Lecture 19 Notes
Data Structures in C

15-122: Principles of Imperative Computation (Summer 1 2015)
Rob Simmons

In this lecture, we will begin our transition to C. In many ways, the lecture is therefore about knowledge rather than principles, a return to the emphasis on programming that we had at the very beginning of the semester. In future lectures, we will explore some deeper issues in the context of C. Today’s lecture is designed to get you to the point where you can translate a simple C0/C1 program or library (one that doesn’t use arrays, which we’ll talk about in the next lecture) from C0/C1 to C. An important complement to this lecture is the “C for C0 programmers” tutorial:

http://c0.typesafety.net/tutorial/From-C0-to-C:-Basics.html

There are two big ideas you need to know about. First, C has a whole separate language wrapped around it, the C preprocessor language. The preprocessor language is big and complicated and can be used for a bunch of things that aren’t C: you only need to understand a couple of ways that it gets used:

- Macro constant definitions: you’ll need to know how these are used in the <limits.h> and <stdbool.h> libraries.

- Macro function definitions: you’ll need to know how these are used to implement "lib/contracts.h" library, and you’ll need to know why they’re generally a dangerous idea.

- Conditional compilation: you need to know how #ifdef and #ifndef are used, along with macro constant definitions, to make separate compilation of libraries work in C.

Second, C has a different notion of allocating memory than C0 did. In particular, C is not garbage collected, so whenever we allocate memory, we have to make sure that memory eventually gets freed.
Our discussion will center around translating a very simple C0 interface and implementation, and a little program that uses that interface.

1 Running example: a simple interface simple.c0

```
#use <util>

/*** Interface ***/
int absval(int x)
    //@requires x >= int_min(); @*
    //@ensures \result >= 0; @*
    ;

struct point2d {
    int x;
    int y;
};

/*** Implementation ***/
int absval(int x)
    //@requires x > int_min();
    //@ensures \result >= 0;
    {
        int res = x < 0 ? -x : x;
        return res;
    }
```

2 Running example: a simple test program: test.c0

```
#use <conio>

int main() {
    struct point2d* P = alloc(struct point2d);
    P->x = -15;
    P->y = P->y + absval(P->x * 2);
    assert(P->y > P->x && true);
    print("x coord: "); printint(P->x); println("\n");
    return 0;
}
```

We can compile this program by running: cc0 -d simple.c0 test.c0
3 Introducing the preprocessor language

In C0 programs, just about the only time we typed the ‘#’ key was to include a built-in library like conio by writing: #use <conio>. The C preprocessor language is built around different directives that start with ‘#’. The first two you need to know about are #include and #define.

The #include directive is what replaces #use in C0. Here are some common #include directives you’ll see in C programs:

```c
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
```

The <stdlib.h> library is related to C0’s <util> library, <stdio.h> is related to <conio> in C0, and <string.h> is related to <string> in C0.

The <stdbool.h> file is also important: the type bool and the constants true and false aren’t automatically included in C, so this library includes them. We’ll talk more about libraries, and in particular the .h extension, later.

4 Macro definitions

C0 has a very simple rule: an interface can describe types, structs, and functions. This leads to some weirdnesses, though: the C0 <util> library has to give you a function, int_max(), for referring to the maximum representable 32-bit two’s complement integer.

The #define macro gives you another way to write down what int_max() is in C0:

```c
#define INT_MAX 0x7FFFFFFF
```

A very weird but very important feature of the C preprocessor language is that this line means, WHENEVER the preprocessor sees INT_MAX later on in your program, it will just replace that token with the definition. The C compiler itself will never see INT_MAX, because the text INT_MAX will have been flat-out replaced by 0x7FFFFFFF by the time the compiler sees it.

This textual replacement must be done very carefully: for instance, this is a valid, if needlessly verbose, definition of INT_MIN:

```c
#define INT_MIN -1 ^ 0x7FFFFFFF
```
Then imagine that later in the program we wrote \( \text{INT\_MIN} / 256 \), which ought to be equal to \( -2^{31}/2^8 = -2^{23} = -16777216 \). This would get expanded by the C preprocessor language to \(-1 \ ^\ 0x7FFFFFFF / 256\), which the compiler would happily treat as \(-1 \ ^\ (0x7FFFFFFF / 256)\), which is \(-8388608\). The problem is that the preprocessor doesn’t know or care about the order of operations in C: it’s just blindly substituting text. Parentheses would fix this particular problem:

\[
\#define \text{INT\_MIN} (-1 \ ^\ 0x7FFFFFFF)
\]

The best idea is to use \#define sparingly and mostly get your macro definitions from standard libraries. The definitions \text{INT\_MIN} and \text{INT\_MAX} are already provided by the standard C library <limits.h>.

## 5 Conditional compilation

Another very powerful but very-easy-to-get-wrong feature of the macro language is conditional compilation. Based on whether a symbol is defined or not, the preprocessor can choose to ignore a whole section of text or choose between separate sections of text. This is used in a couple of different ways. Sometimes we use \#ifndef (if not defined) to make sure we’re not defining something twice:

\[
\#ifndef \text{INT\_MIN}
\#define \text{INT\_MIN} (~0x7FFFFFFF)
\#endif
\]

We can also use \#ifdef and \#else to pick between different pieces of code to define. The code below is very different than C0/C code with an condition if (version_one) statement, because only one of the two print statements below will ever even get compiled. The other one will be cut out of the program by the preprocessor before the compiler even says it!

\[
\#ifdef \text{VERSION\_ONE}
\text{printf("This is version 1\n")};
\#else
\text{printf("This is not version 1\n")};
\#endif
\]

One interesting thing about this example is that we don’t care what \text{VERSION\_ONE} is defined to be: we’re just using the information about whether it is defined or not. We’ll use the \text{DEBUG} symbol in some of our C programs to include certain pieces of code only when \text{DEBUG} is defined.
#ifdef DEBUG
printf("Some helpful debugging information\n");
#endif

6 Macro functions

A more powerful version of macros definitions is macro functions:

#define MULT(x,y) ((x)*(y))

Using parentheses defensively is very important here, because otherwise
the precedence issues we described before will only get worse. The only
place we’ll use macro functions in 15-122 is to define something like C0
contracts in C. The macro functions ASSERT, REQUIRES, and ENSURES turn
into assertions when the DEBUG symbol is present, but otherwise they are
replaced by ((void)0), which just tells the compiler to do nothing at all.

#ifdef DEBUG

#define ASSERT(COND) ((void)0)
#define REQUIRES(COND) ((void)0)
#define ENSURES(COND) ((void)0)

#else

#define ASSERT(COND) assert(COND)
#define REQUIRES(COND) assert(COND)
#define ENSURES(COND) assert(COND)

#endif

The code above isn’t something you have to write yourself: it’s pro-
vided for you in the file contracts.h that will be in the lib directory of all
of our C projects in 15-122. Therefore, we write:

#include "lib/contracts.h"

in order to include these macro-defined contracts in our programs. When
we use quotes instead of angle brackets for #include, as we do here, it just
means that we’re looking for a library we wrote ourself and are using lo-
ocation, not a standard library that we expect the compiler will find wherever
it stores its standard library interfaces.

LECTURE NOTES
7 C0 contracts in C

There’s no assertion language in C: everything starting with //@ and every-thing written inside /*@ ... @*/ is just a treated as a comment and ignored. We’ll still write C0-style contracts in our interfaces, but those contracts are now just comments.

All contracts, including preconditions and postconditions, have to be written inside of the function.

```c
int absval(int x) {
    REQUIRES(x > INT_MIN);
    int res = x < 0 ? -x : x;
    ENSURES (res >= 0);
    return res;
}
```

There’s not a good replacement for loop invariants in C; they just have to be replaced with careful use of ASSERT.

8 Memory allocation

In C0, we allocate pointers of a particular type; in C, we actually allocate pointers of a particular size: the preprocessor function sizeof takes a type and returns the number of bytes in this type, and it is this size that we pass to the allocation function. The default way of allocating a struct or integer (or similar) in C is to use the function malloc, provided in the standard <stdlib.h> library.

C0: int* x = alloc(int);
C: int* x = malloc(sizeof(int));

One quirk with malloc is that it does not initialize the memory, so it would be a memory error to dereference x before storing some integer into x. (The computer is able to allocate memory slightly more efficiently if it doesn’t have to initialize that memory.)

Another quirk with malloc is that it is allowed to return NULL. Ultimately there is only a finite amount of memory accessible to the computer, and malloc will return NULL when there is no memory left to allocate. Therefore, we will usually use a 15-122 library "lib/xalloc.h", which provides the function xmalloc. The xmalloc function works the same way malloc does, except that the result is sure not to be NULL.
C: int* x = xmalloc(sizeof(int)); // x is definitely not NULL

By replacing alloc with xmalloc and sizeof, we can now translate our test.c0 file into test.c. The series of print statements has been replaced by a single function printf, which is a powerful way of writing C functions that we won’t talk about yet.

```c
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "lib/xalloc.h"

int main() {
    struct point2d* P = xmalloc(sizeof(struct point2d));
    P->x = -15;
    P->y = 0;
    P->y = P->y + absval(P->x * 2);
    assert(P->y > P->x && true);
    printf("x coord: %d\n", P->x);
    return 0;
}
```

We needed an extra line, P->y = 0; , that wasn’t present in the original file to cope with the fact that the malloc-ed y field isn’t initialized to 0 the way it was in C0.

9 Compiling

It won’t work yet, but we can now try to compile our modified simple.c and test.c files. When we call gcc, the C compiler, we’ll give it a long series of flags:

```
% gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG ...
```

The flags -Wall, -Wextra, and -Wshadow represent a bunch of optional compilation Warnings we want to get from the compiler, and -Werror means that if we get any warnings the code should not be compiled. The flag -std=c99 means that the version of C we are using is the one that was written down as the C99 standard, a standard we want to adhere to in a -pedantic way.
The flag -g keeps information in the compiled program which will be helpful for valgrind (which will come up in the next lecture). The flag -DDEBUG means that we want the preprocessor to run with the DEBUG symbol Defined. As we talked about before, this means that contracts will actually be checked at runtime: -DDEBUG is the C version of the -d flag for the C0 compiler and interpreter.

10 Separate Compilation

If we try to compile the translated C files we have so far, it won’t work:

```bash
% gcc ...all those flags... lib/* c simple.c test.c
test.c: In function main:
test.c:8:38: error: invalid application of sizeof to incomplete type...
  struct point2d* P = xmalloc(sizeof(struct point2d));
^
test.c:10:3: error: implicit declaration of function absval...
  P->y = P->y + absval(P->x * 2);
```

If compiling C worked like compiling C0, test.c would be able to see the interface from simple.c, which includes the definition of struct point2d and the type of absval, because simple.c came ahead of test.c on the command line. However, C doesn’t work this way: every C file is compiled separately from all the other C files.

To get our code to compile, we want to split up the simple.c file into two parts: the interface, which will go in the header file simple.h, and the implementation, which will stay in simple.c and will #include the interface "simple.h". Then, we can also #include the simple interface in test.c.

This is actually a good thing from the perspective of respecting the interface: test.c will have access to the interface in simple.h, but couldn’t accidentally end up relying on extra things defined in simple.c.
10.1 Interface: simple.h

In addition to containing the interface from simple.c0, the header file containing the simple.h interface, like all C header files, needs to use #ifndef, #define, and #endif. These three preprocessor declarations, in combination, make sure that we can only end up including this code one time, even if we intentionally or accidentally write #include "simple.h" more than once.

#ifndef _SIMPLE_H_
#define _SIMPLE_H_

    int absval(int x)
    
        /*@requires x >= INT_MIN; @*/
        /*@ensures \result >= 0; @*/

    struct point2d {
        int x;
        int y;
    };

#endif

10.2 Implementation: simple.c

The C file will include both the necessary libraries and the interface. The implementation should always #include the interface.

#include <limits.h>
#include "lib/contracts.h"
#include "simple.h"

int absval(int x) {
    REQUIRES(x > INT_MIN);
    int res = x < 0 ? -x : x;
    ENSURES(res >= 0);
    return res;
}
10.3 Main file: test.h

```c
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "lib/xalloc.h"
#include "simple.h"

int main() {
    struct point2d* P = xmalloc(sizeof(struct point2d));
    P->x = -15;
    P->y = 0;
    P->y = P->y + absval(P->x * 2);
    assert(P->y > P->x && true);
    printf("x coord: %d\n", P->x);
    return 0;
}
```

11 Memory leaks

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

```c
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.¹

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

¹or realloc, which we have not discussed
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.

Binary search trees are one example. When we allocate an element to the binary search tree, are we still in charge of freeing that element, or should it be freed when it is removed from the binary search tree? There are arguments to be made for both of these options. If we want the BST to “own” the reference, and therefore be in charge of freeing it, we can write the following functions that frees a binary search tree, given a function pointer that frees elements. We allow this function pointer to be NULL: if it’s NULL we assume we don’t own the elements, and don’t do anything to them.

```c
void elem_free_fn(void* x);

void tree_free(tree *T, elem_free_fn *Fr) {
    REQUIRES(is_ordtree(T));
    if(T != NULL) {
        if (Fr != NULL) (*Fr)(T->data);
        tree_free(T->left);
        tree_free(T->right);
        free(T);
    }
    return;
}

void bst_free(bst B, elem_free_fn *Fr) {
    REQUIRES(is_bst(B));
    tree_free(B->root);
    free(B);
    return;
}
```

Lecture Notes
We should never free elements allocated elsewhere; rather, we should use the appropriate function provided in the interface to free the memory associated with the data structure. Freeing a data structure (for instance, by calling `free(T)`) is something the client itself cannot do reliably, because it would need to be privy to the internals of its implementation. If we called `free(B)` on a binary search tree it would only free the header; the tree itself would be irrevocably leaked memory.