Lecture Notes on
Memory Management

15-122: Principles of Imperative Computation
Frank Pfenning, Rob Simmons

Lecture 19
October 30, 2014

1 Introduction

In this lecture we will start the transition from C0 to C. In some ways, the lecture is therefore about knowledge rather than principles, a return to the emphasis on programming that we had earlier in the semester. In future lectures, we will explore some deeper issues in the context of C, but this lecture is full of cautionary tales.

The main theme of this lecture is the way C manages memory. Unlike C0 and other modern languages like Java, C#, or ML, C requires programs to explicitly manage their memory. Allocation is relatively straightforward, like in C0, requiring only that we correctly calculate the size of allocated memory. Deallocating ("freeing") memory, however, is difficult and error-prone, even for experienced C programmers. Mistakes can either lead to attempts to access memory that has already been deallocated, in which case the result is undefined and may be catastrophic, or it can lead the running program to hold on to memory no longer in use, which may slow it down and eventually crash it when it runs out of memory. The second category is a so-called *memory leak*.

2 The C0 and C Memory Model

Ultimately, "all" data resides in memory. In fact, part of the data may also be kept in fast registers directly on the CPU. You will learn about registers in detail in 15-213 and can learn about their use in programming languages in 15-411. For the purposes of today's lecture, it is sufficient to pretend all data would sit in memory and ignore registers for the time being. This simplifies the principles without losing too much precision.

The data in memory is addressed by memory addresses that fit to the addressing of the CPU in your computer. We will just pretend 32bit addresses, because those are shorter to write down. All addresses are positive, so the lowest address is 0x00000000 and the highest address $2^{32} - 1 =$ `0xFFFFFFFF`. All data (with the caveat about registers) sits in memory at some address. One important question about all data in memory is how big it is, so that the compiler can make sure program data is stored without accidental overlapping regions.

The basic memory layout looks as follows:

```
OS AREA
============
System stack   (local variables and function calls)
============
unused
============
System heap    (data allocated here... alloc or alloc_array)
============
.text (read only)  (program instructions sit in memory)
============
OS AREA
```

One consequence of this memory layout is that the stack grows towards the heap, and the heap usually grows towards the stack. The reason that the stack is called a stack is because it operates somewhat like the principle of the stack data structure. Your program can put new data on the top of the stack. It can also pop elements of the stack if this data is no longer necessary. Unlike the stack abstraction, it may appear as if your program internally also modifies data that is on the stack, even if it is not quite at the top of it. However, the only data on the stack that the program modifies is in the top range of the stack (perhaps the top 512 bytes or so, depending on the function that runs), even if it is not just the top word of the stack.

Programs cannot access memory cells that belong to the operating system. If they try, programs get an "exception" like a segmentation fault. Where can that happen in C0? C0 takes great care to ensure that it never gives you any pointers to uninitialized or random or garbage data in memory, *except*, of course, the `NULL` pointer. `NULL` is a special pointer to the memory address 0, which belongs to the operating system. Any access by a user-land program by dereferencing a NULL pointer causes a segfault.

If, however, you are writing a program that will be running as part of the operating system, your program has no protection against `NULL` pointer dereferencing anymore, because memory address 0 is a valid address for the operating system, even if not for regular programs. When writing code for operating systems, you, thus, need to have mastered the art of protecting against illegal `NULL` dereferences. This is exactly one of the things that contracts, loop invariants, and assertions prepare you for.

3 A First Look at C

Syntactically, C and C0 are very close. Philosophically, they diverge rather drastically. Underlying C0 are the principles of *memory safety* and *type safety*. A program is *memory safe* if it only reads from memory that has been properly allocated and initialized, and only writes to memory that has been properly allocated. A program is *type safe* if all data it manipulates have their declared types. In C0, *all* programs are type safe and memory safe. The compiler guarantees this through a combination of static (that is, compile-time) and dynamic (that is, run-time) checks. An example of a static check is the error issued by the compiler when trying to assign an integer to a variable declared to hold a pointer, such as

```
int* p = 37;
```

An example of a dynamic check is an array out-of-bounds error, which would try to access memory that has not been allocated by the program. Modern languages such as Java, ML, or Haskell are both type safe and memory safe.

In contrast, C is neither type safe nor memory safe. This means that the behavior of many operations in C is *undefined*. Unfortunately, undefined behavior in C may yield any result or have any effect, which means that the outcome of many programs is *unpredictable*. In many cases, even programs that are patently absurd will yield a consistent answer on a given machine with a given compiler, or perhaps even across different machines and different compilers. No amount of testing will catch the fact that such programs have bugs, but they may break when, say, the compiler is upgraded or details of the runtime system changes. Taken together, these design decisions make it very difficult to write correct programs in C. This fact is in evidence every day, when we download so-called *security critical* updates to operating systems, browsers, and other software. In many cases, the security critical flaws arise because an attacker can exploit behavior that is undefined, but predictable across some spectrum of implementations, in order to cause your machine to execute some arbitrary malicious code. You will learn in 15-213 *Computer Systems* exactly how such attacks work.

These difficulties are compounded by the fact that there are other parts of the C standard that are *implementation defined*. For example, the size of values of type `int` is explicitly not specified by the C standard, but each implementation must of course provide a size. This makes it very difficult to write *portable* programs. Even on one machine, the behavior of a program might differ from compiler to compiler. We will talk more about implementation defined behavior in the next lecture.

Despite all these problems, almost 40 years after its inception, C is still a significant language. For one, it is the origin of the object-oriented languages C++ and strongly influenced Java and C#. For another, much systems code such as operating systems, file systems, garbage collectors, or networking code are still written in C. Designing type-safe alternative languages for systems code is still an active area of research, including the Static OS project at Carnegie Mellon University.

4 Undefined Behavior in C

For today's lecture, there are three important undefined behaviors in C are:

Out-of-bounds array access: accessing outside the range of an allocated array has undefined results.

Null pointer dereference: dereferencing the null pointer has undefined results.

Double-free: We'll talk about this in Section 8.

Reading and writing uninitialized memory: In C0, it wasn't possible to ever read from initalized stack memory. In C, this is undefined behavior. It's also possible to create uninitialized memory on the heap in C, and reading from this is also undefined.

4.1 Arrays, pointers, and out-of-bounds access

When compared to C0, the most shocking difference is that C does not distinguish arrays from pointers. Array accesses are not checked at all, and out-of-bounds memory references (whose result is formally undefined) may lead to unpredictable results. For example, the code fragment

```
int main() {
  int* A = malloc(sizeof(int));
  A[0] = 0;    /* ok - A[0] is like *A */
  A[1] = 1;    /* error - not allocated */
  A[317] = 29; /* error - not allocated */
  A[-1] = 32;  /* error - not allocated(!) */
  printf("A[-1] = %d\n", A[-1]);
  return 0;
}
```

will not raise any compile time error or even warnings, even under the strictest settings. Here, the call to `malloc` allocates enough space for a single integer in memory. In this class, we are using `gcc` with the following flags:

```
gcc -Wall -Wextra -Werror -std=c99 -pedantic
```

which generates all warnings (`-Wall` and `-Wextra`), turns all warnings into errors (`-Werror`), and applies the C99 standard (`-std=c99`) pedantically (`-pedantic`). The code above executes ok, and in fact prints `32`, despite four blatant errors in the code.

To discover whether such errors may have occurred at runtime, we can use the `valgrind` tool.

```
% valgrind ./a.out
...
==nnnn== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

which produces useful error messages (elided above) and indeed, flags 4 error in code whose observable behavior was bug-free.

`valgrind` slows down execution, but if at all feasible you should test all your C code in the manner to uncover memory problems. For best error messages, you should pass the `-g` flag to `gcc` which preserves some correlation between binary and source code.

You can also guard memory accesses with approriate assert statements that abort the program when attempting out-of-bounds accesses.

Conflating pointers and arrays provides a hint on how to convert C0 programs to C. We need to convert `t[]` which indicates a C0 array with elements of type t to `t*` to indicate a pointer instead. In addition, the `alloc` and `alloc_array` calls need to be changed, or defined by appropriate macros (we'll talk about this more later).

4.2 Null pointer dereference

In C0, an out of bounds array access or null pointer dereference will *always* cause the program to print out `Segmentation fault` and exit aborting with (abort with signal `SIGSEGV`). In C, reading or writing to an array out of bounds *may* cause a segmentation fault, but it is impossible to rely on this behavior in practice.

In contrast, it is so common that dereferencing the null pointer will lead to a segmentation fault that it may be overlooked that this is not defined. Nevertheless, it is undefined, dereferencing `NULL` may not yield an exception, particularly if your code runs in kernel mode, as part of the operating system,

4.3 What actually happens?

If you do not get an error, then perhaps nothing at all will happen, and perhaps memory will become silently corrupted and cause unpredictable errors down the road. But we were able to describe, in each of the examples above, what sorts of things were *likely* to happen.

There's an old joke that whenever you encounter undefined behavior, your computer could decide to play *Happy Birthday* or it could catch on fire. This is less of a joke considering recent events:

- In 2010, Alex Halderman's team at the University of Michigan successfully hacked into Washington D.C.'s prototype online voting system, and caused its web page to play the University of Michigan fight song, "The Victors."[1]

- The Stuxnet worm caused centrifuges, such as those used for uranium enrichment in Iran, to malfunction, physically damaging the devices.[2]

Not quite playing *Happy Birthday* and catching on fire, but close enough.

5 Memory Allocation

Two important system-provided functions for allocating memory are `malloc` and `calloc`.

[1]Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. *Attacking the Washington, D.C. Internet Voting System*. Proceedings of the 16th Conference on Financial Cryptography and Data Security, February 2012.

[2]Holger Stark. *Stuxnet Virus Opens New Era of Cyber War*. Spiegel Online, August 8, 2011.

`malloc(sizeof(t))` allocates enough memory to hold a value of type t. In C0, we would have written `alloc(t)` instead. The difference is that `alloc(t)` has type `t*`, while `malloc(sizeof(t))` returns a special type `void*`, which we will discuss more in the next lecture. The important thing to realize is that C will not even check that the type we declared for the pointer matches the size we pass to `malloc`, so that while we can write this:

```
int* p = malloc(sizeof(int));
```

we can also write this:

```
int* p = malloc(sizeof(char));
```

which will generally have undefined results. Also, `malloc` does not guarantee that the memory it returns has been initialized, so the following code is an error:

```
int* p = malloc(sizeof(char));
printf("%d\n", *p);
```

Valgrind will report the error "`Use of uninitialised value of size 8`" if code with the above two lines is compiled and run.

`calloc(n, sizeof(t))` allocates enough memory for n objects of type t. Unlike `malloc`, it also guarantees that all memory cells are initialized with 0. For many types, this yields a default element, such as `false` for booleans, `0` for `ints`, `'\0'` for `char`, or `NULL` for pointers.

Both `malloc` and `calloc` may fail when there is not enough memory available. In that case, they just return `NULL`. This means any code calling these two functions should check that the return value is not `NULL` before proceeding. Because that makes it tedious and error-prone to write safe code, we have defined functions `xmalloc` and `xcalloc` which are just like `malloc` and `calloc`, respectively, but abort computation in case the operation fails. They are thereby guaranteed to return a pointer that is not `NULL`, if they return at all. These functions are in the file `xalloc.c`; their declarations are in `xalloc.h` (see Section 6 for an explanation of header files).

6 Header Files

To understand how the `xalloc` library works, and to take our C0 implementation of binary search treas and begin turning it into a C implementation, we will need to start by explaining the C convention of using *header files* to specify interfaces. Header files have the extension `.h` and contain type declarations and definitions as well as function prototypes and macros, but never code. Header files are not listed on the command line when the compiler is invoked, but included in C source files (files with the `.c` extension) using the `#include` preprocessor directive. The typical use is to `#include`[3] the header file both in the implementation of a data structure and all of its clients. In this way, we know both match the same interface.

This applies to standard libraries as well as user-written libraries. For example, the client of the C implementation of BSTs starts with

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "xalloc.h"
#include "contracts.h"
#include "bst.h"
#include "bst-client.h"
```

The form `#include <filename.h>` includes file `filename.h` which must be one of the system libraries provided by the suite of compilation tools (gcc, in our case). The second form `#include "filename.h"` looks for `filename.h` in the current source directory, so this is reserved for user files. The names of the standard libraries and the types and functions they provide can be found in the standard reference book *The C Programming Language, 2nd edition* by Brian Kernighan and Dennis Ritchie or in various places on the Internet.[4]

7 Macros

Macros are another extension of C that we left out from C0. We use macros to get some semblence of contracts in C0, which are defined in the header file `contracts.h`.

Macros are expanded by a preprocessor and the result is fed to the "regular" C compiler. When we do not want `REQUIRES` to be checked (which is the default, just as for `@requires`), there is a macro definition

```
#define REQUIRES(COND) ((void)0)
```

[3] when we say "include" in the rest of this lecture, we mean `#include`
[4] for example, http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

which can be found in the file `contracts.h`. The right-hand side of this definition, `((void)0)` is the number zero, cast to have type `void` which means it cannot be used as an argument to a function or operator; its result must be ignored. When the code is compiled with

```
gcc -DDEBUG ...
```

then it is defined instead as a regular `assert`:

```
#define REQUIRES(COND) assert(COND)
```

In this case, any use of `REQUIRES(e)` is expanded into `assert(e)` before the result is compiled into a runtime test.

The three macros, all of which behave identically are

```
REQUIRES(e);
ENSURES(e);
ASSERT(e);
```

although they are intended for different purposes, mirroring the `@requires`, `@ensures`, and `@assert` annotations of C0. `@loop_invariant` is missing, since there appears to be no good syntax to support loop invariants directly; we recommend you check them right after the exit test or at the end of the loop using the `ASSERT` macro.

Another common use for macros is to define compile-time constants. In general, it is considered good style to isolate "magic" numbers at the beginning of a file, for easy reference; for instance, if we were coding our E0 editor in C, it would make sense to

```
#define GAP_BUFFER_SIZE 16
```

to make it easy to change from size 16 gap buffers to some other size. The C implementation itself uses them as well, for example, `limits.h` defines `INT_MAX` as the maximal (signed) integer, and `INT_MIN` and the minimal signed integer, and similarly for `UINT_MAX` for unsigned integers.

7.1 Conditional compliation

Header guards `_BST_H_` are an example of *conditional compilation* which is often used in systems files in order to make header files and their implementation portable. Another idiomatic use of conditional compilation is

```
#ifdef DEBUG
...debugging statements...
#endif
```

where the variable `DEBUG` is usually set on the gcc command line with

```
gcc -DDEBUG ...
```

Guarding debugging statements in this way generalizes the simple assertion macros provided in `contracts.h`.

8 Freeing Memory

Unlike C0, C does not automatically manage memory. As a result, programs have to free the memory they allocate explicitly; otherwise, long-running programs or memory-intensive programs are likely to run out of space. For that, the C standard library provides the function `free`, declared with

```
void free(void* p);
```

The restrictions as to its proper use are

1. It is only called on pointers that were returned from `malloc` or `calloc`.[5]
2. After memory has been freed, it is no longer referenced by the program in any way.

Freeing memory counts as a final use, so the goals imply that you should not free memory twice. And, indeed, in C the behavior of freeing memory that has already been freed is undefined and may be exploited by an adversary. If these rules are violated, the result of the operations is undefined. The `valgrind` tool will catch dynamically occurring violations of these rules, but it cannot check statically if your code will respect these rules when executed.

The *golden rule of memory management* in C is

You allocate it, you free it!

By inference, if you *didn't* allocate it, you are *not* allowed to free it! But this rule is tricky in practice, because sometimes we do need to transfer ownership of allocated memory so that it "belongs" to a data structure.

[5]or `realloc`, which we have not discussed