Lecture Notes on Polymorphism

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 20
March 29, 2011

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. The underlying issue that we are trying to solve in this lecture is nevertheless a deep one: how can a language support *generic* implementations of data structures that accomodate data elements of different types. The name *polymorphism* derives from the fact that data take on different forms for different uses of the same data structure.

A simple example is the data structure of stacks. In our C0 implementation, the definition of the stack interface used an unspecified type `elem` of elements.

```
typedef struct stack* stack;
bool stack_empty(stack S);       /* O(1) */
stack stack_new();               /* O(1) */
void push(stack S, elem e);      /* O(1) */
elem pop(stack S);               /* O(1) */
```

The type `elem` must be defined before this file is compiled. In our testing code we used

```
typedef int elem;
```

to test stacks of integers. So it was already true that the implementation was generic to some extent, but this genericity could not be exploited. For example, if we wanted a second client using stacks of strings, we would have to cut-and-paste our stack code and rename the functions in its interface to avoid conflicts. In this lecture we will see how we can make the implementation generic to allow reuse at different types.

2 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 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. Advanced 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.

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.

3 Lack of Memory Safety

When compared to C0, the most shocking difference is that C does not distinguish arrays from pointers. As a consequence, array accesses are not checked, 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 -std=c99 -pedantic -Werror
```

which generates all warnings, pedantically applies the C99 standard, and turns all warnings into errors. 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. Of course, there are many cases where C0 programs have a well-defined answer, where the corresponding C program would not. The most significant of these is integer overflow, the outcome of which is not defined in C while modular arithmetic is specified for C0.

4 Undefined Behavior in C

The most important undefined behaviors in C are:

Out-of-bounds array access: see the previous section.

Null pointer dereference: dereferencing the null pointer has undefined results. Because under most configuration this will lead to a "segmentation fault" (abort with signal `SIGSEGV`), it may be overlooked that this is not defined. In particular, if your code runs in kernel mode, as part of the operating system, it may not yield an exception.

Arithmetic overflow: when addition, subtraction, multiplication, or division overflow the precision of the integer type (usually `int`) then the result is undefined. This includes division by 0, but also simple overflow of addition.

Cast: when data values of certain types are cast to other types the result is sometimes undefined. Casts are discussed further in the next section.

5 Void Pointers

In C, a special type void* denotes a pointer to a value of unknown type. We can use this to make data structures generic by assigning the type void* to the stored elements. For example, an interface to generic stacks might be specified as

```
typedef struct stack* stack;
bool stack_empty(stack S); /* O(1) */
stack stack_new(); /* O(1) */
void push(stack S, void* e); /* O(1) */
void* pop(stack S); /* O(1) */
```

Notice the use of void* for the first argument to push and for the return type of pop.

How do we create a value of type void*? That actually is pretty easy, because we can just forget that we know a value has type t* for any type t and treat it as an element of type void*. This "forgetting" of information can be done implicitly, and does not require any special syntax. For example, with the declarations above, we can write

```
stack S = stack_new();
int* p = malloc(sizeof(int));
*p = 3;
push(p, S);
```

to push a pointer p onto the stack. The fact that p has type int* lets us use it as if it had type void*.

Complications arise when we are trying to *use* a pointer of type void*. For example, the following would be an error, after the above statements:

```
int y = *pop(S);
```

The problem is that the return type of pop is void*, dereferencing this would yield a value of type void, which does not exist (or, at least, does not match the type int declared for y). This last declaration is therefore not type-correct and has to be rejected by the compiler. However, we as clients of the stack data structure know that we have stored pointers to integers. Therefore, we are entitled to *cast* the result of type void* to a pointer of type int*. The syntax for casting an expression e to a type t is (t)e. So, above we could write:

```
int y = *(int*)pop(S);
```

As programmers, we are almost entirely on our own here: only our knowledge of what we pushed onto the stack makes this safe and correct. The compiler cannot check this at compile time, and the runtime system cannot check it at runtime. The latter limitation arises from the fact that in C we cannot inspect data at runtime and infer their types. This is unlike type-safe object-oriented languages like Java where so-called *down casts* can be checked at runtime because every object is tagged with its class. In that world, our type `void*` would be like the class `Object`.

If we do this incorrectly in C the result generally speaking is *undefined*. As an example, consider the following code fragment.

```
char* s = "15122";
push(s, S);
int y = 3+*(int*)pop(S);
```

The result is undefined, although there is a good chance it will execute and bind y to 842085684. To understand why, we first note that, in C, strings are represented as character arrays terminated by the `NUL` character `\0`. An element of type `char` is usually 1 byte (although that is certainly not guaranteed), which means the cast followed by the dereference interprets the ASCII code of the first 4 characters of `"15122"` as an integer. Now you only have to look up the ASCII code of the first four characters and know (a) that character arrays are just stored in consecutive bytes, and (b) that numbers are stored with their least significant byte at the lowest address.[1]

In summary, on the client side of a generic data type implementation, we have the following rules.

- When the data structure interface demands an argument of type `void*`, supply data of type `t*`, where t is the type of the data we would like to store in the data structure. C will implicitly consider a value of type `t*` as if it has type `void*`, forgetting some information.

- When the data structure interface returns a value of type `void*`, explicitly cast is to be of type `t*`, where t is as in the first rule, the type of the data previously stored.

Sometimes, C will insert an automatic cast, but as a matter of style it is clearer and easier to spot if such casts are explicit. An example where they are often omitted is in the next section.

[1] A so-called *little-endian* representation. Compare with *big-endian* representations where the most significant byte is at the lowest address.

6 Memory Allocation

Two examples of system-provided functions which return a generic pointer are `malloc` and `calloc`. They have prototypes

```
void* malloc(size_t size);
void* calloc(size_t nobj, size_t size);
```

The type `size_t` is an implementation-defined type. Typically, this would be `unsigned int` which represents words of the same number of bits as `int`, except that all numbers are interpreted as zero or positive. For 32 bit integers, this covers the range from 0 to $2^{32} - 1$, whereas `int` covers the range from -2^{-31} to $2^{31} - 1$. If arguments are actually of type `int` and positive, then they are implicitly cast as `unsigned int`, so in most cases we do not have to know the precise definition of `size_t`.

`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))` has type `void*`. We therefore need to explicitly cast it to the appropriate type. For example,

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

Actually, in this particular case, as the initializer in a declaration or on the right-hand size of an assignment, C can determine the type of the left-hand side and implicitly cast `void*` to `t*`. This may seem obvious here, but in some cases it can hide subtle errors when the left-hand side of the assignment is complex. Also, `malloc` does not guarantee that the memory it returns has been initialized.

`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 `int`s, `'\0'` for `char`, or `NULL` for pointers.

As a rule of thumb, unless `malloc` or `calloc` appear as initializers, one should coerce the result to the appropriate pointer type. A reason this is particularly important is because an incorrect allocation is generally hard to diagnose. Sometimes, too much space is allocated (which does not manifest itself as a bug, even with `valgrind`), sometimes too little (which can escape detection when memory references are not checked for validity).

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. Instead, we have defined functions `xmalloc` and `xcalloc` which are just like malloc and calloc, respectively, but abort computation in case the operaton fails. They thereby guarantee 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 9 for an explanation of header files).

7 Genericity on the Implementation Side

In the implementation of generic data types such as stacks, the treatment of the generic elements of type void* is quite simple. This is because the data structure carries values of this type, but doesn't examine or manipulate them. Some more advanced data structures do; in these cases more advanced techniques are necessary. These will be introduced in the next lecture.

Recall that we used linked lists to implement stacks. For generic stacks, the data in linked lists have type void*.

```
typedef struct list* list;
struct list {
  void* data;                   /* generic data */
  list next;
};

struct stack {
  list top;
  list bottom;
};
```

The function to push an element onto the stack has to store the data into a struct it allocates. Both the argument and the struct field will have type void*, so the data movement is well-typed without knowing what this pointer actually refers to.

```
void push(stack S, void* e)
//@requires is_stack(S);
//@ensures is_stack(S) && !stack_empty(S);
{
  REQUIRES(is_stack(S));
  list first = xmalloc(sizeof(struct list));
  first->data = e;
  first->next = S->top;
  S->top = first;
  ENSURES(is_stack(S) && !stack_empty(S));
}
```

We have left the C0 `@requires` and `@ensures` annotations in the code. The C compiler will see these as comments, since they are preceded by `//`, and this comment syntax is permitted by the C99 standard. We have also indicated how this translates into the use of two macros we have defined for C, `REQUIRES` and `ENSURES`. These are in all capitals because, by convention, macro names are written in all capitals.

8 Macros

Macros are another extension of C that we left out from C0. 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)
```

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. For example, our implementation of ROBBDs has a line at the beginning

```
#define BDD_LIMIT (1<<20)
```

which says that the number of nodes in a BDD can be at most $2^{20} = 1M$, which actually is not very much for BDDs. If it is insufficient, this constant can be changed at the beginning of the file and the code recompiled with a bigger limit. In general, it is considered good style to isolate "magic" numbers in macros at the beginning of a file, for easy reference. 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.

9 Header Files and Conditional Compilation

An important convention in C is the use of *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 (with the .c extension) using the #include preprocessor directive. The typical use is to #include[2] 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 stack implementation we have been discussing in this lecture (file stacks-test.c) starts with

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "xalloc.h"
#include "contracts.h"
#include "stacks.h"
```

[2] when we say "include" in the rest of this lecture, we mean #include

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.[3]

Let's focus on stacks.h, which contains the interface to stacks and has the following contents:

```
#include <stdbool.h>

#ifndef _STACKS_H_
#define _STACKS_H_

typedef struct stack* stack;
bool stack_empty(stack S); /* O(1) */
stack stack_new(); /* O(1) */
void push(stack S, void* e); /* O(1) */
void* pop(stack S); /* O(1) */
void stack_free(stack S); /* O(1), S must be empty! */

#endif
```

It defines the type stack as a pointer to a struct stack, whose implementation remains hidden. It also declares various functions, the last of which (stack_free) we have not yet discussed. It includes the standard library stdbool.h which defines the type bool as well as constants true and false. C actually does not distinguish between booleans and integers, treating the integer 0 as false and any non-zero integer as true. It is good programming style to use bool, true, and false as we have done in C0 whenever the values are indeed booleans. Many C programs write

```
while (1) { ... };
```

for an infinite loop, while I would strongly suggest

```
while (true) { ... };
```

[3] for example, http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

instead, if an infinite loop is indeed necessary.

We also see a certain idiom

```
#ifndef _STACKS_H_
#define _STACKS_H_
...
#endif
```

which is interpreted by the preprocessor, like other directives starting with `#`. This is a *header guard*, which prevents the header from being processed multiple times. The first time the header file is processed, the preprocessor variable `_STACKS_H_` will not be defined, so the test `#ifndef` (*if not defined*) will succeed. The next directive defines the variable `_STACKS_H_` (as the empty string, but that is irrelevant) and then processes the following declarations up to the matching `#endif`, usually at the end of the file.

Now if this file were included a second time, which happens frequently because standard libraries, for example, are included in many different source files that are compiled together, then the variable `_STACKS_H_` would be defined, the test would fail, and the body of the file ignored.

Header guards 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`. In particular, you can declare variables that exist in debug mode only in order to implement `\old(e)`.

10 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`.[4]
2. After memory has been freed, it is not longer referenced by the program in any way. This includes calling free again on a pointer referencing the same memory, which is prohibited.

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.

In this example, we add a simple function to the interface that can free a stack if it is empty.

```
void stack_free(stack S)
//@requires is_stack(S) && stack_empty(S);
{ REQUIRES(is_stack(S) && stack_empty(S));
  assert(stack_empty(S));
  free(S->top); /* == S->bottom */
  free(S);
}
```

In the code in our testing function, we have to call this function after all elements have been popped off the stack.

```
int main () {
  stack S = stack_new();
  int* x = (int*)malloc(sizeof(int));
  *x = 1;                         /* x is heap-allocated */
  push(x, S);
  char* s = "24247";              /* s is a constant string */
  push(s, S);                     /* mixed-type stack; DON'T DO THIS */
  assert((char*)pop(S) == s);     /* EVER! */
  assert(*(int*)pop(S) == 1);
  printf("All tests passed!\n");
  free(x);                        /* free x */
  assert(stack_empty(S));
  stack_free(S);                  /* stack is empty; free */
```

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

```
  return 0;
}
```

Note that we could not free s, because the constant string `"24247"` is a constant string, rather than one that has been allocated with `malloc` or `calloc`. Also, we should never free elements allocated elsewhere, but use the appropriate function provided in the interface to free the memory associated with the data structure. Freeing a data structure is something the client itself cannot do reliably, because it would need to be privy to the internals of its implementation.

In the next lecture we will learn more how to manage memory.