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 accommodate 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.

```c
typedef struct stack_header* 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) */
//@requires !stack_empty(S);
```

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

```c
typedef int elem;
```
Polymorphism

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. This actually happened in the Clac programming assignment, if you recall: we had the type istack, stacks of ints, as well as the type qstack, stacks of queues of strings.

In this lecture we will see how we can make code generic to allow reuse at different types. There are four key ideas that we need to consider in order to turn our type-specific C0 data structures into type-generic C data structures; we cover three in this lecture.

- **Void pointers** allow us to work with pointers to unknown types.
- **C's header files (*.h) and conditional compilation** affect the way that C interfaces and implementations are structured.
- **Function pointers** allow us to pass around functions that know the type of void pointers and can manipulate them accordingly.
- **Memory management** is critical in C, where we do not have a garbage collector as we did in C0. We will cover this in the next lecture.

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 \texttt{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 Undefined Behavior in C

The most important undefined behaviors in C are:

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

\textbf{Null pointer dereference:} dereferencing the null pointer has undefined results.
Polymorphism

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.

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

When compared to C0, the most shocking difference is that C does not distinguish arrays from pointers. 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

```c
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.

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).
3.2 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.

3.3 Undefined arithmetic

There are many cases where C0 programs have a well-defined answer, where the corresponding C program does not. The most significant of these is integer overflow. Overflow only matters in C0 for division by zero or division of `int_min()` by -1. Arithmetic overflow will *always* cause a C0 program to print out `Floating point exception` and exit. Because overflow is simply undefined in C, it is impossible to rely on this behavior. A reliable example is that many C compilers will “helpfully” optimize the expression `(x/y == x/y)` to true, thereby avoiding the division by zero that would occur if `y` was 0.

The fact that overflow of addition, multiplication, and subtraction is undefined is even more significant; the result means that many programs whose answer is defined in C0 have an undefined answers in C.

3.4 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 your 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 sys-
tem, 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.

## 4 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

```c
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

```c
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:

```c
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 will not check this at compile time, and the runtime system will not 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.

```c
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.³

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

³A so-called **little-endian** representation. Compare with **big-endian** representations where the most significant byte is at the lowest address.
• When the data structure interface demands an argument of type \texttt{void*}, supply data of type \texttt{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 \texttt{t*} as if it has type \texttt{void*}, forgetting some information.

• When the data structure interface returns a value of type \texttt{void*}, explicitly cast is to be of type \texttt{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.

5 Memory Allocation

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

\begin{verbatim}
void* malloc(size_t size);
void* calloc(size_t nobj, size_t size);
\end{verbatim}

The type \texttt{size_t} is an implementation-defined type. Typically, this would be \texttt{unsigned int} which represents words of the same number of bits as \texttt{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 \texttt{int} covers the range from \(-2^{31} \) to \( 2^{31} - 1 \). If arguments are actually of type \texttt{int} and positive, then they are implicitly cast as \texttt{unsigned int}, so in most cases we do not have to know the precise definition of \texttt{size_t}.

\texttt{malloc(sizeof(t))} allocates enough memory to hold a value of type \( t \). In C0, we would have written \texttt{alloc(t)} instead. The difference is that \texttt{alloc(t)} has type \texttt{t*}, while \texttt{malloc(sizeof(t))} has type \texttt{void*}. We therefore need to explicitly cast it to the appropriate type. For example,

\begin{verbatim}
int* p = (int*)malloc(sizeof(int));
\end{verbatim}

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 \texttt{void*} to \texttt{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, \texttt{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 ints, ‘\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 a tool like valgrind designed to catch memory errors), 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. Because 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 and Conditional Compilation

To understand how the xalloc library works, and to take our our C0 implementation of stacks and turn it into a good C implementation of stacks, 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 (with the .c extension) using the #include preprocessor directive. The typical use is to #include 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>

4when 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.\(^5\)

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

```c
#include <stdbool.h>
#ifndef _STACKS_H_
#define _STACKS_H_

typedef struct stack_header* 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

#include <stdbool.h>
```

Except for the use of `void*` instead of `elem`, the core of this file is exactly the interface part of our C0 stacks specification. 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`.

\(^5\)for example, [http://www.acm.uiuc.edu/webmonkeys/book/c_guide/](http://www.acm.uiuc.edu/webmonkeys/book/c_guide/)
Polymorphism

and any non-zero integer as true. It is good programming style to usebool, true, and false as we have done in C0 whenever the values are indeed booleans. Many C programs write

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

for an infinite loop, while I would strongly suggest

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

instead, if an infinite loop is indeed necessary.

We also see a certain idiom

```c
#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

```c
#ifndef DEBUG
...debugging statements...
#endif
```

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

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

Guarding debugging statements in this way generalizes the simple assertion macros provided in contracts.h.
7  Genericity on the Implementation Side

In the implementation of generic stacks, the treatment of the generic elements of type \texttt{void*} is quite simple. The data structure carries values of this type, but doesn’t examine or manipulate them. Some more advanced data structures do; we will consider a simple example of this Section 9.

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

\begin{verbatim}
struct list_node {
  void* data;
  struct list_node* next;
};
typedef struct list_node list;
\end{verbatim}

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 \texttt{void*}, so the data movement is well-typed without knowing what this pointer actually refers to.

\begin{verbatim}
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));
}
\end{verbatim}

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, REQUIRE and ENSURE. 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. 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.
9 Function pointers

The use of void* alone is sufficient when we only need to pass around references to data, but often we need to know something about that data. The client interface for hash tables needs to know how to get the key (which is generic) from an element (which is generic), and it also needs to know how to compare keys and hash them as integers. Heap-based priority queues need to be able to determine the priority of a generic element, and binary search trees need to be able to compare keys. Because the client knows what this function should be, it can define the function, but must somehow communicate that function to the library. The way the client does this is by passing the address of the function to the library, taking advantage of the fact that the implementation of a function is stored in memory like everything else in C and C0, and therefore functions have an address. These addresses are passed from client to library as pointers to functions.

We will use a simple example of sorting to demonstrate this. In C, our integer sorting function takes an array of integers, a lower bound, and an upper bound:

```c
void sort(int* A, int lower, int upper);
```

We cannot make this generic by changing int* to void** (an array of void pointers), because we have to be able to compare array elements to sort them.

A comparison function, as we have seen, takes two elements and returns a negative number if the first element is smaller and a positive number if the first element is bigger. So the comparison function for generic void* elements has the following signature:

```c
int compare(void* x, void* y);
```

If we want to compare strings (which have C type char*), we can use the `strcmp` function from the string library `<string.h>`:

```c
#include <string.h>
int string_compare(void* s1, void* s2) {
    return(strcmp((char*)s1, (char*)s2));
}
```

We can get a pointer to this function with the `address-of` operator by writing `&string_compare`. If `cmp` is a pointer obtained in this way, we can use it to compare two strings by writing `(*cmp)((void*)"hi", (void*)"yo")`. 

---

**LECTURE NOTES**  
November 1, 2011
Polymorphism

Note that when we write \((\ast\text{cmp})\), we are dereferencing the function pointer to get at the actual function!

Generic client functions like this comparison function must be used carefully – if \(x\) and \(y\) are pointers to integers, then the result of calling \(\text{string\_compare}((\text{void*})x, (\text{void*})y)\) is undefined. This is an easy mistake to make.

What is the type of a pointer to the function \(\text{string\_compare}\)? In other words, how would we define \(\text{cmp}\)? The answer will initially seem a bit odd. In C, we define \(\text{cmp}\) by writing

\[
\text{int } (*\text{cmp})(\text{void* } e1, \text{void* } e2) = &\text{string\_compare}.
\]

The best way to make sense of this is to think about declarations in C as being pattern matching against the way the declared variables will be used. We tell \(\text{cmp}\) what type it is by mimicking the way it is used, and we use the function pointer \(\text{cmp}\) by writing \((\ast\text{cmp})(e1,e2)\), which produces an integer given the void pointers \(e1\) and \(e2\).

It may be simpler to use a typedef to define the type \(\text{compare\_fun}\). In a typedef, we put the defined type where we would put the declared variable name in a declaration, so we write

\[
\text{typedef int } (*\text{compare\_fun})(\text{void* } e1, \text{void* } e2);
\]

With this type definition, we can declare the generic type of sorting functions in \texttt{sort.h}:

\[
\text{void sort(elem* A, int lower, int upper, compare\_fun compare);}\]

and we can use an implementation of this sorting function to sort an array of \texttt{void*} where each of the elements are actually strings:

\[
\text{void** S = xalloc(4, sizeof(\texttt{void*})\);}\]
\[
\text{S[0] = (\texttt{void*})"pancake";}\]
\[
\text{S[1] = (\texttt{void*})"waffle";}\]
\[
\text{S[2] = (\texttt{void*})"toast";}\]
\[
\text{S[3] = (\texttt{void*})"juice";}\]
\[
\text{sort(S, 0, 4, &\text{string\_compare});}\]

The sorting library doesn’t know, and doesn’t need to know, that the void pointers are actually character arrays (that is, C strings). All it needs to know is that the comparison function we passed to the library knows what these pointers are and is able to compare them.