Lecture 14
Generic Data Structures

15-122: Principles of Imperative Computation (Spring 2024)
Rob Simmons, Iliano Cervesato

Our story about client interfaces in the previous chapter was incomplete. We were able to use client interfaces to implement queues and hash tables that treat the client's types as abstract (`elem` for queues and sets, `key` and `entry` for dictionaries), but any given program could only have a *single* type of keys, a *single* way of hashing.

To solve these problems, we will have to move beyond the C0 language to a language we call C1. C1 gives us two important features that aren't available in C0. The first new feature is a *void pointer*, which acts as a generic pointer. The second new feature is *function pointers*, which allow us to augment hash dictionaries with *methods*, an idea that is connected to Java and object-oriented programming.

Starting in this chapter, we will be working in an extension of C0 called C1. To get the `cc0` compiler to recognize C1, you need to use files with a `.c1` extension. Coin does not currently accept C1.

Additional Resources

- Review slides
  - Generic Pointers (`https://cs.cmu.edu/~15122/handouts/slides/review/12-voidstar.pdf`)
  - Function Pointers (`https://cs.cmu.edu/~15122/handouts/slides/review/14-fpointer.pdf`)
  - Generic Hash Dictionaries (`https://cs.cmu.edu/~15122/handouts/slides/review/14-generic.pdf`)
- Code for this lecture (`https://cs.cmu.edu/~15122/handouts/code/14-generic.tgz`)
- There is one short video associated with this lecture:
  - Generic Pointers (`https://youtu.be/kHthp7Gkyd0`)

Relating to our learning goals, we have

Computational Thinking: Structs with function pointers that can be used to modify the data contained within the struct is an important idea from object oriented programming.

Algorithms and Data Structures: We will revisit the idea of hash dictionaries in a new setting.

Programming: We explore void pointers and function pointers, which are necessary for creating truly generic data structures in C0/C1.

1 Generic Pointers: `void*`

We start by reexamining our queue data structure from last chapter. Then, the library manipulated data of type `elem` which the client had to provide beforehand by a line like

```
typedef string elem;    // supplied by client
```

One drawback of this is that the client program can only use a single queue type — queues of strings in this case. The client can use multiple queues of strings but not a queue of strings and a queue of **`int`**'s for example. If we need to do so, we have to make a copy of the queue library, renaming all its functions and types.

To avoid this, we need to abandon C0 and make use of a feature of the C1 language, the *void pointer* (written **`void*`**). C1 extends C0 with this and a few other features we will examine later in this chapter.

A variable `p` of type **`void*`** is allowed to hold a pointer to *anything*. Any pointer `p` can be turned into a void pointer by a *cast*, written `(void*)p`:

```
int*  ip = alloc(int);
void* p1 = (void*)ip;
void* p2 = (void*)alloc(string);
void* p3 = (void*)alloc(struct produce);
void* p4 = (void*)alloc(int**);
```

Because of this, void pointers are also called *generic pointers*.

When we have a void pointer, we can turn it back into the type it came from by casting in the other direction:

```
int* x = (int*)p1;
string x = *(string*)p2;
```

This is the only operation we are allowed to perform on a void pointer. In particular, void pointers do not support dereferencing: `*p1` would give us back a **`void`** which is not a type at all — it is just a marker to indicate that a function does not return a value. Thus, the name "void pointer" and the notation **`void*`** are terrible! (But that's what they are called in C.)

At run time, a non-`NULL` void pointer has a *tag*: casting incorrectly, like trying to run `(int*)p2` in the example above, is a *safety violation*: it causes a memory error just like a `NULL` dereference or array-out-of-bounds error.

These tags make void pointers a bit like values in Python: a void pointer carries the information about its true pointer type, and an error is raised if we treat a pointer to an integer like a pointer to a string or vice versa. Inside contracts, we can check that type with the `\hastag(type,p)` function:

```
//@assert \hastag(int*, p1);
```

```
//@assert \hastag(string*, p2);
//@assert \hastag(int***, p4);

//@assert !\hastag(string*, p1);
//@assert !\hastag(int**, p1);
//@assert !\hastag(int***, p1);
```

Like `\length` for example, `\hastag` cannot be used outside of contracts.

One quirk: since `NULL` has any pointer type, calling `\hastag(type, p)` on a **`void*`** variable `p` containing `NULL` always returns `true`. This lets us do slightly strange things like this without any error:

```
void* p = NULL;
void* x = (void*)(int*)(void*)(string*)(void*)(struct wcount*)p;
```

2 Generic Data Structures — II

We can make our queue library fully generic by simply choosing **`void*`** as the type `elem`. Therefore, the only change to the library code is to provide this one-line definition:

```
typedef void* elem;
```

On the client side, using this now fully generic queue library is somewhat more laborious. For one, elements must be pointers. Therefore the client cannot have a queue of integers: she shall turn it into a queue of integer pointers. Second, since the type `elem` is ultimately **`void*`**, she must cast her data to **`void*`** (or equivalently `elem`) before enqueuing it, and then cast it back to **`int*`** before accessing the value of a dequeued element. Here is a client code snippet where she enqueues 42 into a new queue, and prints the value at the front of the queue:

```
queue_t I = queue_new();
int* x = alloc(int);     // must store in allocated memory
*x = 42;
enq(I, (void*)x);        // must cast to void*
int* y = (int*)deq(I);   // must cast back before use
printf("%d", *y);
```

Now, the same program can also make use of a queue `S` meant to hold string (pointers). It can in fact make use of arbitrarily many queues, each containing data of a different type.

Nothing prevents a user from putting both **`int*`**'s and **`string*`**'s in the same queue. This is rarely advisable however, since the client would need to be able to predict the type of each dequeued element.

3 Towards Generic Hash Dictionaries

Now that we know about generic pointers, let's apply them to the hash dictionaries we developed in the last chapter. Recall that they were semi-generic, leaving the definition of the types `entry` of hash table entries and `key` of keys to the client — with the result that a program could use a hash table that stored a single type of entries.

The library interface simply declares `entry` and `key` (now a void pointer). The changes are highlighted.

```
typedef void* entry;
typedef void* key;

/*** Client interface ***/
key  entry_key(entry x)                  // Supplied by client
/*@requires x != NULL; @*/ ;
int  key_hash(key k);                    // Supplied by client
bool key_equiv(key k1, key k2);          // Supplied by client

/*** Library interface ***/
// typedef ______* hdict_t;

hdict_t hdict_new(int capacity)
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

entry hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/
/*@ensures \result == NULL
        || key_equiv(entry_key(\result), k); @*/ ;

void hdict_insert(hdict_t H, entry x)
/*@requires H != NULL && x != NULL; @*/
/*@ensures hdict_lookup(H, entry_key(x)) == x; @*/ ;
```

The changes to the client code are more interesting. We show the updates to just the function `entry_key` where they are most pervasive.

```
key entry_key(entry x)
//@requires x != NULL && \hastag(struct wcount*, x);
//@ensures \result != NULL && \hastag(string*, \result);
{
```

```
  string* k = alloc(string);
  *k = ((struct wcount*)x)->word;
  return (key)k;
}
```

The contracts refer to the abstract library types `key` and `entry`, but the client knows that in reality these types are pointers to **`string`**'s and **`struct`** `wcount`'s respectively. This is noted as two `\hastag` annotations. They prevent mistakes like passing an argument of the wrong type or using the result improperly — recall that `key` and `entry` are just nicknames to **`void`***. Aside from the `NULL`-check on the returned key (`NULL` is not a valid key for the client), we need to create space for it in allocated memory, cast the entry `x` to its original type, **`struct`** `wcount*` in order to be able to extract the word component we are using as key, and then cast the result into a `key` before returning.

Because the library contains the prototype of the client functions, these functions do not need to be defined before their use in the library functions. Therefore the library file can now appear first on the compilation command line, followed by the client files. In fact, there is no reason any more to split the client code into two files and sandwich the library in between.

Using void pointers solves the unnatural dependency between client functions and library code. It doesn't make our hash dictionaries fully generic however!

To understand why, try compiling a client program that uses two different dictionaries, one for counting words as above, and a second one with a different notion of entries and keys. Although void pointers can handle the different types, our program will necessarily contain multiple definitions of the client functions, like `entry_key`. Superficially, the compilation will fail because it finds duplicate definitions of such functions. More deeply, our hash tables do not know which version to use when. We need to tell them.

4 Memory Model

Up to now, we had a model of C0 execution where variables lived in what we called *local memory* and data created by means of **`alloc_array`** and **`alloc`** were stored in *allocated memory*. For example, here is a possible snapshot during the execution of an program that uses our hash dictionaries:

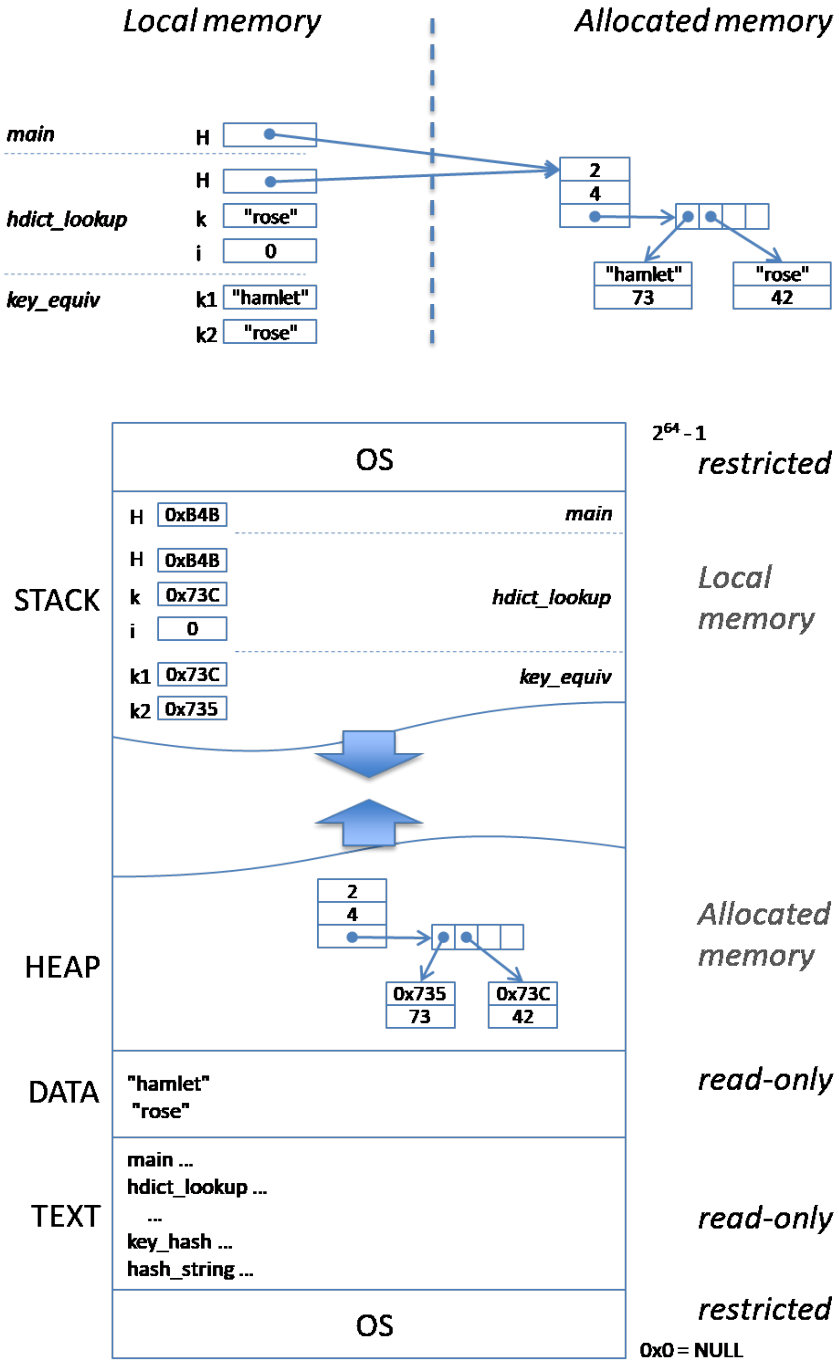

Figure 1: A More Concrete Memory Model

In reality, a computer contains a single entity we call *memory*, and it needs to store many more things than our variables and the data structures they point to. A more realistic model of how memory is organized is depicted in the figure **??**. We can think of memory as an array of bytes indexed by addresses — the very addresses coin reported when allocating arrays and pointers (but now there are more). Because addresses in C0 (and C1) have 64 bits, this array appears to have size 2^{64}. It is organized into a number of *segments*:

OS: The two ends of this memory array belong to the operating system. It uses these segments to hold its own data structures. A C0 program has no access to these areas — they are *restricted*. It is convention to use address `0x0` as `NULL`. It is an address after all, but one that we as programmers cannot do anything with.

Stack: What we referred to so far as "local memory" is normally called the *stack*. If we think about it, each time we enter a function, its local variables come into existence, and they go out of existence when we exit this function. The nested nature of function calls make local memory behave like a stack. The stack grows downward from the OS area at the end of the memory array.

Heap: Our old "allocated memory" is typically called the *heap*. As we allocate new data structures, it grows towards the stack.

DATA: Next comes the *DATA* segment which contains all the string literals present in our program. So far, we had the illusion that strings were stored in variables and in locations in allocated memory. In reality, these are addresses in the text area.

TEXT: The last segment we shall concern ourselves with is where the compiled code of our program resides. The code had to live somewhere after all!

5 Function Pointers

The language C1 provides an operator, written `&` and pronounced "*address-of*", to grab the address in the code segment that corresponds to a function in a program. For example, `&hash_string` is the address where the binary code for the function `hash_string` is located. The address-of operator can only be applied to functions in C1, although its use is much more general in C.

Now that we can get a hold of a function address, we would like to assign it to a variable, for example by writing `F = &hash_string` and then apply `F` to some input. But every variable shall have a type. What is the type of `F`? The *type of a function* describes the number, position and type of its input parameters and its output type. In C1, we declare a function type by just adding the word **`typedef`** in front of its prototype. Since the prototype of `hash_string` is

```
int hash_string(string s);
```

we write the type of this function as

```
typedef int hash_string_fn(string s);
```

By convention, we use `_fn` as a suffix for function types, just like we used `_t` as a suffix for client-side abstract data types. Note that `hash_string_fn` is not the type of just `hash_string`, but of every function that takes a **`string`** as input and returns an **`int`** — for example the predefined string function `string_length` has this type too!

We can now declare the variable `F` and assign `&hash_string` to it as follows:

```
hash_string_fn* F = &hash_string;
```

Note that `F` is pointer to a `hash_string_fn`, which is consistent with the view that it contains an address in memory. `F` is a *function pointer*. Because C1 variables, like C0 variables, can only hold values of small types, we cannot assign directly a function to a variable — thus

```
hash_string_fn F = hash_string;   // NOT ALLOWED
```

is illegal.

We don't create function pointers by dynamically allocating them the way we do structs: all the functions we could possibly have in our program are already known when we compile the program, and we grab them using &.

To call the function assigned to a function pointer, we dereference the pointer and then supply arguments. For example

```
(*F)("hello")
```

The parentheses around `*F` are important: `*F("hello")` would have the effect of attempting to apply `F` to `"hello"` and then dereference the result — the compiler will dutifully inform us that this makes no sense.

Like all other pointers, function pointers can be `NULL`, and it is a safety violation to dereference a `NULL` function pointer.

6 Generic Data Structures — III

Function pointers give us the mechanism for telling the library which client functions it should use at any point in the program. Whenever a hash dictionary function relies on client functions, we will provide it with pointers to the right functions.

Our first step is to convert the client function prototypes in the library to function types:

```
typedef key entry_key_fn(entry x)
        /*@requires x != NULL; @*/ ;
typedef int key_hash_fn(key k);
typedef bool key_equiv_fn(key k1, key k2);
```

Recall that this amounts to simply prefixing them with the keyword **typedef**.

Next, we need to decide *how* to provide the client functions to the library code. A first idea is to simply pass them as additional parameters, for example as in the following header

```
entry hdict_lookup(hdict* H, key k,
                   entry_key_fn* to_key, key_equiv_fn* equiv)
```

Although possible, this approach pushes the burden of using the right functions to the client, for each use of a library function. In a program that makes use of multiple hash dictionaries, it is easy to make mistakes — after all, every key equivalence function will have type `key_equiv_fn`!

A better approach is for the client to specify once and for all which functions to use when creating the hash table, via `hdict_new`, and storing these functions in the type that defines the hash table. To do this, we need to extend **struct** `hdict_header` with three fields of function pointer type:

```
typedef struct hdict_header hdict;
struct hdict_header {
  int size;              // 0 <= size
  chain*[] table;        // \length(table) == capacity
  int capacity;          // 0 < capacity
  entry_key_fn* key;     // != NULL
  key_hash_fn* hash;     // != NULL
  key_equiv_fn* equiv;   // != NULL
};
```

Note the requirements that these pointers be non-`NULL`. These requirements will need to be reflected in the data structure invariant functions `is_hdict`, which we omit.

The new fields get populated by the library function `hdict_new`, which is updated as follows:

```
hdict* hdict_new(int capacity, entry_key_fn* entry_key,
                 key_hash_fn* hash, key_equiv_fn* equiv)
//@requires capacity > 0;
//@requires entry_key != NULL && hash != NULL && equiv != NULL;
//@ensures is_hdict(\result);
{
  hdict* H = alloc(hdict);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  H->key = entry_key;
  H->hash = hash;
  H->equiv = equiv;
  return H;
}
```

Every time we create a hash dictionary, the new fields will contain pointers to the appropriate functions. Therefore, a hash dictionary now contains some of the functions that are used to manipulate it. This is one of the fundamental ideas underlying *object-oriented* programming. Objects are data fields bundled together with functions that operate on them. These functions are called *methods* in object-oriented languages like Java.

At this point, all it takes to make our hash dictionary library fully generic is to replace calls to `key_equiv` to `(*H->equiv)` throughout the implementation, where `H` refers to the current hash dictionary. Note that the operator precedence rules of C1 parse `(*H->equiv)` as `(*(H->equiv))`.

We can actually do slightly better, and at the same time isolate uses of C1's rather cumbersome syntax for method calls (compared to object-oriented languages). We avoid having lots of calls that use the counter-intuitive notation `(*H->equiv)(x,y)` by writing a helper function conveniently called `key_equiv` — our original name for this client function — so we can write calls that look like `key_equiv(H, x, y)`. We do the same for the other two client functions.

We show the resulting code for `hdict_lookup` as an example:

```
entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
/*@ensures \result == NULL
        || key_equiv(H, entry_key(H, \result), k); @*/
```

```
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(H, entry_key(H, p->data), k))
      return p->data;
  }
  return NULL;
}
```

Before moving to the client side, we need to revisit the library interface. Recall that we could write the prototype of `hdict_lookup` as

```
entry hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/
/*@ensures \result == NULL
        || key_equiv(entry_key(\result), k); @*/ ;
```

Now, the functions `key_equiv` and `entry_key` are internal to the implementation, and therefore they should not be made available to the client. Unraveling `key_equiv` into `(*H->equiv)` will not work either because it too exposes the internals of the implementation. Our only choice is to do without this postcondition. The interface prototype of `hdict_lookup` is therefore

```
entry hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/ ;
```

The upgrade we just made to the library has minimal effects on the client side. She simply needs to give new names to the client interface functions. She will also need to pass these functions each time she creates a new hash dictionary using `hdict_new`.

7 Exercises

Exercise 1 (sample solution on page **??**). *One way to store elements of different type in a linked list is to make it generic, giving its* `data` *field type* `void*`. *But then we won't be able to remember the actual type of the element when we need to use it. We use the additional field* `tag` *to keep track of it.*

```
typedef struct list_node genlist;
struct list_node {
  void* data;
  string tag;   // records the actual type of data
  genlist* next;
};
```

Assume that the only types we want to store in such a list are integers and strings, with tags `"an_int"` *and* `"a_string"` *respectively. Write a function that, given a generic linked list, prints each of its elements on a separate line. Include contracts as appropriate. If it encounters an unknown tag, it should call* `error` *with a description of the problem.*

Exercise 2 (sample solution on page **??**). *Define the function type* `int2int_fn` *of all functions that take an integer as input and return an integer. Then, implement the following two functions*

```
int compose(int2int_fn* f, int2int_fn* g, int x)
/*@requires f != NULL && g != NULL; @*/ ;

int pipeline(int2int_fn*[] F, int n, int x)
/*@requires \length(F) == n; @*/ ;
```

Given two mathematical functions f and g on the integers and an integer x, the first implement applying the composition of f and g, often written $f \circ g$, to x. Recall that $(f \circ g)(x) = f(g(x))$. The second applies all the functions in the `n`*-element array* `F` *to the integer* `x`*, left to right. If* `F` *is empty, it shall return* `x` *unchanged.*

Exercise 3 (sample solution on page **??**). *You're opening up a breakfast restaurant that sells pancakes and eggs. For every stack of pancakes that you sell, you want to keep track of the type of pancakes, of whether the pancakes have syrup, and of the number of pancakes. For eggs, you'll want to keep track of the number and style the eggs are cooked. Any customer that comes through will only ever get either the pancakes or eggs. We keep a pancake struct and an eggs struct and keep track of the breakfast for a customer, which could only ever be pancakes or eggs. Here are their type definitions:*

```
typedef struct pancakes pancakes;
struct pancakes {
  string type;
  bool syrup;
  int num;
};

typedef struct eggs eggs;
struct eggs {
  string style;
  int num;
};

typedef struct customer customer;
struct customer {
  string name;
  void* breakfast;
};
```

The following code is executed, and we have two customers: one that ordered a stack of two pancakes and the other a plate of eggs:

```
int main() {
  customer*[] customers = alloc_array(customer*, 2);

  eggs* janes = alloc(eggs);
  janes->style = "scrambled";
  janes->num = 3;

  customers[0] = alloc(customer);
  customers[0]->name = "Jane";
  customers[0]->breakfast = (void*)janes;

  pancakes* johns = alloc(pancakes);
  johns->type = "blueberry";
  johns->syrup = true;
  johns->num = 2;

  customers[1] = alloc(customer);
  customers[1]->name = "John";
  customers[1]->breakfast = (void*)johns;

  return 0;
```

```
}
```

Write the code you would need to get the type of pancake from the first customer and the number of eggs from the second customer. Then, draw the memory diagram of the code above. Indicate the tag of every memory cell containing a value of type `void*` *(if it has one).*

Sample Solutions

Solution of exercise ??
As we traverse the input list, we test the `tag` field of each node and use the appropriate print command to print its data value. Before we do so, we need to remember to cast and dereference the `data` field appropriately. An assertion that uses `\hastag` is a good way to anticipate the correct cast.

```
void print_genlist(genlist* L) {
  for (genlist* p = L; p != NULL; p = p->next) {
    if (string_equal(p->tag, "an_int")) {
      //@assert \hastag(int*, p->data);
      printf("%d\n", *(int*)(p->data));
    } else if (string_equal(p->tag, "a_string")) {
      //@assert \hastag(string*, p->data);
      printf("%s\n", *(string*)(p->data));
    } else {
      printf("%s", p->tag);
      error(" is an unknown tag");
    }
  }
}
```

Solution of exercise ??
The type declaration of `int2int_fn` is as follows:

```
typedef int int2int_fn(int x);
```

The function `compose(f,g,x)` simply applies `g` to `x` and then `f` to the resulting value. Because `f` and `g` are function *pointers*, we need to dereference them. To do so safely, we need to be sure they are not `NULL`, which we do by providing preconditions.

```
int compose(int2int_fn* f, int2int_fn* g, int x)
//@requires f != NULL && g != NULL;
{
  return (*f)((*g)(x));
}
```

The function `pipeline` follows similar ideas, this time for all the functions in the array `F`. Here, we perform our safety check as an assertion. Alternatively, we could have written a specification function that checks that all the element of the input array are non-`NULL`.

```
int pipeline(int2int_fn*[] F, int n, int x)
//@requires \length(F) == n;
{
  int res = x;
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  {
    //@assert F[i] != NULL;
    res = (*(F[i]))(res);
  }
  return res;
}
```

Solution of exercise ??
Here's the code to retrieve the pancake type of John's breakfast:

```
string type = ((pancakes*)(customers[1]->breakfast))->type;
```

and here's how to retrieve Jane's number of eggs:

```
int num = ((eggs*)(customers[0]->breakfast))->num;
```

The state of memory after this code executes is as follows: