Heap Review & Abstract Data Types

Joe Sventek

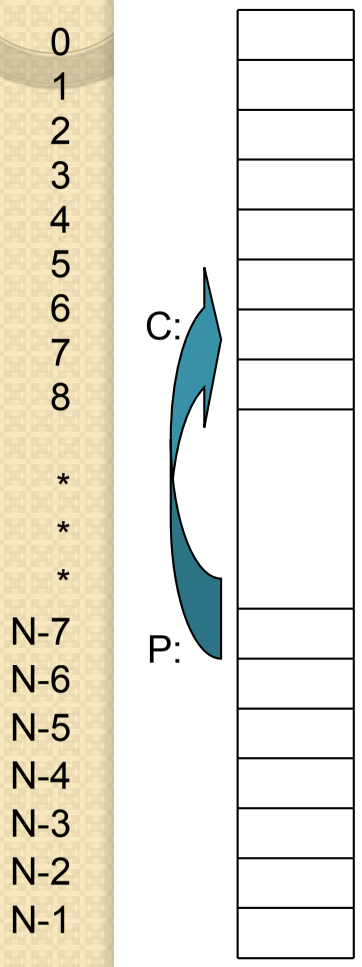
Objectives

- 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 .. N-1`
- Suppose that we have a `char` variable named `c`, and that is 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 access the object the pointer points to
- Consider the following artificial sequence of statements showing the use of `&` and `*`

```
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:

```
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

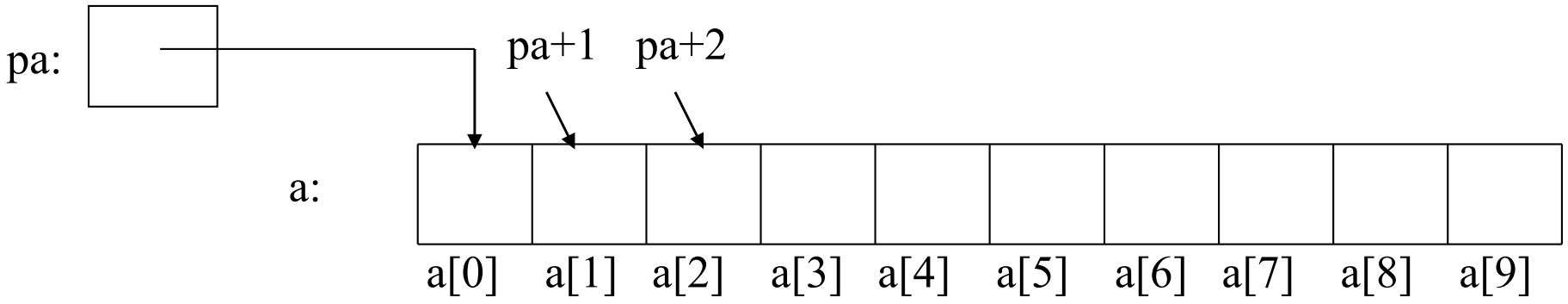
- Pointers and arrays are strongly related in C
- Any operation that can be achieved by array subscripting can also be done with pointers
- Consider the following declaration

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

- This defines an array `a` of size 10 – i.e. a block of 10 consecutive `int` objects named `a[0]`, `a[1]` , …, `a[9]`
- `a[i]` refers to the i^{th} element of the array
- assume `pa` is a pointer to an integer, declared as

```
int *pa;
```

More pointers and arrays



- the assignment `pa = &a[0];`
  causes `pa` to point to element zero of `a`; i.e. `pa` contains the address of `a[0]`
- the assignment `x = *pa;`
  copies the contents of `a[0]` into `x`
- **by definition**, `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`

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`
- `pa[i]` is identical to `*(pa+i)`
- Since a pointer is a variable, expressions like `pa=a` and `pa++` are legal
- Since an array name is ***not*** a variable, expressions like `a=pa` 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

Last slide on pointers and arrays

```
/* strlen: return length of string */
int strlen(char *s)
{
        int n;

        for (n = 0; *s++ != '\0'; n++)
                ;
        return n;
}
```

- 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 `==`, `!=`, `>`, `>=`, `<`, `<=` ; when comparing two pointers, you are comparing their contents, **which are addresses in memory**.
- 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

```
/* 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
- The 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

Function prototypes

```
/*
 * 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:
  `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 in read-only memory
- **C does not provide any operators for processing an entire string as a unit! Become familiar with the functions defined in <string.h>!**
- There is an important difference between these definitions:
  ```
  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')
                ;
}
```

strcmp

```
/* 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;

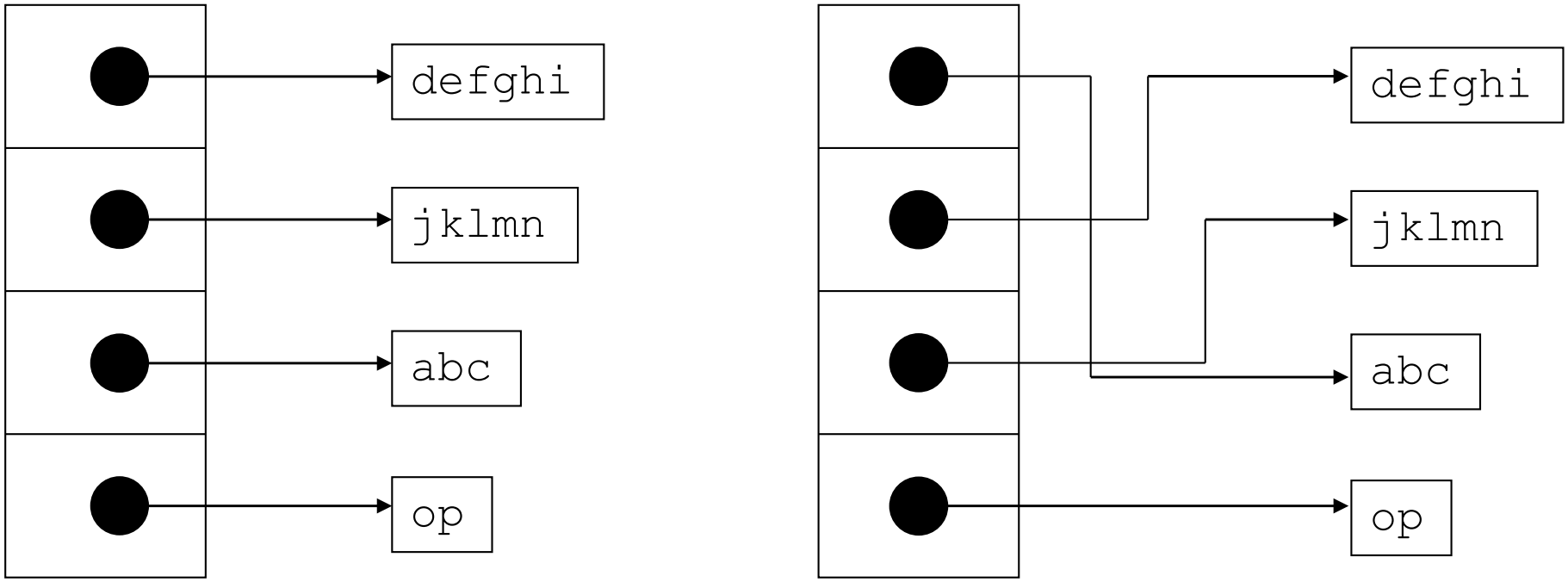
        for (i = 0; s[i] == t[i]; i++)
                if (s[i] == '\0')
                        return 0;
        return s[i] - t[i];
}
```

```
/* 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:

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

Arguments to main()

- main() has parameters that are provided by the operating system when it is invoked

```
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()`

```
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

```
#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:

```
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:

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

- This defines `f` as a function returning a pointer to an integer;
- Whereas, the following function prototype:

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

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

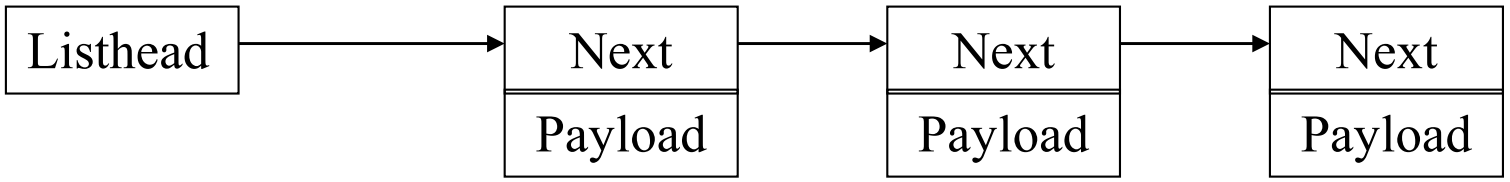
  `struct <name> *`
- In C, we use the `.h` file for the specification, the `.c` file for implementation
- The `.h` file contains
  - Public type and constant declarations
  - Function prototypes for the operations on an instance of the ADT
  - extern declarations (if any) for any global data defined in 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

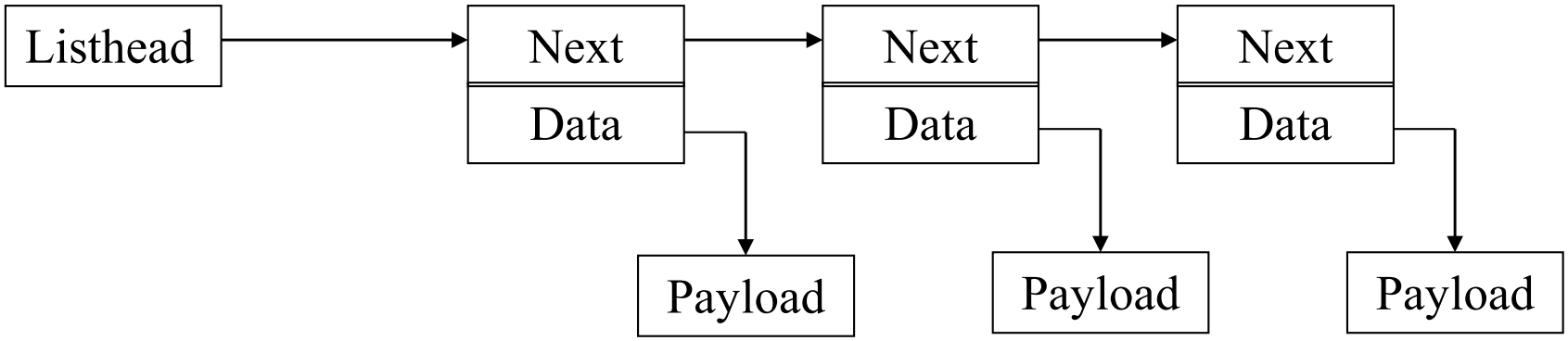
- Recall from CIS 212 that in Java we can define generic classes that are parameterized with respect to types
- A particular class of generic classes upon which you focussed were containers – e.g. lists, sets, tables, … that were parameterized with respect to the type of the payload of the data structures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

```
#ifndef _FOO_H_
#define _FOO_H_
/* interface definition for generic Foo container */
#include “iterator.h”
typedef struct foo Foo;    /* forward reference */
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 Iterator *(*itCreate)(const Foo *f);
};
const Foo *Foo_create(/* appropriate arguments */);

#endif /* _FOO_H_ */
```

What does each line mean?

- ```
  typedef struct foo Foo;  /* forward reference */
  struct foo {
      void *self;
      /* method signatures */
  };
  ```
  This defines the dispatch table and a **`void *`** member for instance-specific data; you will note that the constructor and the methods on this ADT only ever refer to **`const Foo *`**
- **`const 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 `const` pointer to the `Foo` is returned as the value of the function; if it is unsuccessful, `NULL` will be returned
  ```

Method signatures

- **`void (*destroy) (const 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 (*clear)(const 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 into a Foo and retrieving elements from a Foo (either destructively or non-destructively); the two examples shown below assume a storage container with destructive retrieval**
- **`int (*put)(const Foo *f, void *element);`** adds an element to the `Foo`; return value of the function is **1/0** if the call was successful/unsuccessful;
- **`int (*get)(const 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 (*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()`

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 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, the caller must invoke the `destroy()` method on the iterator

Generic iterator – iterator.h

```
#ifndef _ITERATOR_H_
#define _ITERATOR_H_

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

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);
};

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

#endif /* _ITERATOR_H_ */
```

iterator.c (1/3)

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

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

static int it_hasNext(const Iterator *it) {
    ItData *itd = (ItData *)it->self;

    return (itd->next < itd->size) ? 1 : 0;
}
```

iterator.c (2/3)

```
static int it_next(const 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;
}

static void it_destroy(const 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
};
```

iterator.c (3/3)

```
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) {
            itd->next = 0L;
            itd->size = size;
            itd->elements = elements;
            *it = template;
            it->self = itd;
        } else {
            free(it);
            it = NULL;
        }
    }
    return it;
}
```

Using an iterator

```
const Iterator *it;
void *element;

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

while (it->hasNext(it)) {
    it->next(it, &element);
    /* use element, suitably cast */
}
it->destroy(it);
```

Generic stack – stack.h

```
#ifndef _STACK_H_
#define _STACK_H_

#include "iterator.h"                /* needed for factory method */

typedef struct stack Stack;          /* forward reference */

const Stack *Stack_create(long capacity);

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);
    long (*size)(const Stack *st);
    int (*isEmpty)(const Stack *st);
    void **(*toArray)(const Stack *st, long *len);
    const Iterator *(*itCreate)(const Stack *st);
};

#endif /* _STACK_H_ */
```

stack.c (1/7)

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

#define DEFAULT_CAPACITY 50L
#define MAX_INIT_CAPACITY 1000L

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

stack.c (2/7)

```
/* 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[i]);  /* user frees elem storage */
    }
}

static void st_destroy(const Stack *st, void (*freeFxn)(void *element)) {
    StData *std = (StData *)st->self;
    purge(std, freeFxn);
    free(std->theArray);          /* free array of pointers */
    free(std);                    /* free structure with instance data */
    free((void *)st);             /* free dispatch table */
}

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

stack.c (3/7)

```
static int st_push(const Stack *st, void *element) {
    StData *std = (StData *)st->self;
    int status = 1;

    if (std->capacity <= std->next) {       /* need to reallocate */
        size_t nbytes = 2 * std->capacity * sizeof(void *);
        void **tmp = (void **)realloc(std->theArray, nbytes);

        if (tmp == NULL)
            status = 0;                     /* allocation failure */
        else {
            std->theArray = tmp;
            std->capacity nbytes;
        }
    }
    if (status)
        std->theArray[std->next++] = element;
    return status;
}
```

stack.c (4/7)

```
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];
        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];
        status = 1;
    }
    return status;
}
```

stack.c (5/7)

```
static long st_size(const Stack *st) {
    StData *std = (StData *)st->self;
    return std->next;
}

static int st_isEmpty(const Stack *st) {
    StData *std = (StData *)st->self;
    return (std->next == 0L);
}

/* helper function - duplicates array of void * pointers on the heap */
static void **arrayDupl(StData *std) {
    void **tmp = NULL;
    if (std->next > 0L) {
        size_t nbytes = std->next * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;
            for (i = 0L; i < std->next; i++)
                tmp[i] = std->theArray[i];
        }
    }
    return tmp;
}
```

stack.c (6/7)

```
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;
}

static const Iterator *st_itCreate(const Stack *st) {
    StData *std = (StData *)st->self;
    const Iterator *it = NULL;
    void **tmp = arrayDupl(std);
    if (tmp != NULL) {
        it = Iterator_create(std->next, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}

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

stack.c (7/7)

```
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;
            void **array = NULL;
            cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
            cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
            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;
        }
    }
    return st;
}
```

An application to use the stack

- Write a program that checks whether a string of brackets is well-formed.
- Each string contains only the characters []({}<>
- A string is well-formed *iff* it meets 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 pair is also 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-formed-ness from stdin
  - Read each line from stdin and remove the newline character
  - Create an empty stack
  - For each character in the line
    - If it is an opening bracket, push that character on the stack
    - If it is a closing bracket
      - See if the top item on the stack is the corresponding opening bracket; if so, pop it off the stack and continue to the next character
      - If it is not the corresponding bracket, or if the stack is empty, the string is not well-formed, and we stop processing
  - 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' on stdout
  - Otherwise, we print 'NO'

brackets.c (1/2)

```
#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) {
        int i, wellformed = 1;
```

brackets.c (2/2)

```
        st = Stack_create(0L);
        fgets(buf, sizeof buf, stdin);
        for (i = 0; buf[i] != '\n' && buf[i] != '\0'; i++) {
            int c = buf[i];
            long l = strindex(open, c);
            if (l > -1)
                st->push(st, (void *)l);
            else {
                long j = strindex(close, c);
                int stat = st->peek(st, (void **)&l);
                if (stat && l == j)
                    st->pop(st, (void **)&l);
                else {
                    wellformed = 0;
                    break;
                }
            }
        }
        if (! st->isEmpty(st))
            wellformed = 0;
        if (wellformed)
            printf("YES\n");
        else
            printf("NO\n");
        st->destroy(st, NULL);
    }
    return 0;
}
```