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
- Review dynamic memory allocation and the use of pointers in C

jectives
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 variable named `c`, and that 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 $T$, and a pointer will typically occupy $4$ or $8$ bytes, depending upon the memory architecture of your processor.

A typical machine has an array of consecutively numbered memory cells that can be manipulated individually or in contiguous groups — another variable.

A pointer is a variable that contains the address of another variable.
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"
- \( \& \) can only be applied to variables and array elements; it cannot be applied to register variables.
- \( \& \) is said to "point to" \( c \) and is verbalized as "address of".

We make \( p \) point to \( c \) with a

\[ \text{More on Pointers} \]
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
More on Pointers

Note that the declaration for a pointer to an int is:

```c
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, then *p can occur in any context where x could.

What happens for each of the following declarations?

Assume the following declarations:

```c
int x[2] = {1, 9}, *p = &x[0];
```

- Increment p to point to x[1], return its value
- Increment the value of x[0] (now 3), then return its value
- Add 1 to x[0], storing the value in x[1] (now 2)
- Retrieve x[0], add 1, store in y; y now has a value of 2
- Get the value of x[0] = y
- Increment p to point to x[1], return its value

Retrieve x[0], add 1, store in y; y now has a value of 2

What happens for each of the following?

1. `y = *p + 1;`
   - Increment p to point to x[1], return its value (9)
2. `*p += 1;`
   - Return the value of x[0] (3), then increment value of x[0] (now 4)
3. `++*p;`
   - Return the value of x[0] (now 3), then increment value of x[0] (now 4)
4. `*(p++);`
   - Increment p to point to x[1], return its value (9)

Note that the declaration for a pointer to an int is:

```c
int *p;
```
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:

```c
int *pa;
```

This defines an array of array of size 10 — i.e., a block of 10 consecutive int objects named `a[0]`, `a[1]`, ..., `a[9]`.

Assume `pa` is a pointer to an integer, declared as:

```c
int *pa;
```

It refers to the ith element of the array:

```c
a[i]
```

Consider the following declaration:

```c
int a[10];
```

Pointers and arrays are strongly related in C.
More pointers and arrays

Before $pa$

$pa+1$ points to the next element of the array, $pa+i$ points $i$ elements past $pa$, and $pa-i$ points $i$ elements before $pa$.

By definition, $pa+1$ points to the next element of $a$; i.e., $pa$ contains the address of $a[0]$.

The assignment $x = *pa$ copies the contents of $a[0]$ into $x$.

The assignment $pa = a[0]$ causes $pa$ to point to element zero of $a$; i.e., $pa$ contains the value of $a[0]$.
More pointers and arrays

The preceding statements are true regardless of the type or size of the variables in the array.

- The value of a variable or expression of type array is the address of element 0 of the array – i.e. $a[0] == \&a[0]$.
- Thus, the following are equivalent:
  - $pa = a$
  - $pa = \&a[0]$

The meaning of “add 1 to a pointer” and by extension, all pointer arithmetic, is that $pa + 1$ points to the next object beyond $pa$. All pointers to the next object, and $pa + 1$ points to the $i$th object beyond $pa$, where $i$ is the type or size of the variables in the array. The preceding statements are true regardless of

The meaning of “add 1 to a pointer” and by extension, all pointer arithmetic, is that $pa + 1$ points to the next object beyond $pa$. All pointers to the next object, and $pa + 1$ points to the $i$th object beyond $pa$, where $i$ is the type or size of the variables in the array. The preceding statements are true regardless of
name parameter is a pointer argument is a local variable; thus, an array
element within the called function, the
element what is passed is the location of the initial
when an array name is passed to a function,

expressions like \( a = pa \) and \( a++ \) are illegal
since an array name is not a variable,

\( pa = a \) and \( pa++ \) are legal
since a pointer is a variable, expressions like

\( pa \) is identical to \( \tau \) [\( pa + 1 \) \( \tau \) is identical to \( \tau + 1 \)] is identical to \( a + 1 \)

a reference to \( a \) can be written as \( \tau \) [\( \tau + 1 \)]

*arrays*

Penultimate slide on pointers and
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, and if the call is made with an array as the actual argument, it will treat it as the actual argument in a call.

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

\[
f(a+2)
\]

\[
\text{if \( a \) is an array name, then \( f(s) \) is equivalent to passing a pointer to \( s[2] \)}
\]

\[
\text{or \( f(s) \) is equivalent to passing a pointer to \( s[2] \)}
\]

CIS 415, Spring 2018

Heap Review & ADTs

11

```c
int strlen(char *s)
{
    int n;
    for (n = 0; *s++ != \0; n++) ;
    return n;
}
```

/* strlen: return length of string */

Last slide on pointers and arrays
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.

A pointer and an integer may be added or subtracted, their contents, which are addresses in memory.

- There is a distinguished pointer value, \texttt{NULL}, which means that the pointer does not point at anything valid;
- Pointer values can be compared using \texttt{==}, \texttt{!=}, \texttt{>, <, >=, <, =, !, \&, \&=, \&\&, \&\&=, \|, \|=, \|\|, \|\|=, \&\&\&, \&\&\&=, \|\|\|, \|\|\|=, \&\&\&\&, \&\&\&\&=, \|\|\|\|, \|\|\|\|=}, which is defined in \texttt{<stdio.h>};
- \( p+n \) means the address of the \( n \)th object beyond the one \( p \) currently points to.
- When comparing two pointers, you are comparing their contents, which are addresses in memory.
- \( p \) is a pointer to some element of an array, \texttt{arr}[\texttt{n}], which
- The current element it points to point to elements beyond the current element.
- \( p++ \) increments \( p \) to point to the next element, and \( p += i \) increments it to point \( i \) elements beyond the current element.
- If \( p \) is a pointer to some element of an array, then \( p++ \) increments it to point to the next element.
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') {
        char s = *p;
    }
    return p - s;
}
```

Heap Review & ADTs
More Pointer Arithmetic

Valid pointer arithmetic operations are:

- assigning a pointer of one type to a pointer of another type
- adding or subtracting two pointers to members of the same array
- assigning or comparing to NULL
- adding or subtracting a pointer and an integer
- assigning of pointers of the same type

You CANNOT perform the following operations on pointers:

- adding two pointers
- multiplying, dividing, shifting, or masking pointers
- adding float or double to pointers
- assigning a pointer of one type to a pointer of another type without an explicit cast
`<stdlib.h>`

All allocation routines defined in the dynamic memory is through the `void *`.

- The most common initial exposure to `void *` is used to construct modules that provide generic capabilities at runtime.
- `void *` is used to cast to `void *` and back again without loss of information.
- Any pointer can be cast to `void *`.

**Pointers**
Heap memory leaks in your program

- If you do not explicitly free the allocated memory, you will have memory leaks.
- C does **not** provide garbage collection.
- Use `free()` when no longer needed.

Heap memory must be returned when no longer needed.

- A pointer to the first byte is returned as a `void*`.
- Request a given number of bytes in a type; this is a **compile-time function**.
- `sizeof` returns the number of bytes in a type; this is a **compile-time function**.
- `malloc()` returns a pointer to the first byte.
- Use `malloc()` similar to `new` in Java.

Heap memory is allocated on demand.
Function Prototypes

void *malloc(size_t size);
/* The space is uninitialized to zero bytes. The space is not satisfied. */

void free(void *p);
/* free: deallocates space pointed to by 'p'; it does nothing if 'p' is NULL. */

void *calloc(size_t nobj, size_t size);
/* calloc() returns a pointer to space for an array of nobj objects, each of size, or NULL if the request cannot be satisfied. */

void *realloc(void *ptr, size_t size);
/* realloc() returns a pointer to space for an object of size, or NULL if the request cannot be satisfied. */
Use of malloc() and free()

malloc() is used in a similar way to new in Java to dynamically allocate memory explicitly. It 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.
/* this program reads the first 100 lines from standard input, stores
these lines in dynamic memory, and then frees the dynamic memory */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NLINES 100
#define MAXLINESIZE 1024

int main() {
    char *lines[NLINES];
    char *buf[MAXLINESIZE];
    int i; int nl = 0;

    while(nl < NLINES && fgets(buf, MAXLINESIZE, stdin) != NULL) {
        for (i = 0; i < nl; i++) {
            lines[i] = p;
            int i; int nl = 0;
        }
        strcpy(p, buf);
        free((void *)lines[i]);
        return 0;
    }
}
Character Pointers and Functions

character of the constant is passed to the argument to a function, a pointer to the first character of the string constant is specified as an array of characters, with the null character '\0' terminated.

The internal representation of the literal is an array of characters, with the character '0'

When a string constant is specified as an argument to a function, a pointer to the first character of the string constant is passed to the function.

String literals are written as:

"This is a string"

String literals are written as:

encounter are pointers to characters encountered in the string literals that you will

The most common pointers that you will

String literals are written as:

"This is a string"

The internal representation of the literal is an array of characters, with the null character '\0' terminated.

When a string constant is specified as an argument to a function, a pointer to the first character of the string constant is passed to the function.

String literals are written as:

"This is a string"

The most common pointers that you will
More character pointers ...
More character pointers …

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

More character pointers …
```c
void strcpy(char *s, char *t) {
    int i = 0;
    while ((s[i] = t[i]) != '\0') i++;
}

void strcpy(char *s, char *t) {
    while (*s != '\0') { s++; t++; }
}

void strcpy(char *s, char *t) {
    while (*s++ != '\0') { t++; }
}

void strcpy(char *s, char *t) {
    while (t != s) { t++; }
}
```

/* strcpy: copy t to s; array subscript version */
/* strcpy: copy t to s; pointer version 1 */
/* strcpy: copy t to s; pointer version 2 */
/* strcpy: copy t to s; array subscript version */
The function `strcmp` is defined as follows:

```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;
}
```
Since pointers are variables themselves, they can be stored in arrays just as other variables can. Since pointers are variables themselves, they can be stored in arrays just as other variables can.

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

**Pointer arrays and sorting**
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. You might want to write a test for each command. If you have written a hash table user

```
char *keywords[] = {
  "insert",
  "delete",
  "lookup",
  "test",
  NULL
};
```

The following declaration shows how you could declare these keywords:

- **Implementation**: Your program would understand as commands from a user.
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
- `argv[0]` → ./program
- `argv[1]` → joe
- `argv[2]` → sventek
- `argv[3]` → NULL

If the invocation of the program was:

```
./program joe sventek
```

Then:

- `argc == 3`
- `argv[0]` → ./program
- `argv[1]` → joe
- `argv[2]` → sventek
Pointers to functions

The program desired through a flag in the arguments used to invoke the user should be able to choose which type of sort is leading number the lines to be sorted numerically according to the lines at the beginning of each line, and we would like character strings; at other times, there may be a character string (i.e., as we want it to sort the strings lexicographically, i.e., as we want it to sort the strings lexicographically).

Consider a sort program that sorts strings; sometimes, functions, returned by functions, these can be assigned, placed in arrays, passed to it is possible to define pointers to functions.

A function itself is not a variable.
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.

CIS 415, Spring 2018
Heap Review & ADTs

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

```c
#include <string.h>

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

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

```c
int strcmp(const char *s, const char *t);
```

The function `strcmp` compares two strings, returning 0 if they are equal, a negative number if the first string is lexicographically less than the second, and a positive number if the first string is lexicographically greater than the second. The `sort` function sorts an array of pointers according to the values at those pointers, using the `strcmp` function to determine the order.

In other words, a function like `

```c
main()
```

would invoke `sort()` to sort the `lineptr` array lexicographically, based on the content of the lines of text.

Points to functions

---

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 the following definition:

```c
#include <stdlib.h>

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

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 the following definition:

```c
#include <stdlib.h>

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

To do this, we can define a function `numcmp` that takes two character pointers and returns an integer. This function uses `atoi` to convert the characters to integers and then returns the difference between the two integers.

In main, we can invoke `sort` as follows:

```c
main()
{
    sort(lineptr, 0, n-1, numcmp);
}
```

This would sort the lines in the file by the leading number in each line.
Due to the precedence of C's operators, you must be careful when defining function pointers. For example, consider the following function prototype:

```c
int (*pf)(void *);
```

This defines `pf` as a pointer to a function returning an integer to an integer;

Whereas, the following function prototype:

```c
int *f(void *);
```

This defines `f` as a function returning a pointer to an integer;

Due to the precedence of C's operators, you must be careful when defining function pointers.

Complicated declarations
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 (via the `private` keyword), and we use the `.h` file for the specification and 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

In C, we use the `.h` file for the specification, the `.c` file contains:

- Structure declaration:

```c
struct <name> *<name>
```

- Type by declaring the public type to be

In C, we hide the representation of an abstract data type (ADT) (via the `private` keyword) the `ADT` hides the representation of the data type. Recalling from Java that the specification for an abstract data type: ADTs in C.
Users of the ADT

- #include the .h file (to make types, constants, functions, any externs visible)
- invoke the available functions
- Other functions as needed to complete the implementation
- Implementations of the callable functions
- Additional type definitions
- Implementations
- Other includes for libraries and ADTs needed for the ADT
- #include of the matching .h file (to detect inconsistencies)

The .c file contains

- NEVER EVER #include a .c file!!!
- Implementations of the callable functions
- Additional type definitions
- Implementations
- Other includes for libraries and ADTs needed for the ADT
- #include of the matching .h file (to make types, constants, functions, any externs visible)

ADTs in C (cont)
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 focused was containers – e.g., lists, sets, tables, … that were containers of elements of a particular 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”. Data structures in the aggregate data type parameterized with respect to the type of the payload of the data structure in the aggregate data type.
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. Unlike Java, in which a generic class is instantiated at runtime, in C we must instantiate the class at runtime via "new", in C the `void *` pointer 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 by exploiting `void *` pointers, the "Data" fields below are pointers to `void *` pointers.
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
{
    void *self;
    void (*destroy)(const Foo *f, void (*freeFxn)(void *element));
    void (*clear)(const Foo *f, void (*freeFxn)(void *element));
    int (*put)(const Foo *f, void *element);
    int (*get)(const Foo *f, void **element);
    int (*isEmpty)(const Foo *st);
    long (*size)(const Foo *f);
    void **(*toArray)(const Foo *f, long *len);
    const Foo *(*itCreate)(const Foo *f);
} Foo;

const Foo *Foo_create(/* appropriate arguments */);

#endif /* _FOO_H_ */
```

Heap Review & ADTs
What does each line mean?
Method Signatures

```c
void (*destroy) (const Foo *f, void (*freeFxn)(void *e));
```

This destroys the `Foo` instance; for each element in the
`Foo`, any heap storage associated with the element in that
element is returned.

```c
void (*clear) (const Foo *f, void (*freeFxn)(void *e));
```

This purges all elements from the `Foo`; for each element, if
`freeFxn` is `NULL`, that function is invoked on that
that element to return any heap storage associated with
the element; then, any heap storage associated with the
`Foo` is returned to the heap.

```c
void (destroy)(const Foo *foo, void (*freeFxn)(void *e));
```

This destroys the `Foo` instance; for each element in the
`Foo`, any heap storage associated with the element in that
element is returned.

This purges all elements from the `Foo`; for each element, if
`freeFxn` is `NULL`, that function is invoked on that
that element to return any heap storage associated with the
`Foo` is returned to the heap.
What does each line mean?
What does each line mean?

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

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

`void **(*toArray)(const 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, the caller should return it to the heap via a call to `free()`.

`CIS 415, Spring 2018
Heap Review & ADTs
44`
What does each line mean?

`const Iterator *(*itCreate)(const Foo *f);`

creates a generic iterator for this `Foo` instance; successive calls to the `next()` method on the 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, the caller must invoke the `destroy()` method on the iterator.
/* iterator.h */

const Iterator *Iterator_create(size, void **elements);

typedef struct iterator Iterator;

struct iterator {
    void *self;
    int (*hasNext)(const Iterator *it);
    int (*next)(const Iterator *it, void **element);
    void (*destroy)(const Iterator *it);
};

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

#define _ITERATOR_H_

Generic Iterator — Iterator
typedef struct it_data {
    long next;
    long size;
    void **elements;
} ItData;

static int it hasNext(Iterator *it) {
    ItData *itd = (ItData *)it->self;
    return (itd->next < itd->size) ? 1 : 0;
}

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

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

    if (itd->next < itd->size) {
        *element = itd->elements[itd->next++];
        status = 1;
    }
    return status;
}

static void it_destroy(Iterator *it) {
    ItData *itd = (ItData *)it->self;
    free(itd->elements);
    free(itd);
    free((void *)it);
}

static Iterator template = {
    NULL,
    it_hasNext,
    it_next,
    it_destroy,
};
const Iterator * Iterator_create(long size, void **elements) {
    Iterator *it = (Iterator *)malloc(sizeof(Iterator));
    if (it != NULL) {
        ItData *itd = (ItData *)malloc(sizeof(ItData));
        if (itd != NULL) {
            it->next = 0L;
            it->size = size;
            it->elements = elements;
            it->self = itd;
        } else {
            free(it);
            it = NULL;
        }
    }
    return it;
}
Using an Iterator

```c
const Iterator *it;

void *element;

/* obtain iterator using the Factory method in another ADT */

using an iterator
```
```c
#ifndef _STACK_H_
#define _STACK_H_

#include "iterator.h"

/* needed for factory method */
typedef struct stack
    Stack;

/* forward reference */
const Iterator *(*itCreate)(const Stack *st);

struct stack {
    void *self;
    void (*destroy)(const Stack *st, void (*userFxn)(void *element));
    void (*clear)(const Stack *st, void (*userFxn)(void *element));
    int (*push)(const Stack *st, void *element);
    int (*pop)(const Stack *st, void **element);
    int (*peek)(const Stack *st, void **element);
    int (*isEmpty)(const Stack *st);
    void **(*toArray)(const Stack *st, long *len);
    const Iterator *(*itCreate)(const Stack *st);
};

#endif /* _STACK_H_ */
```

Generic Stack – stack.h
```c
#include "stack.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 50L
#define MAX_INIT_CAPACITY 1000L

typedef struct st_data {
    void **theArray;
    long next;
    long capacity;
} StData;

#define INIT_CAPACITY 100
#define DEFAULT_CAPACITY 50L

#include <stdlib.h>
#include "stack.h"

stack.c (1/7)
```
/* helper fxn, traverses stack, calling freeFxn on each element */

static void purge(StData *std, void (*freeFxn)(void *)) {
    if (freeFxn != NULL) {
        long i;
        for (i = 0L; i < std->next; i++)
            (*freeFxn)(std->theArray[std->next]);
    }
}

static void st_destroy(const Stack *st, void (*freeFxn)(void *)) {
    StData *std = (StData *)st->self;
    purge(std, freeFxn);
    free((void *)std->theArray);
    free((void *)std->next);
    free((void *)std->self);
}

static void st_clear(const Stack *st, void (*freeFxn)(void *)) {
    StData *std = (StData *)st->self;
    purge(std, freeFxn);
    std->next = 0L;
}

/* user frees elem storage */

// free dispatch table
free((void *)(st->self));
free((void *)(st->theArray));
free((void *)(st->next));
free((void *)(st->self));
static int st_push(const Stack *st, void *element) {
    StData *std = (StData *)st->self;
    int status = 1;
    if (std->theArray[std->next++] = element) {
        return status;
    } else {
        status = 0;
        size_t nbytes = 2 * (std->capacity * sizeof(void *)) + 1;
        void **tmp = realloc((void **)std->theArray, nbytes);
        if (tmp == NULL) status = 0;
        else std->theArray = tmp;
    }
}
static int st_pop(const Stack *st, void **element) {
    StData *std = (StData *)st->self;
    int status = 0;
    if (std->next > 0L) {
        *element = std->theArray[std->next - 1];
        if (std->next < 0L) {
            status = 1;
        }
    }
    return status;
}

static int st_peek(const Stack *st, void **element) {
    StData *std = (StData *)st->self;
    int status = 0;
    if (std->next > 0L) {
        *element = std->theArray[std->next - 1];
        if (std->next < 0L) {
            status = 1;
        }
    }
    return status;
}
```c
static long st_size(const Stack *st) {
    StData *std = (StData *)st;
    for (int i = 0; i < std->next; i++)
        if (tmp != NULL)
            tmp = malloc(sizeof(void *) * std->next);
    return std->next;
}

static int st_empty(const Stack *st) {
    StData *std = (StData *)st;
    return std->next == 0;
}
```

/* helper function - duplicates array of void * pointers on the heap */

```c
static void **arrayDupl(StData *std) {
    void **tmp = NULL;
    if (std->next > 0) {
        size_t nbytes = std->next * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;
            for (i = 0; i < std->next; i++)
                tmp[i] = std->theArray[i];
        }
    }
    return tmp;
}
```
static void **
st_toArray
( const Stack *st,
long *len ) {

    StData *
    std = ( StData * )
st->self;

    void **
tmp =
arrayDupl( std );

    if ( tmp != NULL ) {

        *len =
std->next;

        return tmp;
    }

    free( tmp );

    if ( std == NULL )

        tmp =
iterator_create( std->next, tmp );

    if ( tmp == NULL )

        free( std );

    const Iterator *it =
std->create( std->self );

    const Iterator *
    iter =
iterator_create( std->next, tmp );

    if ( iter == NULL )

        free( tmp );

    return iter;
}

static Stack template = { NULL,
st_destroy,
st_clear,
st_push,
st_pop,
st_peek,
st_size,
st_isEmpty,
st_toArray,
st_itCreate };
Stack creation function:

```c
const Stack *Stack_create(long capacity) {
    Stack *st = (Stack *)malloc(sizeof(Stack));
    if (st != NULL) {
        StData *std = (StData *)malloc(sizeof(StData));
        if (std != NULL) {
            long cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
            cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
            void **array = (void **)malloc(cap * sizeof(void *));
            if (array != NULL) {
                std->capacity = cap;
                std->next = 0L;
                std->theArray = array;
                *st = template;
                st->self = std;
            } else {
                free(std);
                free(st);
                st = NULL;
            }
        } else {
            free(st);
            st = NULL;
        }
    } else {
        free(st);
        st = NULL;
    }
    return st;
}
```
Write a program that checks whether a string of brackets is well-formed.

Each string contains only the characters `[ ] `{ } `<.

The following criteria:

- Each bracket is matched – i.e., for every open bracket (, [, {, < there is a corresponding closing bracket } ), ] }, }.
- The substring contained within each matched bracket pair is also well-formed.
- The substring contained within each matched pair is also well-formed.

Write a program that checks whether a string of brackets is well-formed.
An application to use the stack

The general approach is as follows:

- Read the number of strings that we must check for well-formedness from stdin
- Read each line from stdin and remove the newline character
- For each character in the line:
  - If it is an opening bracket, push that character on the stack
  - If it is a closing bracket, push that character on the stack
  - See if the top item on the stack is the corresponding opening bracket; if so, pop it off
- Create an empty stack, and we stop processing the string and continue to the next character
- If the string is well-formed, we will have processed the entire string and the stack will be empty, in which case we print 'YES'
- Otherwise, we print 'NO'
- Otherwise, we print 'NO', on stdout
#include "stack.h"
#include <stdio.h>
#define UNUSED __attribute__((unused))

char *open = "[\{"
char *close = "]\}>

long strindex(char s[], int c) {
    long i;
    for (i = 0; s[i] != '\0'; i++)
        if (s[i] == c)
            return i;
    return -1;
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    const Stack *st;
    int nlines;
    char buf[1024];
    fgets(buf, sizeof(buf), stdin);
    sscanf(buf, "%d\n", &nlines);
    while (nlines-- > 0)
        for (int i = 0; s[i] = '\0', i++)
            long strindex(char s[]); /* unfinished */

#define UNUSED Attribute
#define UNUSED Attribute
#include <stdio.h>
#include "stack.h"

brackets.c (1/2)
{ return

    if (st->destroy(st, NULL)
        printf("NO\n")
           st = stack_create(NULL);
    else
        printf("YES\n")

    if (!st->isEmpty(st))
        wellformed = 0;
    if (wellformed)
        printf("YES\n");
    else
        printf("NO\n");

    st->destroy(st, NULL);
}

return 0;