Generic
Hash Dictionaries

Hash Dictionaries so far

The Hash Dictionary Library

Implementation

```
// Implementation-side types
typedef struct chain_node chain;
struct chain_node {
  entry data;            // data != NULL
  chain* next;
};
struct hdict_header {
  int size;              // size >= 0
  int capacity;          // capacity > 0
  chain*[] table;        // \length(table) == capacity
};
 typedef struct hdict_header hdict;

// Representation invariant
bool is_hdict (hdict* H) {
  return H != NULL
      && H->size >= 0
      && H->capacity > 0
      && is_array_expected_length(H->table, H->capacity)
      && is_valid_hashtable(H);
}

// Implementation of interface functions
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs(key_hash(k) % H->capacity);
}

entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
//@ensures \result == NULL
               || key_equiv(entry_key(\result), k);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next)
    if (key_equiv(entry_key(p->data), k))
      return p->data;
  return NULL;
}

void hdict_insert(hdict* H, entry e)
//@requires is_hdict(H) && e != NULL;
//@ensures hdict_lookup(H, entry_key(e)) == e;
//@ensures is_hdict(H);
{
  key k = entry_key(e);
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(entry_key(p->data), k)) {
      p->data = e;
      return;
    }
  }
  chain* p = alloc(chain);
  p->data = e;
  p->next = H->table[i];
  H->table[i] = p;
  (H->size)++;
}

hdict* hdict_new(int capacity)
//@requires capacity > 0;
//@ensures is_hdict(\result);
{
  hdict* H = alloc(hdict);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  return H;
}

// Client type
typedef hdict* hdict_t;
```

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry  e)
  /*@requires e != NULL;  @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

Library Interface

```
// typedef ______* hdict_t;

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

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

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

How

What

Is this Library Generic?

- Generic data structures
  - work the same way no matter the type of their data

> - Dictionaries are intrinsically generic
>   - ➢ they map keys to entries
>   - they work the same for any type of key and entry
> - *Hash* dictionaries should be generic
>   - they abstract key manipulations into client functions
>   - ➢ key_entry, key_hash and key_equiv

- Generic libraries
  - a single implementation that
    - ➢ lets the clients choose the type of their data
    - ➢ allows multiple instances of the data structure with different data types in the same application

Is this Library Generic?

- A single implementation that
  - lets the clients choose the type of their data
    - **Yes!**
    - the client interface mandates that the client define the types key and entry

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry  e)
  /*@requires e != NULL;   @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

    - the client does so in the client definition file
    - Let’s try it out to be sure

Client definition file

```
  }
  return h;
}

// What the client wants to store in the dictionary
struct inventory_item {
  string fruit;         // key
  int quantity;
};

/******* Fulfilling the library  interface *******/
typedef struct inventory_item* entry;
typedef string key;

key entry_key(entry e)
//@requires e != NULL;
{
  return e->fruit;
}

bool key_equiv(key k1, key k2) {
  return string_equal(k1, k2);
}

int key_hash(key k) {
  return lcg_hash_string(k);
}
```

Is this Library Generic?

- A single implementation that
  - lets the clients to chose the type of their data ✓

Compiles and run the code

Linux Terminal

```
# cc0 -dx produce.c0 hdict.c0 produce-main.c0
All produce tests passed!
0
# cc0 -dx lib/*.c0 words.c0 hdict.c0 words-main.c0
All word count tests passed!
0
```

Another client application that uses the hash dictionary to count the occurrences of each word in a file

Is this Library Generic?

- A single implementation that
  - lets the clients to chose the type of their data ✓
  - allows multiple instances of the data structure with different data types in the same application

Linux Terminal

```
# cc0 -dx produce.c0 lib/*.c0 words.c0 hdict.c0 combined-main.c0
words.c0:29.1-29.30:error:type name 'entry' defined more than once
previous definition at produce.c0:28.1-28.38
typedef struct wcount* entry;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Compilation failed
```

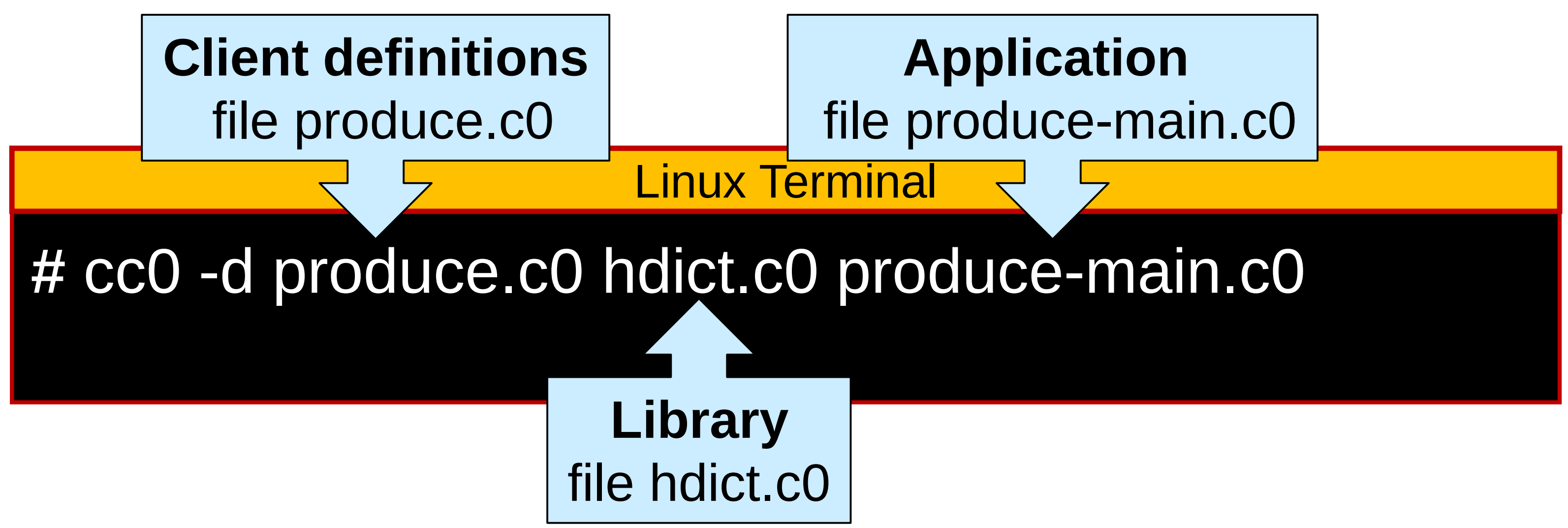
➢ there can be at most one definition of the types key and entry

✗

Is this Library Generic?

- *A single implementation that*
  - *lets the clients choose the type of their data* ✓
  - *allows multiple instances of the data structure with different data types in the same application* ✗

- This approach also forced clients to split their application code into two files



  - This is an unnatural compilation pattern
    - We would like to compile the hash dictionary library just the way we compile a stack library

Making this Library Generic

- With the stack library, setting the element type to void* solved both problems

```
typedef void* elem;
```

  - This was now C1 code

- Let's do the same with the hash dictionary library

void*
to the Rescue

Upgrading the Library

- The **only** changes we need to make to the library are defining key and entry as void*

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry e)
  /*@requires e != NULL; @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

Client Interface

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

key entry_key(entry e)
  /*@requires e != NULL; @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

We should not add NULL-check on keys since the client could have
- chosen key to be a pointer type
- chosen NULL to be a valid key

  - This only affects the client interface

That's it!

Upgrading the Client Definitions

Client Interface

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

key entry_key(entry e)
  /*@requires e != NULL;  @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- The client does not need to define key and entry
  - the library defines both to void*

```
/******* Fulfilling the library interface *******/
typedef struct inventory_item* entry;
typedef string key;
```

```
/******* Fulfilling the library interface *******/
// typedef struct inventory_item* entry;
// typedef string key;
```

- For the client
  - an entry is still a struct inventory_item*
  - a key is now a string*
    - "lime" must now live in a cell in allocated memory to be used as a key
    - NULL does not correspond to any valid key

It's got to be a pointer

Upgrading the Client Definitions

Client Interface

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

key entry_key(entry  e)
  /*@requires e != NULL;   @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- *For the client*
  - *an entry is still a struct inventory_item**
  - *a key is now a string**
- So,
  - every value of type entry must have tag struct inventory_item*
  - every value of type key must have tag string*

```
int key_hash(key k) {
  return lcg_hash_string(k);
}
```

⇨

```
int key_hash(key k)
//@requires k != NULL && \hastag(string*, k);
{
    …
}
```

NULL is not a valid fruit name

Every key is in reality a string*

Upgrading the Client Definitions

Client Interface

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

key entry_key(entry e)
  /*@requires e != NULL;  @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- *For the client, a key is now a string**
- Before using a value of type key, we need to
  - cast it to string*
  - dereference the result to a string

```
int key_hash(key k) {
  return lcg_hash_string(k);
}
```

```
int key_hash(key k)
//@requires k != NULL && \hastag(string*, k);
{
  return lcg_hash_string(*(string*)k);
}
```

lcg_hash_string takes a string as input

A void* cast to a string* and dereferenced to a string

Upgrading the Client Definitions

Client Interface

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

key entry_key(entry  e)
  /*@requires e != NULL;   @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- *For the client*
  - *an entry is still a struct inventory_item**
  - *a key is now a string**
    - *"lime" must now live in a cell in allocated memory to be used as a key*
    - *NULL does not correspond to any valid key*

- When extracting a key from an entry, we must put it in a cell in allocated memory

```
key entry_key(entry e)
//@requires e != NULL;
{
  return e->fruit;
}
```

```
key entry_key(entry e)
//@requires e != NULL && \hastag(struct inventory_item*, e);
//@ensures \result != NULL && \hastag(string*, \result);
{
  struct inventory_item* E = (struct inventory_item*)e;
  string* K = alloc(string);
  *K = E->fruit;
  return (key)K;
}
```

Auxiliary variables can help make those casts more readable

Upgrading the Client Definitions

Client Interface

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

key entry_key(entry  e)
  /*@requires e != NULL;   @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- When extracting a key from an entry, we must put it in a cell in allocated memory
  - add a helper function that turns a string to a string* for even better readability

```
key entry_key(entry e)
//@requires e != NULL;
{
  return e->fruit;
}
```

```
string* to_string_ptr(string s)
//@ensures \result != NULL;
{
  string* s_ptr = alloc(string);
  *s_ptr = s;
  return s_ptr;
}

key entry_key(entry e)
//@requires e != NULL && \hastag(struct inventory_item*, e);
//@ensures \result != NULL && \hastag(string*, \result);
{
  struct inventory_item* E = (struct inventory_item*)e;
  string* K = to_string_ptr(E->fruit);
  return (key)K;
}
```

We could write it in one line as
`return (key) to_string_ptr(((struct inventory_item*)e)->fruit);`
but that's unreadable

Upgrading the Client Definitions

Client Interface

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

key entry_key(entry e)
  /*@requires e != NULL;  @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

```
// What the client wants
struct inventory_item {
  string fruit;        // key
  int quantity;
};

/******* Fulfilling the library  interface *******/
typedef struct inventory_item* entry;
typedef string key;

key entry_key(entry e)
//@requires e != NULL;
{
  return e->fruit;
}

bool key_equiv(key k1, key k2) {
  return string_equal(k1, k2);
}

int key_hash(key k) {
  return lcg_hash_string(k);
}
```

Similar to key_hash

```
// What the client wants to store
struct inventory_item {
  string fruit;        // key
  int quantity;
};

/******* Fulfilling the library  interface *******/
key entry_key(entry e)
//@requires e != NULL && \hastag(struct inventory_item*, e);
//@ensures \result != NULL && \hastag(string*, \result);
{
  struct inventory_item* E = (struct inventory_item*)e;
  string* K = to_string_ptr(E->fruit);
  return (key)K;
}

bool key_equiv(key k1, key k2)
//@requires k1 != NULL && \hastag(string*, k1);
//@requires k2 != NULL && \hastag(string*, k2);
{
  return string_equal(*(string*)k1, *(string*)k2);
}

int key_hash(key k)
//@requires k != NULL && \hastag(string*, k);
{
  return lcg_hash_string(*(string*)k);
}
```

Upgrading the Client Application

- Cast entries and keys before calling the library operations
- Turn values of type string to string* before using them as keys

```
int main () {
  struct inventory_item* A = make_inventory_item("apple", 20);
  struct inventory_item* B = make_inventory_item("banana", 10);
  struct inventory_item* C = make_inventory_item("pumpkin", 50);
  struct inventory_item* D = make_inventory_item("banana", 20);

  hdict_t H = hdict_new(10);
  hdict_insert(H, A);
  hdict_insert(H, B);
  hdict_insert(H, C);
  assert(hdict_lookup(H, "apple") != NULL);
  assert(hdict_lookup(H, "lime") == NULL);
  hdict_insert(H, D);

 return 0;
}
```

```
int main () {
  struct inventory_item* A = make_inventory_item("apple", 20);
  struct inventory_item* B = make_inventory_item("banana", 10);
  struct inventory_item* C = make_inventory_item("pumpkin", 50);
  struct inventory_item* D = make_inventory_item("banana", 20);

  hdict_t H = hdict_new(10);
  hdict_insert(H, (entry)A);
  hdict_insert(H, (entry)B);
  hdict_insert(H, (entry)C);
  assert(hdict_lookup(H, (key)to_string_ptr("apple")) != NULL);
  assert(hdict_lookup(H, (key)to_string_ptr("lime")) == NULL);
  hdict_insert(H, (entry)D);

 return 0;
}
```

Generic Hash Dictionaries

- Let's try it on our examples

Linux Terminal

```
# cc0 -dx hdict.c1 produce.c1 produce-main.c1
All produce tests passed!
0
# cc0 -dx hdict.c1 lib/*.c0 words.c1 words-main.c1
All word count tests passed!
0
```

We upgrade the word count application in the same way

  - we can now put the library **before** all client files
  - this means we could merge produce.c1 and produce-main.c1 into a single file
    - same thing for word.c1 and words-main.c1

Generic Hash Dictionaries

- Let’s try it on our examples

Linux Terminal

```
# cc0 -dx hdict.c1 produce.c1 lib/*.c0 words.c1 combined-main.c1
words.c1:38.1-45.2:error:function 'entry_key' defined more than once
previous definition at produce.c1:31.1-38.2
key entry_key(entry x) ... }
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Compilation failed
```

✘

- This still doesn’t work!
  - both produce.c1 and words.c1 define entry_key
    - and key_equiv and key_hash
  - This is not allowed in C0/C1
  - Even if it were, the library wouldn’t know which version to use with what data

Function Pointers to the Rescue

Renaming the Client Functions

- We avoid having duplicate client definition function names by renaming them
  - key_hash to key_hash_produce
  - etc
  - and similarly for the word count application

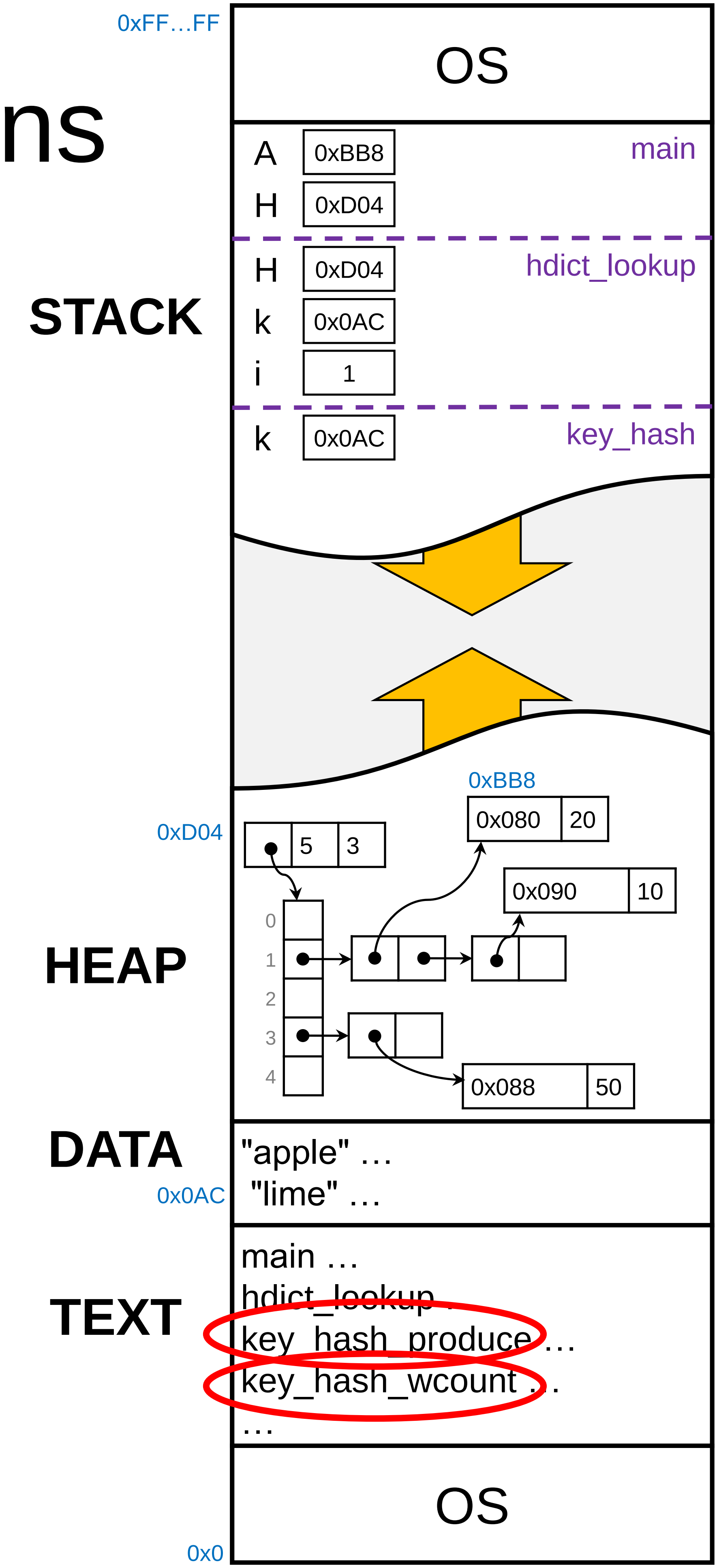
- But how to tell the library which function to use?

This is all we need to do in the client definition file

```
// What the client wants to store
struct inventory_item {
  string fruit;        // key
  int quantity;
};

/******* Fulfilling the library  interface *******/
key entry_key_produce(entry e)
//@requires e != NULL && \hastag(struct inventory_item*, e);
//@ensures \result != NULL && \hastag(string*, \result);
{
  struct inventory_item* E = (struct inventory_item*)e;
  string* K = to_string_ptr(E->fruit);
  return (key)K;
}

bool key_equiv_produce(key k1, key k2)
//@requires k1 != NULL && \hastag(string*, k1);
//@requires k2 != NULL && \hastag(string*, k2);
{
  return string_equal(*(string*)k1, *(string*)k2);
}

int key_hash_produce(key k)
//@requires k != NULL && \hastag(string*, k);
{
  return lcg_hash_string(*(string*)k);
}
```

Accessing the Right Functions

- During execution, functions live in the TEXT segment of memory
  - & allows us to obtain their address and pass it around as a function pointer
  - we can call a function through a pointer to it
- **Idea:** make pointers to the appropriate client function available to the library
  - *but how to do so?*



Accessing the Right Functions – I

- One option is to pass the right client functions to the library functions that use them
  - So,

```
entry A = make_inventory_item("apple", 20);
hdict_insert(H, A);
```

becomes

```
entry A = make_inventory_item("apple", 20);
hdict_insert(H, A,  &entry_key_produce,
                    &key_hash_produce,
                    &key_equiv_produce);
```

- Then do this for every use of hdict_insert and hdict_lookup
  - We will make mistakes
  - This is also a poor way of thinking about dictionaries ✘
    - H is a dictionary where we want to store produce
    - every insertion and lookup on H will need these client functions
      - not others

Accessing the Right Functions – II

- A better option is to pass the right client functions when we *create* a dictionary
  - in hdict_new

> This is all we need to do in the client application file

```
hdict_t H = hdict_new(10);
```

```
hdict_t H hdict_new(H,&entry_key_produce
                      &key_hash_produce,
                      &key_equiv_produce);
```

- hdict_new needs to store the client functions in H itself
  - we need to modify the internal representation
  - but first we need to give types to the client functions

Client Function Types

- We need to define types for the client functions
  - entry_key_fn
  - key_hash_fn
  - key_equiv_fn

  since
  - hdict_new takes them as arguments
  - we will store them in the concrete implementation type

- These definitions go in the client interface

Client Interface

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

key entry_key(entry e)
  /*@requires e != NULL;  @*/ ;
int key_hash(key k);
bool key_equiv(key k1, key k2);
```

Client Interface

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

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

Upgrading the Concrete Type

- We store the client definitions in the data structure itself
  - extend struct hdict_header with three additional fields

```
struct hdict_header {
  int size;              // size >= 0
  int capacity;          // capacity > 0
  chain*[] table;        // \length(table) == capacity
};
typedef struct hdict_header hdict;
```

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

- Storing both data and functions in a struct is a fundamental concept in **object-oriented programming**
  - these structs are called **objects**
  - the functions are called **methods**

There is a lot more to object-oriented programming however

Upgrading the Representation Invariant

- A valid hdict cannot have NULL in the added fields

```
bool is_hdict (hdict* H) {
  return H != NULL
      && H->size >= 0
      && H->capacity > 0
      && is_array_expected_length(H->table, H->capacity)
      && is_valid_hashtable(H);
}
```

```
bool is_hdict (hdict* H) {
  return H != NULL
      && H->size >= 0
      && H->capacity > 0
      && is_array_expected_length(H->table, H->capacity)
      && H->key   != NULL
      && H->hash  != NULL
      && H->equiv != NULL
      && is_valid_hashtable(H);
}
```

Upgrading hdict_new

- hdict_new
  - takes the client functions as inputs
  - expects them to be non-NULL
  - stores them in the added fields of the concrete type

```
hdict* hdict_new(int capacity)
//@requires capacity > 0;
//@ensures is_hdict(\result);
{
  hdict* H = alloc(hdict);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  return H;
}
```

⇒

```
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;
}
```

Calling the Client Functions

- Whenever we need a client function,
we call the function pointer in the data structure

E.g., key_hash

Here, H->hash

- For example,

```
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs(key_hash(k) % H->capacity);
}
```

```
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs((*H->hash)(k) % H->capacity);
}
```

This is the same as (*(H->hash))(…)

Upgrading hdict_lookup

- Proceed in the same way
  - change client function calls to function pointer calls

```
entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
//@ensures \result == NULL
                || key_equiv(entry_key(\result), k);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(entry_key(p->data), k))
        return p->data;
  }
  return NULL;
}
```

```
entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
//@ensures \result == NULL
                || key_equiv(entry_key(\result), k);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if ((*H->equiv)((*H->key)(p->data), k))
        return p->data;
  }
  return NULL;
}
```

- The function pointer syntax is hard to read
  - factor it out in helper functions similar to index_of_key

Upgrading hdict_lookup

- The function pointer syntax is hard to read
  - factor it out in helper functions similar to index_of_key

```
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs((*H->hash)(k) % H->capacity);
}

key entry_key(hdict* H, entry x)
//@requires is_hdict(H) && x != NULL;
{
  return (*H->key)(x);
}

bool key_equiv(hdict* H, key k1, key k2)
//@requires is_hdict(H);
{
  return (*H->equiv)(k1, k2);
}
```

```
entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
//@ensures \result == NULL
               || key_equiv(entry_key(\result), k);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(entry_key(p->data), k))
      return p->data;
  }
  return NULL;
}
```

⇒

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

- hdict_insert is similar

Updating the Library Interface

- The client interface does not contain entry_key and key_equiv any more
  - We cannot call them in the library interface

Library Interface

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

✗

- Using the fields of the implementation type would violate the interface

Library Interface

```
entry hdict_lookup(hdict_t D, key k)
  /*@requires D != NULL;                      @*/
  /*@ensures \result != NULL
             || (*H->equiv)((*H->key)(\result, k); @*/ ;
```

✗

- We must give up on this refined postcondition

Library Interface

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

✓

The Hash Dictionary Library

Implementation

```
// Implementation-side types
typedef struct chain_node chain;
struct chain_node {
  entry data;              // data != NULL
  chain* next;
};
struct hdict_header {
  int size;                // size >= 0
  int capacity;            // capacity > 0
  chain*[] table;          // \length(table) == capacity
  entry_key_fn* key;       // != NULL
  key_hash_fn* hash;       // != NULL
  key_equiv_fn* equiv;     // != NULL
};
 typedef struct hdict_header hdict;

// Representation invariant
bool is_hdict (hdict* H) {
  return H != NULL
     && H->size >= 0
     && H->capacity > 0
     && is_array_expected_length(H->table, H->capacity)
     && H->key   != NULL
     && H->hash  != NULL
     && H->equiv != NULL
     && is_valid_hashtable(H);
}

// Implementation of interface functions
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
   return abs((*H->hash)(k) % H->capacity);
}
key entry_key(hdict* H, entry x)
//@requires is_hdict(H) && x != NULL;
{
  return (*H->key)(x);
}
```

```
bool key_equiv(hdict* H, key k1, key k2)
//@requires is_hdict(H);
{
  return (*H->equiv)(k1, k2);
}

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

void hdict_insert(hdict* H, entry e)  // left as exercise

hdict* hdict_new(int capacity,
     entry_key_fn* entry_key, key_hash_fn* hash,
     key_equiv_fn* equiv)
//@requires capacity > 0 && entry_key != NULL ;
//@requires 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;
}

// Client type
typedef hdict* hdict_t;
```

Client Interface

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

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

Library Interface

```
// typedef ______* hdict_t;

hdict_t  hdict_new(int capacity ,
                   entry_key_fn* entry_key,
                   key_hash_fn* hash,
                   key_equiv_fn* equiv)
  /*@requires capacity > 0 && entry_key != NULL ;@*/
  /*@requires hash != NULL && equiv != NULL;     @*/
  /*@ensures \result != NULL;                    @*/ ;

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

void hdict_insert(hdict_t D, entry e)
  /*@requires D != NULL && e != NULL;            @*/ ;
```

Is it Generic?

Linux Terminal

```
# cc0 -dx hdict.c1 produce.c1 lib/*.c0 words.c1 combined-main.c1
All word count tests passed!
All produce tests passed!
0
```

- Yes!

✓

Harmony

Experimenting with Client Definitions

- Now that we have an easy way to specify which client definition functions to use, we can test a few things
  - Let's consider alternatives versions of key_hash and key_equiv
    - they look at the length of the key

This is a bad hash function. Let's use it anyway for simplicity

```
int key_hash_produce(key k)
//@requires k != NULL && \hastag(string*, k);
{
  return lcg_hash_string(*(string*)k);
}
```

```
int key_hash_produce_alt(key k)
//@requires k != NULL && \hastag(string*, k);
{
  return string_length(*(string*)k);
}
```

```
bool key_equiv_produce(key k1, key k2)
//@requires k1 != NULL && \hastag(string*, k1);
//@requires k2 != NULL && \hastag(string*, k2);
{
  return string_equal(*(string*)k1, *(string*)k2);
}
```

```
bool key_equiv_produce_alt(key k1, key k2)
//@requires k1 != NULL && \hastag(string*, k1);
//@requires k2 != NULL && \hastag(string*, k2);
{
  return string_length(*(string*)k1) == string_length(*(string*)k2);
}
```

This are our original functions

This are our alternative functions

Mixing and Matching

- key_hash_produce and key_equiv_produce are meant to be used together

```
hdict_t H hdict_new(H,  &entry_key_produce,
                        &key_hash_produce, &key_equiv_produce);
```

  - Same for key_hash_produce_alt and key_equiv_produce_alt

```
hdict_t H hdict_new(H,  &entry_key_produce,
                        &key_hash_produce_alt, &key_equiv_produce_alt);
```

- But what if we mix and match them?

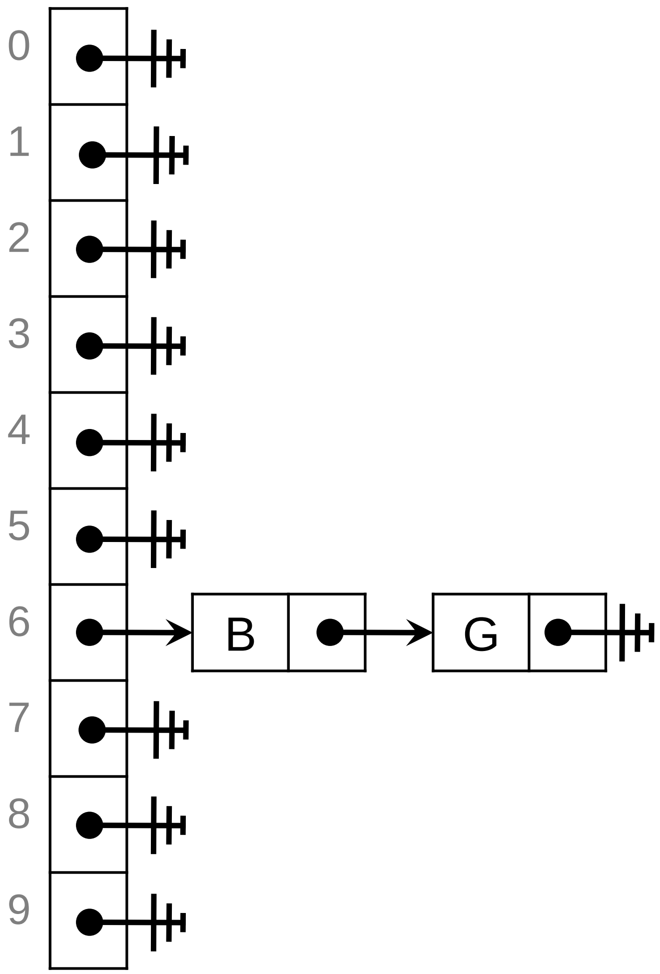
Mixing and Matching

- But what if we mix and match them?
  - key_hash_produce_alt with key_equiv_produce

```
hdict_t H hdict_new(H,  &entry_key_produce,
                        &key_hash_produce_alt, &key_equiv_produce);
```

  - Let's use the dictionary

✓ new dictionary
✓ insert B = ("banana", 10)
✓ insert G = ("grapes", 30)



- ❑ key_hash_produce_alt returns 6 on both
  - both "banana" and "grapes" have length 6
- ❑ both end up in the same bucket
  - but key_hash_produce would have sent them in different buckets

➢ This is not as **efficient** as using key_hash_produce
- ❑ the dictionary works correctly, but not is not as fast

Mixing and Matching

- But what if we mix and match them?
  - key_hash_produce with key_equiv_produce_alt

That's the other way around

```
hdict_t H hdict_new(H, &entry_key_produce,
                    &key_hash_produce, &key_equiv_produce_alt);
```

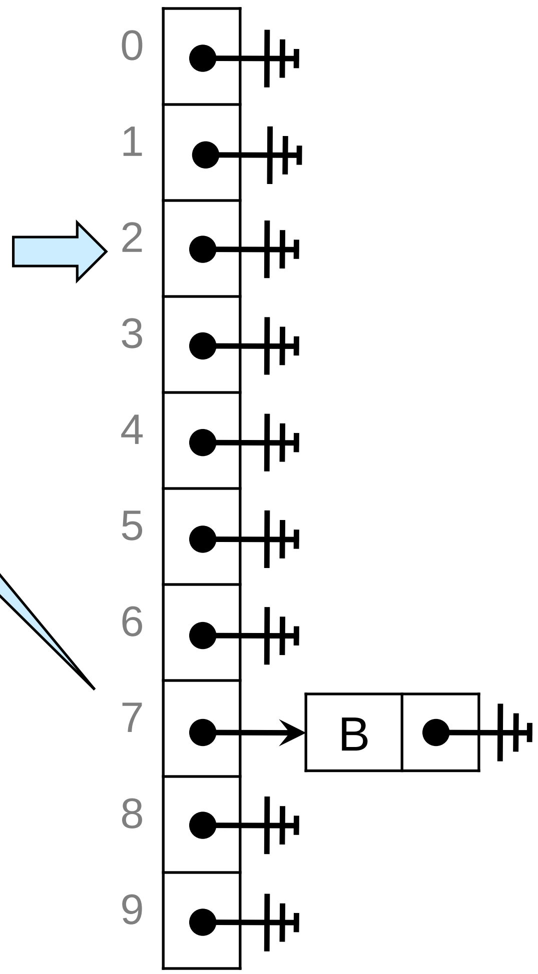
  - Let's use the dictionary

✓ new dictionary
✓ insert B = ("banana", 10)
look up "grapes"

key_hash_produce returns 7 on "banana"

- key_hash_produce returns 2 on "grapes"
- there is nothing in bucket 2
- lookup "grapes" returns NULL

➢ **This is incorrect!**
  - "grapes" and "banana" have length 6
  - key_equiv_produce_alt treats them as equal
- What *look up "grapes"* asks is *find an entry whose key has length 6*
  - B fits the bill, but it is not found



Harmony

- The key hash and equivalence functions are in **harmony** when equivalent entries have the same hash value
  - key_hash_produce and key_equiv_produce_alt are **not** in harmony
    - "banana" and "grapes" are equal according to key_equiv_produce_alt
    - but, according to key_hash_produce
      "banana" hashes to index 7 while "grapes" hashes to index 2
  - the other combinations are in harmony
- When they are not in harmony, the hash dictionary does not work correctly
  - it may return the wrong answer
- Harmony does not guarantee efficiency