
// - program lifetime
// - program visibility
int global = 10;

// global variable
// internal linkage
// - file scope
// - program lifetime
// - file visibility
static int private = 2;

int func() {
  // local variable
  // (automatic storage duration)
  //    - block scope
  //    - block lifetime
  int local = 2;
}

int func2() {
  // local variable
  // (static storage duration)
  // - block scope
  // - program lifetime
  static int local = 2;
}
```

Scope and lifetime:

- Scope of a variable is where the name is accessible.
- Lifetime is the duration for which the variable has a storage location (in memory) associated with it, i.e. how long it can hold (remember) a value.

So far, we have not explicitly specified anything for scope or lifetime. We have used:

- *local variables*: scope local to the statement block.
- *global variables*: file scope (starts at point where declared until the end of the (preprocessed!) file).

Regarding lifetime:

- *local* variables by default have *automatic storage duration*: they are destroyed when they go out of scope (end of block) and recreated when re-entered. They *do not hold their value when destroyed.*
- *static* storage duration means that the lifetime is *until the program ends*. Global variables (outside of any function) have static storage.

For global variables, adding the **static** keyword

Scope and lifetime revisited

```c
// - file scope
// - program lifetime
// - program visibility
int global = 10;

// global variable
// internal linkage
// - file scope
// - program lifetime
// - file visibility
static int private = 2;

int func() {
  // local variable
  // (automatic storage duration)
  //    - block scope
  //    - block lifetime
  int local = 2;
}

int func2() {
  // local variable
  // (static storage duration)
  // - block scope
  // - program lifetime
  static int local = 2;
}
```

Scope and lifetime:

- Scope of a variable is where the name is accessible.
- Lifetime is the duration for which the variable has a storage location (in memory) associated with it, i.e. how long it can hold (remember) a value.

So far, we have not explicitly specified anything for scope or lifetime. We have used:

- *local variables*: scope local to the statement block.
- *global variables*: file scope (starts at point where declared until the end of the (preprocessed!) file).

Regarding lifetime:

- *local* variables by default have *automatic storage duration*: they are destroyed when they go out of scope (end of block) and recreated when re-entered. They *do not hold their value when destroyed.*
- *static* storage duration means that the lifetime is *until the program ends*. Global variables (outside of any function) have static storage.

For global variables, adding the **static** keyword

Dynamic Memory Allocation

Why do we need dynamic memory allocation

Pointers

Pointers have a number of benefits:

- Allow 'aliasing' of variables
- Allow easy traversing of arrays and other data structures
- Provide access to variables outside of the scope their normal scope.

They do not:

- change the lifetime of existing variables
- enable 'variable length' variables

Remaining issues; examples:

- Read a variable length line from a file
- Create a variable in a function and pass it to the caller
  (i.e. lifetime not bound to scope)

Dynamic Memory Allocation

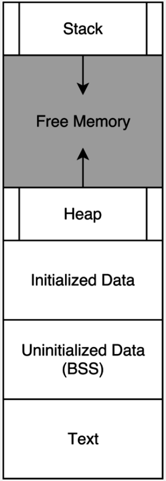
malloc, free

```c
// request memory
char * ptr = (char *)
              malloc(100);

// Use ptr as usual
copyAndConvert("test",
               ptr, 100);

// free memory
free(ptr);
```

- malloc does not know what you want to store in the requested memory (i.e. type) so it returns **void** *. You will need to cast.
- There is no garbage collection; You are responsible for freeing resources.
- There are no checks to ensure you stay within your allocated memory.
- You should not try to pass an invalid pointer to free or not try to free the pointer more than once. (undefined behaviour will result)
- The returned memory region is *not initialized*. It may contain random bytes. (if you need to clear it, use calloc)

firstmalloc.c: malloc, free examples.

Memory Regions
(Implementation Detail)



Typical memory layout:

- automatic variables go on the stack.
- uninitialized global and static storage duration variables go into BSS.
- Initialized global and static storage duration variables go into data.
- malloc allocates memory from the heap.

This is an implementation detail, and not required by the standard. The standard only specifies variable lifetime etc., not how to map these concepts to memory.

Walk through stack changes on function call.

Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Associativity defines the order in which adjacent operators *with the same precedence level* are evaluated.

- a−b−c => (a−b)−c
- q && r || s => (q && r) || s

Does not say in which order the operator *operands* have to be evaluated!

precedence.c

Evaluation Order

```
// Unspecified function
// evaluation order;
// Associativity only
// defines addition order.
f1()+f2()+f3();

// bad; no sequence point
a[i]=b[i++];

// bad; no sequence point
// however, all side effects
// done *before* entering
// function
f(i++,i,j++);

// OK; , is seq point
i++,a=i;
```

Order of operations of the operands of *almost all* operators is not defined by the language. The compiler is free to evaluate operands in any order (and does not have to be consistent).

Only the sequential-evaluation (,), logical-AND (&&), logical-OR (||), conditional-expression (? :), and function-call operators constitute **sequence points** and therefore guarantee a particular order of evaluation for their operands.

(sequence point: all side effects of previous evaluations have occurred)

The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma operator in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.)

Boolean Short-circuit Evaluation

```c
// call to f never happens
0 && f();

// OK but avoid due to short circuit
// with side-effects;
// logical operator is seq point
q && r || s--;
```

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. This is called "short-circuit" evaluation. Thus, some operands of the expression may not be evaluated.

Expert topic; Better to avoid relying on specific sequencing whenever possible.

Recursion

```c
void repeat(unsigned int i)
{
    if (!i)
        return;
    // do something
    puts("X");
    repeat(i-1);
}
```

Recursion:

- Recursion happens when a function calls itself.
- Each invocation of the function receives its own copy of local (automatic) variables (variables go on the stack).
- Local variables exists until the function returns.
  - Watch out for memory (stack) usage!
- Recursion is a form of looping.

Recursion needs to end: *base case*.

Calculate fibonacci numbers: $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0, F_1 = 1$

Solution

The base case and recursion step are explicit in the mathematical definition.

Calculate fibonacci numbers

Recursion
Example

Recursion is very useful for divide-and-conquer type algorithms: split the problem in smaller problems and try to solve the smaller problem.

Example

Count the number of occurrences of a number in an array.

Solution

- *Base case (conquer): array of size 1*
- *Divide: split in two arrays, add counts.*

```
unsigned int count (int * array, unsigned int size);
```

Implement count().

Valgrind

Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

- Valgrind can help debug pointer issues
  (null pointer usage, accessing memory outside of allocated memory region, using memory that has been freed, memory leaks, . . .)
- Homework 3 will use valgrind to validate your code.
- Valgrind is available on linux.cs.uchicago.edu.
- **Make sure to compile your code with debug information enabled! (`-g` option for gcc)**

demo memory leak, out-of-bounds, use-after-free

- General documentation: `http://valgrind.org/`
- Memory-debugging: `http://valgrind.org/docs/manual/quick-start.html#quick-start.mcrun`

Valgrind Example

Example

```
LEAK SUMMARY:
    definitely lost: 0 bytes in 0 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 0 bytes in 0 blocks
    still reachable: 0 bytes in 0 blocks
         suppressed: 88 bytes in 1 blocks
```

- Useful options for memory debugging:
  - `--show-reachable=yes`
  - `--leak-check=full`
- Valgrind cannot detect all errors or mistakes!
- Normally you can ignore 'suppressed' errors or leaks.

See `incorrect_program.c` in lecture materials

GDB

GDB

GDB is a command-line debugger (what's a debugger?)

- GDB is installed on linux.cs.uchicago.edu.
- GDB allows us to run the program step-by-step and to inspect the value of variables.
- GDB can do post-mortem analysis (i.e. on a coredump file).

NOTE

Add –ggdb to your compile commandline to instruct the compiler to include more information. Adding –O0 is also recommended.

setting breakpoints, inspecting variables, use a coredump.

- Easy tutorial: `http://www.techbeamers.com/how-to-use-gdb-top-debugging-tips/`
- `https://www.gnu.org/software/gdb/documentation`

GDB

Core dumps

```
dries@linux2:~$ ulimit -c
0
dries@linux2:~$ ./incorrect_program
Segmentation fault
dries@linux2:~$ ulimit -c unlimited
dries@linux2:~$ ./incorrect_program
Segmentation fault (core dumped)
dries@linux2:~$ ls -l core
-rw------- 1 dries dries 253952 ...
dries@linux2:~$
```

- A core dump is a snapshot of the memory state of the program at the moment an unrecoverable error occurs
- It can be used afterwards to observe the state of the program (variables, which function, etc.)
- Normally, code dumps are disabled. Enable them using the `ulimit` command.

check & change `ulimit`, cause core, open gdb with core (`incorrect_program.c`)

Value Types

Value Types and Reference Types

In general, there are two kinds of types (not meaning 'C datatype' here but logical type): *value* type and *reference* type.

```c
// Example value type
int a = 10;
b = a;
++b; // b != a

// Example reference type
FILE * f = fopen(...);
FILE * f2 = f;
fgetc(f);
// f2 is modified as well!
```

- value types (example: POD types in C++) are fully contained in the memory the underlying (C) type holds. They do not refer to outside data.
- Because of this, value types can be created, copied and 'destroyed' without special considerations. (Compare with FILE * which needs to be properly closed)
- The underlying C type for a reference type does not hold (or only partially holds) the state for the logical type. It cannot be copied by assignment (since this would not copy the external state).

Modular Code

Example

Good code can easily be reused. An often overlooked property is that ideally, changes in the implementation details of the code should not affect users of that code.

```c
// Library for employee management
// header
struct Person
{
    int age;
};

// User of library
struct Person p;
// bad!
printf ("%i", p.age);
```

This is not good reusable code.

- Can never change struct Person (for example store age elsewhere)
- (Example) Can not add access control or logging.

Person: Improved

```c
// in header:
struct Person { int age; }
void setAge (Person * p, int age);

// in main
struct Person p;
setAge(&p, 21);
```

Possible issues:

- Can look in header and access age directly.
- Can assume Person to be value type.

Modular Code

Better: Encapsulation

```
// In java:
//     p.setAge(10);

// Update age
void setAge (struct Person * p, int a);

// Better
void setAge (Person * p, int age);

// Even better
void setAge (Person p, int age);
```

- Object-oriented principle: encapsulation
  - Information hiding
  - Only provide information needed (can't be 'abused')
- C does not support C++/Java syntax
  - but we can emulate it

Homework 3 - First Impressions

Some questions/remarks:

- Why do we need maxchar?
  - Think about somebody else using your code, and given only your header file.
  - Explicit checks/safety is better than documentation/assumptions.
  - Same reason why gets almost impossible to use safely.
- Be very careful with array lenghts/indices!

```
...
while (i<maxchar && ...)
{
}
dest[i]=0;
...
```

- Avoid specifying explicit sizes!
- Think 0-based!

- Maximum line length. Those days are over! You should be able to handle any input provided the system has enough memory.
- Array decay and consequences for `operator =`

```
char a[] = "1234";
char * b;
b=a; // does *not* make a copy of a!
```

Assignment: Homework 4 & Reading

Homework Assignment

See `https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/raw/master/homework/hw4/hw4.pdf`

Reading Assignment

O'Reilly Mastering Algorithms in C:

Required Chapter 4, 5

Recommended Chapter 1-3 (should mostly be refresh)

Discuss 'handle' types.

Quiz next week

Topics:

- Lectures 1–4
- Homework 1–4

Quiz Format

- Duration: max 1.5h
- Closed book / no laptop, phone, . . .
- Mostly pratical focus:
  - What is the output of this code?
  - Is this code correct?
  - Write small functions/programs.
  - Some terms and definitions.

Homework is excellent preparation!