Lecture Notes on
Generic Data Structures

15-122: Principles of Imperative Computation
Frank Pfenning, Penny Anderson

Lecture 21
April 3, 2014

1 Introduction

Using `void*` to represent pointers to values of arbitrary type, we were able to implement *generic* stacks in that the types of the elements were arbitrary (see Section 9 of Lecture 20). The main remaining restriction was that they had to be pointers. Generic queues or unbounded arrays can be implemented in an analogous fashion. However, when considering, say, hash tables or binary search trees, we run into difficulties because implementations of these data structures require operations on data provided by the client. For example, a hash table implementation requires a hash function and an equality function on keys. Similarly, binary search trees require a comparison function on keys with respect to an order. In this lecture we show how to overcome this limitation using function pointers.

2 The Hash Table Interface Revisited

Recall the client-side interface for hash tables, online here. The client must provide a type `elem` (which must be a pointer), a type `key` (which was arbitrary), a hash function on keys, an equality function on keys, and a function to extract a key from an element. We write `___` while a concrete type must be supplied there in the actual file.

```
/************************************/
/* Hash table client-side interface */
/************************************/
```

```
typedef ___* elem;
typedef ___ key;

int hash(key k, int m)
//@requires m > 0;
//@ensures 0 <= \result && \result < m;
  ;

bool key_equal(key k1, key k2);

key elem_key(elem e)
//@requires e != NULL;
  ;
```

We were careful to write the implementation so that it did not need to know what these types and functions were. But due to limitations in C0, we could not obtain multiple implementations of hash tables to be used in the same application, because once we fix `elem`, `key`, and the above three functions, they cannot be changed.

Given the above the library provides a type `ht` of hash tables and means to create, insert, and search through a hash table.

```
/*************************************/
/* Hash table library side interface */
/*************************************/
typedef struct ht_header* ht;

ht ht_new(int capacity)
//@requires capacity > 0;
  ;
elem ht_lookup(ht H, key k);    /* O(1) avg. */
void ht_insert(ht H, elem e)    /* O(1) avg. */
//@requires e != NULL;
  ;
```

3 Generic Types

Since both keys and elements are defined by the clients, they turn into generic pointer types when we implement a truly generic structure in C. We might try the following in a file `ht.h`.

```
#include <stbool.h>
#include <stdlib.h>

#ifndef _HASHTABLE_H_
#define _HASHTABLE_H_

typedef void *ht_elem;
typedef void *ht_key;

/* Hash table interface */
typedef struct ht_header *ht;

ht ht_new (size_t capacity);
void ht_insert(ht H, ht_elem e);
ht_elem ht_lookup(ht H, ht_key k);

#endif
```

We use type definitions instead of writing `void*` in this interface so the role of the arguments as keys or elements is made explicit (even if the compiler is blissfully unaware of this distinction). We write `ht_elem` now in the C code instead of `elem` to avoid clashes with functions of variables of that name.

However, this does not yet work. Before you read on, try to think about why not, and how we might solve it

4 Function Pointers

The problem with the approach in the previous section is that the implementation of hashtables must call the functions `elem_key`, `key_equal`, and `hash`. Their types would now involve `void*` but in the environment in which the hash table implementation is compiled, there can still only be one of each of these functions. This means the implementation cannot be truly generic. We could not even use two hash tables with different element types simultaneously this way, without copying code and renaming things. (This actually happened with stacks 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.)

The underlying issue that we are trying to solve in this lecture is 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. Sophisticated mechanisms to support polymorphism have been developed for modern high-level languages like Java and ML. Here we will look at a simple mechanism, the function pointer. In combination with void pointers and header files, function pointers give us the ability to write generic implementations of data structures. We use void pointers to pass around generic references to data, and function pointers to allow the client to specify to the library how to handle that data.

Because the client knows what these functions should be, it can define them, but must somehow communicate the definitions to the library. The way the client does this is by passing the *address* of a defined 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 therefore a function has an address. These addresses are passed from client to library as pointers to functions.

Leaving generic hash tables aside for a moment, we will use a simple example of sorting to demonstrate this. In C, we can write an integer sorting function that takes an array of integers, a lower bound, and an upper bound:

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

We cannot make this generic by simply 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, zero if they are equal, and a positive number if the first element is bigger. So the comparison function for generic `void*` elements has the following signature:

```
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>:

```
#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")`. Note that when we write `(*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 `string_compare((void*)x, (void*)y)` is undefined. This is an easy mistake to make.

What is the *type* of a pointer to the function `string_compare`? In other words, how would we define `cmp`? The answer will initially seem a bit odd. In C, we define `cmp` by writing

```
int (*cmp)(void* e1, void* e2) = &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 `cmp` what type it is by mimicking the way it is used, and we use the function pointer `cmp` by writing `(*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 `compare_fun`. In a typedef, we put the defined type where we would put the declared variable name in a declaration, so we write

```
typedef int (*compare_fun)(void* e1, void* e2);
```

With this type definition, we can declare the generic type of sorting functions in `sort.h`:

```
void sort(elem* A, int lower, int upper, compare_fun compare);
```

where `elem` is defined as `void*` for readability puruposes and we can use an implementation of this sorting function to sort an array of `void*` where the elements are actually strings:

```
void** S = xcalloc(4, sizeof(void*));
S[0] = (void*)"pancake";
S[1] = (void*)"waffle";
S[2] = (void*)"toast";
S[3] = (void*)"juice";
sort(S, 0, 4, &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.

5 Generic Operations via Function Pointers

We now return to our hash table implementation problem. With function pointers, we can make the hash table implementation truly generic by allowing the client to provide pointers to the functions for extracting, comparing, and hashing keys.

But where do we pass them? We could pass all three to `ht_insert` and `ht_lookup`, where they are actually used. However, it is awkward to do this on every call. We notice that for a particular hash table, all three functions should be the same for all calls to insert into and search this table, because a single hash table stores elements of the same type and key. We can therefore pass these functions just once, when we first create the hash table, and store them with the table!

This gives us the following interface (in file `ht.h`):

```
#include <stbool.h>
#include <stdlib.h>

#ifndef _HASHTABLE_H_
#define _HASHTABLE_H_

typedef void* ht_key;
typedef void* ht_elem;
```

```
/* Hash table interface */
typedef struct ht* ht;
ht ht_new (size_t capacity,
           ht_key (*elem_key)(ht_elem e),
           bool (*key_equal)(ht_key k1, ht_key k2),
           unsigned int (*key_hash)(ht_key k, unsigned int m));
void ht_insert(ht H, ht_elem e);
ht_elem ht_lookup(ht H, ht_key k);
void ht_free(ht H, void (*elem_free)(ht_elem e));

#endif
```

We have added the function `ht_free` to the interface. The latter takes a pointer to the function that frees elements stored in the table.

We have made some small changes to exploit the presence of unsigned integers (in `key_hash`) and the `size_t` type (also unsigned) to provide more appropriate types to certain functions.

Storing the function for manipulating the data brings us closer to the realm of object-oriented programming where such functions are called *methods*, and the structures they are stored in are *objects*. We don't pursue this analogy further in this course, but you may see it in follow-up courses, specifically 15-214 *Software System Construction*.

6 Using Generic Hashtables

First, we see how the client code works with the above interface. We use here the example of word counts, which we also used to illustrate and test hash tables earlier. The structure contains a string and a count.

```
/* elements */
struct wc {
  char *word;                   /* key */
  int count;                    /* information */
};
```

As mentioned before, strings are represented as arrays of characters (type `char*`). The C function `strcmp` from library with header `string.h` compares strings. We then define:

```
bool word_equal(ht_key w1, ht_key w2) {
```

```
  return strcmp((char*)w1,(char*)w2) == 0;
}
```

Keep in mind that `ht_key` is defined to be `void*`. We therefore have to cast it to the appropriate type `char*` before we pass it to `strcmp`, which requires two strings as arguments. Similarly, when extracting a key from an element, we are given a pointer of type `void*` and have to cast it as of type `struct wc*`.

```
/* extracting keys from elements */
ht_key elem_key(ht_elem e) {
  REQUIRES(e != NULL);
  struct wc *wcount = (struct wc*)e;
  return wcount->word;
}
```

The hash function is defined in a similar manner.

Here is an example where we insert strings created from integers (function `itoa`) into a hash table and then search for them.

```
int n = (1<<10);
ht H = ht_new(n/5, &elem_key, &key_equal, &key_hash);
for (int i = 0; i < n; i++) {
  struct wc* e = xmalloc(sizeof(struct wc));
  e->word = itoa(i);
  e->count = i;
  ht_insert(H, e);
}
for (int i = 0; i < n; i++) {
  char *s = itoa(i);
  struct wc *wcount = (struct wc*)(ht->lookup(H, s));
  assert(wcount->count == i);
  free(s);
}
```

Note the required cast when we receive an element from the table, while the arguments e and s do not need to be cast because the conversion from `t*` to `void*` is performed implicitly by the compiler.

7 Implementing Generic Hash Tables

The hash table structure, defined in file `hashtable.c` now needs to store the function pointers passed to it.

```
struct ht_header {
  size_t size;                  /* size >= 0 */
  size_t capacity;              /* capacity > 0 */
  chain **table;                /* \length(table) == capacity */
  ht_key (*elem_key)(ht_elem e);
  bool (*key_equal)(ht_key k1, ht_key k2);
  unsigned int (*key_hash)(ht_key k, unsigned int m);
  void (*elem_free)(ht_elem e);
};
```

We have also decided here to add the `elem_free` function to the hash table header, instead of passing it in to the free function. This exploits that we can generally anticipate how the elements will be freed when we first create the hash table. A corresponding change must be made in the header file `ht.h`.

```
ht ht_new(size_t capacity,
          ht_key (*elem_key)(ht_elem e),
          bool (*key_equal)(ht_key k1, ht_key k2),
          unsigned int (*key_hash)(ht_key k, unsigned int m),
          void (*elem_free)(ht_elem e))
{
  REQUIRES(capacity > 0);
  ht H = xmalloc(sizeof(struct ht_header));
  H->size = 0;
  H->capacity = capacity;
  H->table = xcalloc(capacity, sizeof(chain*));
  /* initialized to NULL */
  H->elem_key = elem_key;
  H->key_equal = key_equal;
  H->key_hash = key_hash;
  H->elem_free = elem_free;
  ENSURES(is_ht(H));
  return H;
}
```

When we search for an element (and insertion is similar) we retrieve the functions from the hash table structure and call them. It is good style to wrap this in short functions to make the code more readable. We use here the directive `static inline` to instruct the compiler to *inline* the function, which means that wherever a call to this function occurs, the compiler just replaces the call by the function body, for the sake of efficiency. This provides a similar but semantically cleaner and less error-prone alternative to C preprocessor macros.

```
static inline ht_key elemkey(ht H, ht_elem e) {
  return (*H->elem_key)(e);
}

static inline bool keyequal(ht H, ht_key k1, ht_key k2) {
  return (*H->key_equal)(k1, k2);
}

static inline unsigned int keyhash(ht H, ht_key k, unsigned int m) {
  return (*H->key_hash)(k, m);
}
```

We exploit here that C allows function pointers to be directly applied to arguments, implicitly dereferencing the pointer. We use

```
/* ht_lookup(H, k) returns NULL if key k not present in H */
ht_elem ht_lookup(ht H, ht_key k)
{
  REQUIRES(is_ht(H));
  int i = keyhash(H, k, H->capacity);
  chain* p = H->table[i];
  while (p != NULL) {
    ASSERT(p->data != NULL);
    if (keyequal(H, elemkey(H,p->data), k))
      return p->data;
    else
      p = p->next;
  }
  /* not in chain */
  return NULL;
}
```

This concludes this short discussion of generic implementations of libraries, exploiting `void*` and function pointers.

In more modern languages such as ML, so-called *parametric polymorphism* can eliminate the need for checks when coercing from `void*`. The corresponding construct in object-oriented languages such as Java is usually called *generics*. We do not discuss these in this course.

8 A Subtle Memory Leak

Let's look at the beginning code for insertion into the hash table.

```
void ht_insert(ht H, ht_elem e) {
  REQUIRES(is_ht(H));
  REQUIRES(e != NULL);
  ht_key k = elemkey(H, e);
  unsigned int i = keyhash(H, k, H->capacity);

  chain *p = H->table[i];
  while (p != NULL) {
    ASSERT(is_chain(H, i, NULL));
    ASSERT(p->data != NULL);
    if (keyequal(H, elemkey(H, p->data), k)) {
      /* overwrite existing element */
      p->data = e;
      return;
    } else {
      p = p->next;
    }
  }
  ASSERT(p == NULL);
  ...
}
```

At the end of the while loop, we know that the key k is not already in the hash table. But this code fragment has a subtle memory leak. Can you see it?[1]

[1] The code author overlooked this in the port of the code from C0 to C, but one of the students noticed.

The problem is that when we overwrite `p->data` with `e`, the element currently stored in that field may be lost and can potentially no longer be freed.

There seem to be two solutions. The first is for the hash table to apply the `elem_free` function it was given. We should guard this with a check that the element we are inserting is indeed new, otherwise we would have a freed element in the hash table, leading to undefined behavior.

```
  if (keyequal(H, elemkey(H, p->data), k)) {
    /* free existing element, if different from new one */
    if (p->data != e) (*H->elem_free)(e);
    /* overwrite existing element */
    p->data = e;
    return;
 }
```

The client has to be aware that the element already in the table will be freed when a new one with the same key is added.

In order to avoid this potentially dangerous convention, we can also just *return* the old element if there is one, and `NULL` otherwise. The information that such an element already existed may be useful to the client in other situations, so it seems like the preferable solution. The client could always immediately apply the element free function if that is appropriate. This requires a small change in the interface, but first we show the relevant code.

```
chain *p = H->table[i];
while (p != NULL) {
  ASSERT(p->data != NULL);
  if (keyequal(H, elemkey(H, p->data), k)) {
    /* overwrite existing element and return it */
    ht_elem tmp = p->data;
    p->data = e;
    return tmp;
  } else {
    p = p->next;
  }
}
```

The relevant part of the revised header file `ht.h` now reads:

```
typedef void* ht_elem;
typedef void* ht_key;

typedef struct ht_header* ht;

ht ht_new(size_t capacity,
          ht_key (*elem_key)(ht_elem e),
          bool (*key_equal)(ht_key k1, ht_key k2),
          unsigned int (*key_hash)(ht_key k, unsigned int m),
          void (*elem_free)(ht_elem e));

/* ht_insert(H,e) returns previous element with key of e, if exists */
ht_elem ht_insert(ht H, ht_elem e);

/* ht_lookup(H,k) returns NULL if no element with key k exists */
ht_elem ht_lookup(ht H, ht_key k);

void ht_free(ht H);
```

9 Separate Compilation

Although the C language does not provide much support for modularity, convention helps. The convention rests on a distinction between *header files* (with extension `.h`) and *program files* (with extension `c`).

When we implement a data structure or other code, we provide not only `filename.c` with the code, but also a header file `filename.h` with declarations providing the interface for the code in `filename.c`. The implementation `filename.c` contains `#include "filename.h"` at its top, and client will have the same line. The fact that both implementation and client include the same header file provides a measure of consistency between the two.

A header file `filename.h` should never contain any function definitions (that is, code), only type definitions, structure declarations, macros, and function declarations (so-called function prototypes). In contrast, a program file `filename.c` can contain both declarations and definitions, with the understanding that the definitions are not available to other files.

These header files have *header guards* that prevent the compiler from processing them more than once when compiling several files at the same time (thus they are sometimes called "once-only headers". The guards are directives to the C preprocessor, perhaps best explained by example. Here again is the header file for hashtables:

```
#include <stbool.h>
#include <stdlib.h>

#ifndef _HASHTABLE_H_
#define _HASHTABLE_H_

typedef void* ht_key;
typedef void* ht_elem;

/* Hash table interface */
typedef struct ht* ht;
ht ht_new (size_t capacity,
           ht_key (*elem_key)(ht_elem e),
           bool (*key_equal)(ht_key k1, ht_key k2),
           unsigned int (*key_hash)(ht_key k, unsigned int m));
void ht_insert(ht H, ht_elem e);
ht_elem ht_lookup(ht H, ht_key k);
void ht_free(ht H, void (*elem_free)(ht_elem e));

#endif
```

The presence of `#ifndef` ... `#endif` causes the preprocessor to check whether it has already defined `_HASHTABLE_H_`. The first time it scans the file, it will not have defined it (note the importance of choosing a name that is unlikely to occur in other headers!), and so it processes everything up to the `#endif`. Any subsequent scans will skip everything between `#ifndef` and `#endif`. In the case of this particular header, no harm is done other than a waste of time in processing it more than once. But unpleasant compiler errors can occur if headers in general are not once-only.

We only ever `#include` header files, never program files, in order to maintain the separation between code and interface.

When `gcc` is invoked with multiple files, it behaves somewhat differently than `cc0`. It compiles each file *separately*, referring only to the included header files. Those come in two forms, `#include <syslib.h>` where `syslib` is a system library, and `#include "filename.h"`, where `filename.h` is provided in the local directory. Therefore, if the right header files are not included, the program file will not compile correctly. We never pass a header file directly to `gcc`.

The compiler then produces a separate so-called *object file* `filename.o` for each `filename.c` that is compiled. All the object files are then *linked* together to create the executable. By default, that is `a.out`, but a name for the executable can be provided with the `-o` switch.

Let us summarize the most important conventions:

- Every file `filename`, except for the one with the `main` function, has a header file `filename.h` and a program file `filename.c`.
- The program `filename.c` and any client that would like to use it has a line `#include "filename.h"` at the beginning.
- The header file `filename.h` never contains any code, only macros, type definition, structure definitions, and function headers (prototypes). It has appropriate header guards to avoid problems if it is loaded more than once.
- We never `#include` any program files, only header files (with `.h` extension).
- We only pass program files (with `.c` extension) to `gcc` on the command line.

Exercises

Exercise 1 *Convert the interface and implementation for binary search trees from C0 to C and make them generic. Also convert the testing code, and verify that no memory is leaked in your tests. Make sure to adhere to the conventions described in Section 9.*