Heap Review & Abstract Data Types

Joe Sventek
Objectives

- Review the use of pointers in C
- Review dynamic memory allocation and return using malloc()/free()
- Describe how “void *” can be exploited to provide generic abstract data types in C
- Demonstrate this through the complete specification of a generic Stack ADT in C
Pointers

- A pointer is a variable that contains the address of another variable.
- A typical machine has an array of consecutively numbered (or addressed) memory cells that can be manipulated individually or in contiguous groups – assume \( N \) cells, numbered \( 0 \ldots N-1 \).
- Suppose that we have a `char` variable named `c`, and that it is assigned to location `M`.
- Now suppose that we have a variable `p` which is a pointer to a character; `p` will be assigned to a location, say `L`, and a pointer will typically occupy >1 bytes, usually 4 or 8, depending upon the memory architecture of your processor.
More on pointers

- We make $p$ point to $c$ with a statement of the form $p = \&c$;
- The unary operator $\&$ gives the address of a variable, and is verbalized as “address of”
- $p$ is said to “point to” $c$
- $\&$ can only be applied to variables and array elements; it cannot be applied to expressions, constants, or register variables.
More on pointers

- The unary operator \* is the indirection or dereferencing operator.
- When applied to a pointer, it accesses the object the pointer points to.
- Consider the following artificial sequence of statements showing the use of & and *:

```c
int x = 1, y = 2, z[10];
int *p, *q;
/* p and q are pointers to an int */

p = &x; /* p now points to x */
y = *p; /* y is now 1 */
*p = 0; /* x is now 0 */
q = &z[0]; /* q now points to z[0] */
p = q; /* p now points to z[0] */
```
More on pointers

- Note that the declaration for a pointer to an `int` is `int *p;`
- This indicates that `*p` can be used anywhere that an `int` is legal, or that `p` must be dereferenced to yield an `int` – i.e. `p` is a pointer to an `int`
- Pointers are constrained to point to a particular kind of object – in this case, `p` is a pointer to an `int`
- If `p` points to an integer `x`, then `*p` can occur in any context where `x` could
- What happens for each of the following? Assume the following declarations:

```c
int y, x[2] = {1, 9}, *p = &x[0];
y = *p + 1; // Retrieve x[0], add 1, store in y; y now has a value of 2
*p += 1; // Add 1 to x[0], storing the value in x[0] (now 2)
++*p; // Increment value of x[0] (now 3), then return its value
*p++; // Return the value of x[0] (3), then increment value of x[0] (now 4)
*(p++); // Increment p to point to x[1], return its value (9)
```
Pointers and arrays

- Pointers and arrays are strongly related in C
- Any operation that can be achieved by array subscripting can also be done with pointers
- Consider the following declaration
  \[
  \text{int } a[10];
  \]
- This defines an array \( a \) of size 10 \( \)\text{ – i.e. a block of 10 consecutive int objects named } a[0], a[1], \ldots, a[9] \( \)
- \( a[i] \) refers to the \( i^{th} \) element of the array
- assume \( \text{pa} \) is a pointer to an integer, declared as \( \text{int } \ast \text{pa}; \)
More pointers and arrays

- the assignment \( \text{pa} = \&\text{a}[0] \); causes \text{pa} to point to element zero of \text{a}; i.e. \text{pa} contains the address of \text{a}[0]

- the assignment \( x = *\text{pa} \); copies the contents of \text{a}[0] into \text{x}

- by definition, \( \text{pa} + 1 \) points to the next element of the array, \( \text{pa} + i \) points \( i \) elements past \text{pa}, and \( \text{pa} - i \) points \( i \) elements before \text{pa}
More pointers and arrays

- The preceding statements are true regardless of the type or size of the variables in the array `a`.
- The meaning of “add 1 to a pointer” and by extension, all pointer arithmetic, is that `pa+1` points to the next object, and `pa+i` points to the `i`th object beyond `pa`.
- The value of a variable or expression of type `array` is the address of element 0 of the array – i.e. `a == &a[0]`.
- Thus, the following are equivalent:
  ```
  pa = &a[0];
  pa = a;
  ```
Penultimate slide on pointers and arrays

- A reference to \( a[i] \) can be written as \(*a+i\)
- A reference to \( &a[i] \) is identical to \( a+i \)
- \( p_a[i] \) is identical to \( *(p_a+i) \)
- Since a pointer is a variable, expressions like \( p_a=a \) and \( p_a++ \) are legal
- Since an array name is not a variable, expressions like \( a=p_a \) and \( a++ \) are illegal
- When an array name is passed to a function, what is passed is the location of the initial element; within the called function, the argument is a local variable; thus, an array name parameter is a pointer
As formal parameters to a function definition, \( s[] \) and \( *s \) are equivalent

If an array name has been passed as the actual argument in a call, the function can believe that it has been handed either an array or a pointer

Part of an array can be passed to a function by passing a pointer to the beginning of the subarray; e.g., \( f(a[2]) \) or \( f(a+2) \)
Address arithmetic

- If $p$ is a pointer to some element of an array, then $p++$ increments $p$ to point to the next element, and $p+=i$ increments it to point $i$ elements beyond the current element.
- There is a distinguished pointer value, `NULL`, which means that the pointer does not point at anything valid; it is defined in `<stdio.h>`.
- Pointer values can be compared using `==`, `!=`, `>`, `>=`, `<`, `<=`.
- A pointer and an integer may be added or subtracted; $p+n$ means the address of the $n^{th}$ object beyond the one $p$ currently points to.
More pointer arithmetic

- Pointer subtraction is valid; if \( p \) and \( q \) point to elements of the same array, and \( p < q \), then \( q - p + 1 \) is the number of elements from \( p \) to \( q \), inclusive.

```c
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;
    while (*p++ != '\0')
        ;
    return p - s;
}
```
More pointer arithmetic

- Valid pointer arithmetic operations are:
  - assignment of pointers of the same type
  - adding or subtracting a pointer and an integer
  - subtracting or comparing two pointers to members of the same array
  - assigning or comparing to NULL

- You CANNOT perform the following operations on pointers:
  - add two pointers
  - multiply, divide, shift, or mask pointers
  - add float or double to pointers
  - assign a pointer of one type to a pointer of another type without an explicit cast
void * pointers

- void * is the generic pointer type
- Any pointer can be cast to void * and back again without loss of information
- void * is used to construct modules that provide generic capabilities at runtime
- Most common initial exposure to void * is through the dynamic memory allocation routines defined in <stdlib.h>
Heap Memory

- Heap memory is allocated on demand
  - Use malloc(), similar to `new` in Java
  - Request a given number of bytes
  - A pointer to the first byte is returned as a void *

- `sizeof(type)` returns the number of bytes in a type; this is a compile-time function; it does not determine the length of a string variable

- Heap memory must be returned when no longer needed.
  - Use free()
  - C does not provide garbage collection.
  - If you do not explicitly free the allocated memory, you will have memory leaks in your program
/*
 * malloc: return a pointer to space for an object of size 'size', or NULL
 *       if the request cannot be satisfied. The space is uninitialized.
 * /
void *malloc(size_t size);

;/*
 * free: deallocates space pointed to by 'p'; it does nothing if 'p'
 *       is NULL. 'p' must be a pointer to space previously allocated by
 *       calloc(), malloc(), or realloc().
 * /
void free(void *p);

/*
 * calloc: returns a pointer to space for an array of 'nobj' objects, each
 *         of size 'size', or NULL if the request cannot be satisfied.
 *         The space is initialized to zero bytes
 * /
void *calloc(size_t nobj, size_t size);
Use of malloc() and free()

- `malloc()` is used in a similar way to `new` in Java – to dynamically allocate memory
- `free()` is used to explicitly return such dynamically allocated memory
- The simple program on the following page reads the first 100 lines from standard input and stores these lines into dynamic memory
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NLINES 100
#define MAXLINESIZE 1024

/* this program reads the first 100 lines from standard input, stores
these lines in dynamic memory, and then frees the dynamic memory */
int main() {
    char *lines[NLINES];
    char buf[MAXLINESIZE];
    char *p;
    int i;
    int nl = 0;

    while(nl < NLINES && fgets(buf, MAXLINESIZE, stdin) != NULL) {
        p = (char *)malloc(strlen(buf)+1); /* leave room for EOS */
        strcpy(p, buf);
        lines[nl++] = p;
    }

    for (i = 0; i < nl; i++)
        printf("%s", lines[i]);
    for (i = 0; i < nl; i++)
        free((void *)lines[i]);
    return 0;
}
```
Character Pointers and Functions

- The most common pointers that you will encounter are pointers to characters.
- String literals are written as: “This is a string”
- The internal representation of the literal is an array of characters, with the array terminated with the null character ‘\0’
- When a string constant is specified as an argument to a function, a pointer to the first character of the constant is passed to the function.
More character pointers …

- Suppose the following declaration:
  ```c
  char *pmessage = "this is a string";
  ```
- This does not cause the string to be copied; `pmessage` is assigned the address of the first character of the string constant
- **C does not provide any operators for processing an entire string as a unit!**
- There is an important difference between these definitions:
  ```c
  char amsg[] = "this is a string";
  char *pmsg = "this is a string";
  ```
- `amsg` is an array, just big enough to hold the sequence of characters and the `\0` that initializes it; individual characters in the array may be changed, but `amsg` always refers to the same storage
More character pointers …

- `pmsg` is a pointer, initialized to point to a string literal; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you attempt to modify the contents of the string literal.

- The following slide shows three different versions of `strcpy`, a function for copying one string to another; each successive version is more succinct, taking fuller advantage of C’s expressiveness.

- The subsequent slide shows two different versions of `strcmp`, a function that compares two strings to each other; again, the 2nd version is more succinct than the first.
strcpy

/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t) {
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}

/* strcpy: copy t to s; pointer version 1 */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++; t++;
    }
}

/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0')
        ;
}
```c
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) { /* array subscript version */
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == \0)
            return 0;
    return s[i] - t[i];
}

/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) { /* pointer version */
    for (; *s == *t; s++, t++)
        if (*s == \0)
            return 0;
    return *s - *t;
}
```
Pointer Arrays – pointers to pointers

• Since pointers are variables themselves, they can be stored in arrays just as other variables can
• As an example, suppose we wish to create a program that will sort text lines
• For fixed-size data types, like integers, we simply need an array of integers; since the lines of text are variable-length, we need an efficient data representation to cope with these variable-length lines
• Therefore, we will create an array of pointers to char, and swap actions invoked as part of the sort algorithm will simply swap the pointers; when finished, if one proceeds linearly through the pointer array, one will have the lines sorted
Pointer arrays and sorting

See section 5.6 on pp 107-110 of the C Programming Language.
Initializing arrays of pointers

- Suppose you wanted to define a list of keywords that your program would understand as commands from a user.
- For example, if you have written a hash table implementation, you might want to write a test program that can be used to exercise the implementation.
- The following declaration shows how you could declare these keywords:

```c
char *keywords[] = {
    "insert",
    "delete",
    "lookup",
    "list",
    NULL
};
```
Arguments to main()

- **main()** has parameters that are provided by the operating system when it is invoked
  ```c
  int main(int argc, char *argv[]);
  ```
- **argv** is an array of pointers to strings
- **argc** is the number of pointers to strings
- **If the invocation of the program was:**
  ```
  ./program joe sventek
  ```
- **Then**
  ```
  argc == 3
  argv[0] → "./program"
  argv[1] → "joe"
  argv[2] → "sventek"
  argv[3] → NULL
  ```
Pointers to functions

- A function itself is not a variable
- It is possible to define pointers to functions
- These can be assigned, placed in arrays, passed to functions, returned by functions, …
- Consider a sort program that sorts strings; sometimes, we want it to sort the strings lexicographically (i.e. as character strings); at other times, there may be a number at the beginning of each line, and we would like the lines to be sorted numerically according to the leading number
- The user should be able to choose which type of sort is desired through a flag in the arguments used to invoke the program
Pointers to functions

- The pseudocode for our main() looks something like the following:

  process command arguments
  read all lines of input
  sort them
  print them in order

- Assuming there is a `sort()` function that performs the “sort them” part of the pseudocode, we need to have some way to inform that function how we want the strings to be compared.
Pointers to functions

- Assume the following declarations in `main()`

```c
char *lineptr[MAXLINES];
void sort(char *lineptr[], int left, int right,
    int (*comp)(char *, char *));
```

- This function prototype says that `sort()` is invoked with an array of pointers to strings, the left and right index in this array over which to sort, and the last formal parameter is a pointer to a function that returns an integer; this function takes two `char *` arguments, and returns `<0, 0, or >0` depending upon whether `arg1 < arg2`, `arg1 == arg2`, or `arg1 > arg2`

- How could we make the signature to “sort” be more general? What impact would it have on code that uses it?
Pointers to functions

• Suppose we have read n lines of text, such that `lineptr[0] ... lineptr[n-1]` have valid pointers. If we wanted to do a lexicographical sort, `main()` would invoke `sort()` as:

```
#include string.h

sort(lineptr, 0, n-1, strcmp);
```

• Recall that the signature for `strcmp()` as defined in `string.h` is

```
int strcmp(const char *s, const char *t);
```
Pointers to functions

• If we wanted to do a numeric sort, we must implement a function that converts the leading number in each line to an integer: consider

```c
#include <stdlib.h>

int numcmp(char *s, char *t) {
    int i1, i2;
    i1 = atoi(s);
    i2 = atoi(t);
    return i1 - i2;
}

• main() would invoke sort() as:

```c
sort(lineptr, 0, n-1, numcmp);
```
Complicated declarations

- Due to the precedence of C’s operators, you must be careful when defining function pointers.
- For example, consider the following function prototype:
  \[
  \text{int } \ast f(\text{void } \ast);
  \]
- This defines \( f \) as a function returning a pointer to an integer;
- Whereas, the following function prototype:
  \[
  \text{int } (\ast pf)(\text{void } \ast);
  \]
- defines \( pf \) as a pointer to a function returning an integer.
ADTs in C
ADT’s in C

- Recall from Java that the specification for an abstract data type (ADT) hides the representation of the data type (via the `private` keyword)
- In C, we hide the representation of an abstract data type by declaring the public type to be
  ```
  struct <name> *
  ```
- In C, we use the `.h` file for the specification, the `.c` file for implementation
- The `.h` file contains
  - Public type and constant declarations
  - Function prototypes for the operations on an instance of the ADT
  - `extern` declarations (if any) for any global data defined in the `.c` file
ADT’s in C (cont)

- Users of the ADT
  - `#include` the `.h` file (to make types, constants, functions, any externs) visible
  - invoke the available functions
  - **NEVER, EVER `#include` a `.c` file!!!**

- The `.c` file contains
  - `#include` of the matching `.h` file (to detect inconsistencies)
  - Other includes for libraries and ADTs needed for the implementation
  - Additional type definitions
  - Implementations of the callable functions
  - Other functions as needed to complete the implementation – these should be declared `static`
Generic container data types

- Recall from CIS 212 that in Java we can define generic classes that are parameterized with respect to types.
- A particular class of generic classes upon which you focussed were containers – e.g. lists, sets, tables, ... that were parameterized with respect to the type of the payload of the data structures in the aggregate data type.

The generic implementation concentrates on navigating through the pointer/control aspect of the data structure, and treats the payload as a “bag of bits”.
Generic containers in C

- The type `void *` is a generic pointer; it can be cast to any other type of pointer, and any other type of pointer can be cast to a `void *`, without loss of information.

- Thus, we can implement generic container data structures in C exploiting `void *` pointers; the “Data” fields below are `void *` pointers.

- Unlike Java, in which a generic class is instantiated at run time via “new”, in C we must instantiate the class at runtime through a function call.
Outline of a generic container interface

```c
#ifndef _FOO_H_
#define _FOO_H_

/* interface definition for generic Foo container */
#include "iterator.h"

typedef struct foo Foo;
Foo *foo_create(/* appropriate arguments */);
void foo_destroy (Foo *f, void (*freeFxn)(void *element));
void foo_purge(Foo *f, void (*freeFxn)(void *element));
int foo_put(Foo *f, void *element);
int foo_get(Foo *f, void **element);
int foo_isEmpty(Foo *st);
long foo_size(Foo *f);
void **foo_toArray(Foo *f, long *len);
Iterator *foo_it_create(Foo *f);
#endif /* _FOO_H_ */
```
What does each line mean?

- `typedef struct foo Foo;` this defines an opaque data type; you will note that the methods on this ADT only ever refer to “Foo *” – “struct foo” will be defined in the implementation

- `Foo *foo_create(/* appropriate arguments */);` this is called to create a new instance of a Foo; the required arguments are specific to the ADT; this method is the equivalent to a Java constructor; if successful, a pointer to the Foo is returned as the value of the function; if it is unsuccessful, NULL will be returned
What does each line mean?

- `void foo_destroy (Foo *f, void (*freeFxn)(void *e));`  
  this destroys the Foo instance; for each element in the Foo, if `freeFxn` != NULL, that function is invoked on that element to return any heap storage associated with the element; then, any heap storage associated with the Foo is returned to the heap.

- `void foo_purge (Foo *f, void (*freeFxn)(void *e));`  
  purges all elements from the Foo; for each element, if `freeFxn` != NULL, that function is invoked on that element to return any heap storage associated with the element; any heap storage associated with the element in the Foo is then returned; upon return, ‘f’ will be empty.
What does each line mean?

- There can be a number of methods for inserting elements in Foo, retrieving elements from the Foo (either destructively or non-destructively); the two examples shown below assume a storage container with destructive retrieval
  - `int foo_put(Foo *f, void *element);`  
    adds an element to the Foo; return value of function is 1/0 if the call was successful/unsuccessful;
  - `int foo_get(Foo *f, void **element);`  
    fetches an element from Foo, returning the element in *element; if successful, function return value is 1; otherwise, it is 0
What does each line mean?

- `int foo_isEmpty(Foo *f);`
  returns true if the Foo is empty, returns false if not

- `long foo_size(Foo *f);`
  returns the number of elements in the Foo

- `void **foo_toArray(Foo *f, long *len);`
  returns an array of pointers to the elements in the Foo in the natural order defined by Foo’s; the number of elements in the array is returned in *len; after the caller has finished using the array of pointers, it should be returned to the heap via a call to free()
What does each line mean?

- `Iterator *foo_it_create(Foo *f);`
  creates a generic iterator for this Foo instance;
  successive calls to `it_next()` on the returned iterator
  will return the elements of the Foo in the natural order
  defined by Foo’s; if unsuccessful, NULL is returned;
  when the caller has finished with the iterator,
  `it_destroy()` must be called on the iterator
#ifndef _ITERATOR_H_
define _ITERATOR_H_

/**
 * interface definition for generic iterator
 *
 * patterned roughly after Java 6 Iterator class
 */

typedef struct iterator Iterator;

Iterator *it_create(long size, void **elements);

int it_hasNext(Iterator *it);

int it_next(Iterator *it, void **element);

void it_destroy(Iterator *it);

#endif /* _ITERATOR_H_ */
iterator.c (1/2)

```c
#include "iterator.h"
#include "stdlib.h"

struct iterator {
    long next;
    long size;
    void **elements;
};

Iterator *it_create(long size, void **elements) {
    Iterator *it = (Iterator *)malloc(sizeof(Iterator));
    if (it != NULL) {
        it->next = 0L;
        it->size = size;
        it->elements = elements;
    }
    return it;
}
```
```c
int it_hasNext(Iterator *it) {
    return (it->next < it->size) ? 1 : 0;
}

int it_next(Iterator *it, void **element) {
    int status = 0;
    if (it->next < it->size) {
        *element = it->elements[it->next++];
        status = 1;
    }
    return status;
}

void it_destroy(Iterator *it) {
    free(it->elements);
    free(it);
}
```
Generic stack – stack.h

#ifndef _STACK_H_
#define _STACK_H_

/*
 * interface definition for generic stack implementation
 *
 * patterned roughly after Java 6 Stack generic class
 */
#include “iterator.h”

typedef struct stack Stack; /* opaque type definition */

Stack *stack_create(long capacity);
void stack_destroy(Stack *st, void (*freeFxn)(void *element));
void stack_purge(Stack *st, void (*freeFxn)(void *element));
int stack_push(Stack *st, void *element);
int stack_pop(Stack *st, void **element);
int stack_peek(Stack *st, void **element);
int stack_isEmpty(Stack *st);
long stack_size(Stack *st);
void **stack_toArray(Stack *st, long *len);
Iterator *stack_it_create(Stack *st);

#endif /* _STACK_H_ */
#include "stack.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 50L
#define MAX_INIT_CAPACITY 1000L

struct stack {
    long capacity;
    long delta;
    long next;
    void **theArray;
};
Stack *stack_create(long capacity) {
    Stack *st = (Stack *)malloc(sizeof(Stack));

    if (st != NULL) {
        long cap;
        void **array = NULL;

        cap = (capacity <= 0) ? DEFAULT_CAPACITY : capacity;
        cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
        array = (void **) malloc(cap * sizeof(void *));
        if (array == NULL) {
            free(st);
            st = NULL;
        } else {
            st->capacity = cap;
            st->delta = cap;
            st->next = 0L;
            st->theArray = array;
        }
    }

    return st;
}
/* traverses stack, calling freeFxn on each element */
static void purge(Stack *st, void (*freeFxn)(void*)) {
    if (freeFxn != NULL) {
        long i;
        for (i = 0L; i < st->next; i++)
            (*freeFxn)(st->theArray[i]); /* user frees element storage */
    }
}

void stack_destroy(Stack *st, void (*freeFxn)(void*)) {
    purge(st, freeFxn);
    free(st->theArray); /* free array of pointers */
    free(st); /* free the Stack struct */
}

void stack_purge(Stack *st, void (*freeFxn)(void*)) {
    purge(st, freeFxn);
    st->next = 0L;
}
int stack_push(Stack *st, void *element) {
    int status = 1;

    if (st->capacity <= st->next) { /* need to reallocate */
        size_t nbytes = (st->capacity + st->delta) * sizeof(void *);
        void **tmp = (void **)realloc(st->theArray, nbytes);
        if (tmp == NULL)
            status = 0; /* allocation failure */
        else {
            st->theArray = tmp;
            st->capacity += st->delta;
        }
    }
    if (status)
        st->theArray[st->next++] = element;
    return status;
}
int stack_pop(Stack *st, void **element) {
    int status = 0;
    if (st->next > 0L) {
        *element = st->theArray[--st->next];
        status = 1;
    }
    return status;
}

int stack_peek(Stack *st, void **element) {
    int status = 0;
    if (st->next > 0L) {
        *element = st->theArray[st->next - 1];
        status = 1;
    }
    return status;
}
```c
int stack_isEmpty(Stack *st) {
    return (st->next == 0L);
}

long stack_size(Stack *st) {
    return st->next;
}

/* local function that duplicates the array of void * pointers on the heap
* returns pointer to duplicate array or NULL if malloc failure */
static void **arraydupl(Stack *st) {
    void **tmp = NULL;
    if (st->next > 0L) {
        size_t nbytes = st->next * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;
            for (i = 0; i < st->next; i++)
                tmp[i] = st->theArray[i];
        }
    }
    return tmp;
}
```
void **stack_toArray(Stack *st, long *len) {
    void **tmp = arraydupl(st);

    if (tmp != NULL)
        *len = st->next;
    return tmp;
}

Iterator *stack_it_create(Stack *st) {
    Iterator *it = NULL;
    void **tmp = arraydupl(st);

    if (tmp != NULL) {
        it = it_create(st->next, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}