# Abstract Data Types (ADTs), After More on the Heap

Prof. David August

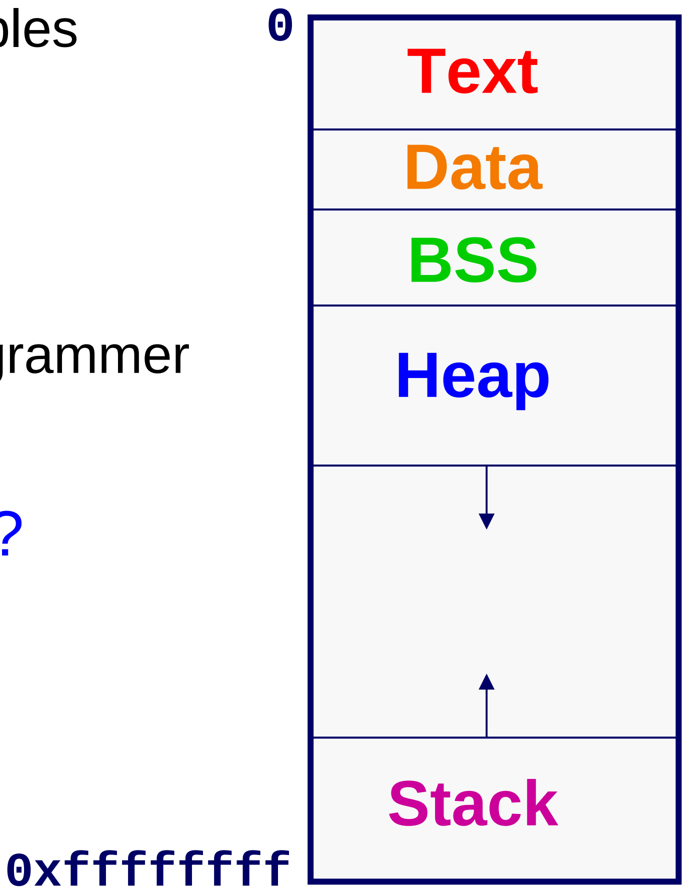
COS 217

# Preparing for the Midterm Exam

- Exam logistics
  - Date/time: Thursday October 26 in lecture
  - Open books, open notes, open mind, but not open laptop/PDA
  - Covering material from lecture, precept, and reading, but not tools
- Preparing for the midterm
  - Lecture and precept materials available online
  - Course textbooks, plus optional books on reserve
  - Office hours and the course listserv
  - Old midterm exams on the course Web site

# A Little More About the Heap…

- Memory layout of a C program
  - Text: code, constant data
  - Data: initialized global & static variables
  - BSS: uninitialized global & static variables
  - Heap: dynamic memory
  - Stack: local variables
- Purpose of the heap
  - Memory allocated explicitly by the programmer
  - Using the functions **`malloc`** and **`free`**
- But, why would you ever do this???
  - Glutton for punishment???



# Example: Read a Line (or URL)

- Write a function that reads a word from stdin
  - Read from stdin until encountering a space, tab, ‘\n’, or EOF
  - Output a pointer to the sequence of characters, ending with ‘\0’
- Example code (very, very buggy)

```
#include <stdio.h>

int main(void) {
   char* buf;

   scanf(“%s”, buf);
   printf(“Hello %s\n”, buf);
   return 0;
}
```

# Problem: Need Storage for String

- Improving the code
  - Allocate storage space for the string
  - Example: define an array
- Example (still somewhat buggy)

```
#include <stdio.h>

int main(void) {
   char buf[64];

   scanf(“%s”, buf);
   printf(“Hello %s\n”, buf);
   return 0;
}
```

# Problem: Input Longer Than Array

- Improving the code
  - Don’t allow input that exceeds the array size
- Example (better, but not perfect)

```
#include <stdio.h>

int main(void) {
    char buf[64];

    if (scanf(“%63s”, buf) == 1)
        printf(“Hello %s\n”, buf);
    else
        fprintf(stderr, Input error\n”);
    return 0;
}
```

# Problem: How *Much* Storage?

- Improving the code
  - Finding out how much space you need from the user
  - Allocate exactly that much space, to avoid wasting
- Beginning of the example (is this really better?)

```
int main(void) {
    int n;
    char* buf;

    printf(“Max size of word: “);
    scanf(“%d”, &n);

    buf = malloc((n+1) * sizeof(char));
    scanf(“%s”, buf);
    printf(“Hello %s\n”, buf);
    return 0;
}
```

# Really Solving the Problem

- Remaining problems
  - User can’t input long words
  - Storage wasted on short words
- But, how do we proceed?
  - Too little storage, and we’ll run pass the end or have to truncate
  - Yet, we don’t *know* how big the word might be
- The gist of a solution
  - Pick a storage size (“line_size”) and read up to that length
  - If we stay within the limit, we’re done
  - If the user input exceeds the space, we can
    - – Allocate space for another line, and keep on reading
    - – At the end, allocate one big buffer and copy all the lines into it

# Abstract Data Types (ADTs)

# Abstract Data Type (ADT)

- An ADT module provides:
  - Data type
  - Functions that operate on the type
- Client does not manipulate the data representation directly
  - The client should just call functions
- "Abstract" because the observable results (obtained by client) are independent of the data representation
- Programming language support for ADT
  - Ensure that client cannot possibly access representation directly
  - C++, Java, other object-oriented languages have *private* fields
  - C has opaque pointers

# An ADT Example: Stacks

- LIFO:  Last-In, First-Out
- Like the stack of trays at the cafeteria
  - “Push” a tray onto the stack
  - “Pop” a tray off the stack
- Useful in many contexts

# Stack Interface (stack.h)

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

typedef struct Item *Item_T;
typedef struct Stack *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, Item_T item);
extern Item_T Stack_pop(Stack_T stk);

#endif
```

What's this for?

# Notes on `stack.h`

- Type **`Stack_T`** is an opaque pointer
  - Clients can pass **`Stack_T`** around but can't look inside
- Type **`Item_T`** is also an opaque pointer
  - … but defined in some other ADT
- **`Stack_`** is a disambiguating prefix
  - A convention that helps avoid name collisions

# Stack Implementation: Array

**stack.c**

```
#include <assert.h>
#include <stdlib.h>
#include “stack.h”

enum {CAPACITY = 1000};

struct Stack {
    int count;
    Item_T data[CAPACITY];
};

Stack_T Stack_new(void) {
  Stack_T stk = malloc(sizeof(*stk));
  assert(stk != NULL);
  stk->count = 0;
  return stk;
}
```

# Careful Checking With Assert

**stack.c**

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

enum {CAPACITY = 1000};

struct Stack {
    int count;
    Item_T data[CAPACITY];
 };

Stack_T Stack_new(void) {
  Stack_T stk = malloc(sizeof(*stk));
  assert(stk != NULL);
  stk->count = 0;
  return stk;
}
```

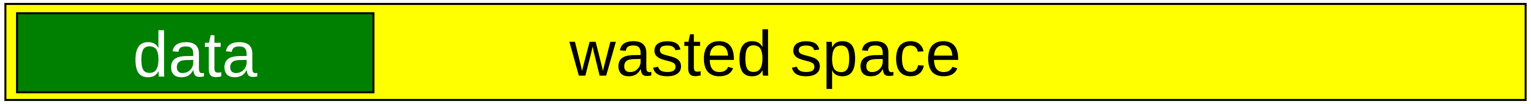
*Make sure stk!=NULL, or halt the program!*

# Stack Implementation: Array (Cont.)

```
int Stack_empty(Stack_T stk) {
  assert(stk != NULL);
  return (stk->count == 0);
}
void Stack_push(Stack_T stk, Item_T item) {
  assert(stk != NULL);
  assert(stk->count < CAPACITY);
  stack->data[stack->count] = item;
  stack->count++;
}
Item_T Stack_pop(Stack_T stk) {
  assert(stk != NULL);
  assert(stk->count > 0);
  stk->count--;
  return stk->data[stk->count];
}
```

# Problems With Array Implementation

**`CAPACITY`** too large: waste memory

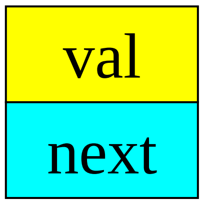



**`CAPACITY`** too small:


assertion failure (if you were careful)

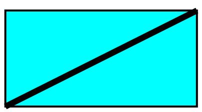
buffer overrun (if you were careless)

# Linked List Would be Better…




```
struct Stack {
        int val;
        struct Stack *next;
} *head;
```

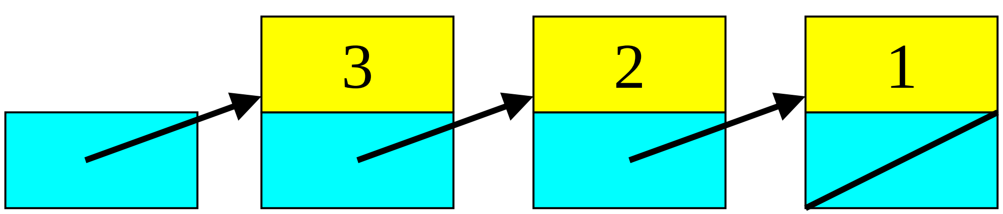
head



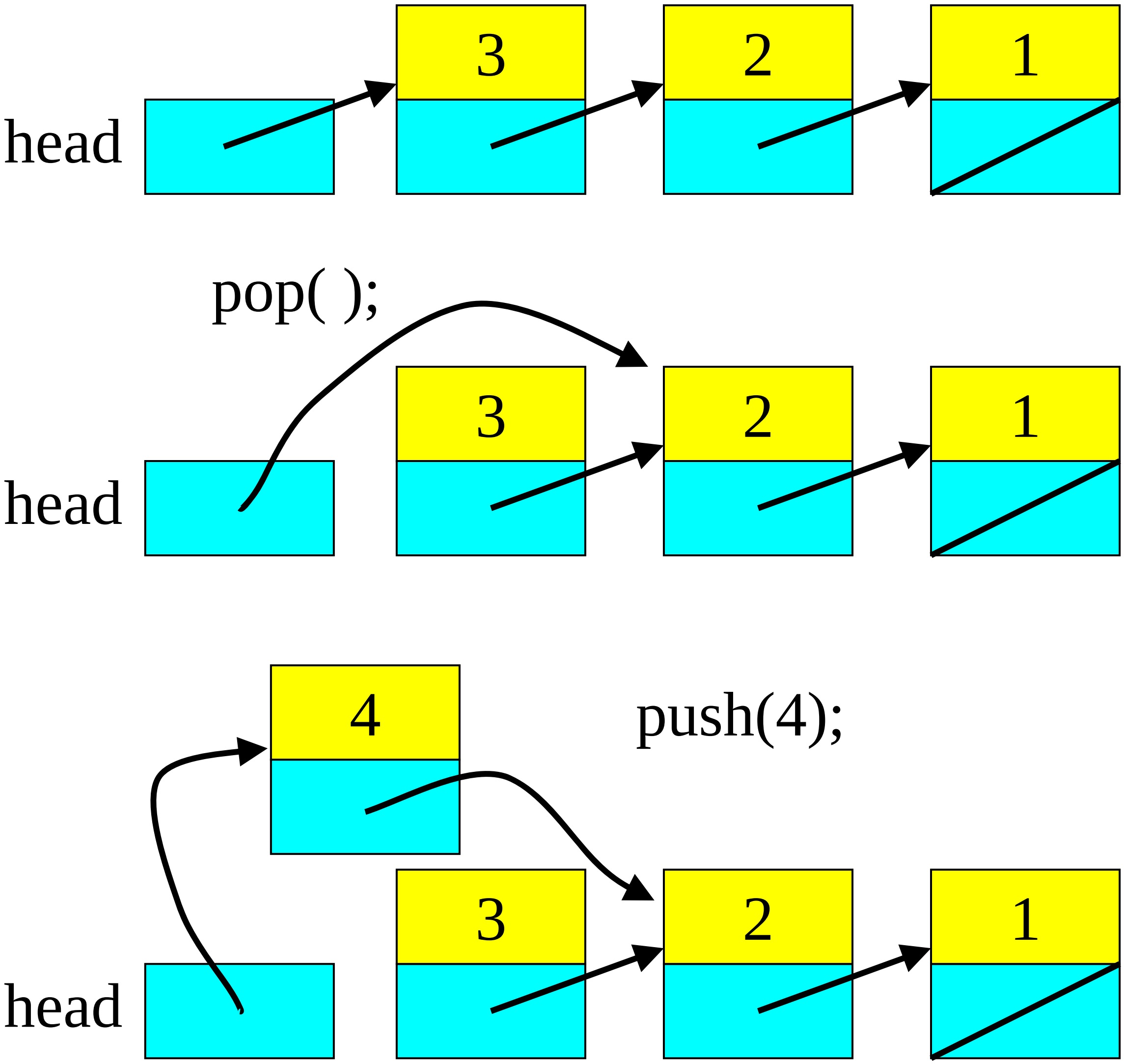
empty stack

push(1); push(2); push(3);

head



# Popping and Pushing



# Stack Implementation: Linked List

## stack.c

```
#include <assert.h>
#include <stdlib.h>
#include “stack.h”

struct Stack {struct List *head;};
struct List {Item_T val; struct List *next;};

Stack_T Stack_new(void) {
  Stack_T stk = malloc(sizeof(*stk));
  assert(stk != NULL);
  stk->head = NULL;
  return stk;
}
```

# Stack Implementation: Linked List

```
int Stack_empty(Stack_T stk) {
  assert(stk != NULL);
  return (stk->head == NULL);
}

void Stack_push(Stack_T stk, Item_T item) {
  Stack_T t = malloc(sizeof(*t));
  assert(t != NULL);
  assert(stk != NULL);
  t->val = item;
  t->next = stk->head;
  stk->head = t;
}
```

Draw pictures of
these data structures!

# `stack.c`, continued

```
Item_T Stack_pop(Stack_T stk) {
  Item_T x;
  struct List *p;
  assert(stk != NULL);
  assert(stk->head != NULL);
  x = stk->head->val;
  p = stk->head;
  stk->head = stk->head->next;
  free(p);
  return x;
}
```

Draw pictures of these data structures!

# Client Program: Uses Interface

**client.c**

```
#include <stdio.h>
#include <stdlib.h>
#include “item.h”
#include “stack.h”

int main(int argc, char *argv[]) {
   int i;
   Stack_T s = Stack_new();
   for (i = 1; i < argc; i++)
      Stack_push(s, Item_new(argv[i]));
   while (!Stack_empty(s))
      Item_print(Stack_pop(s));
   return 0;
}
```

# Problem: Multiple Kinds of Stacks?

- Good, but still not flexible enough
  - What about a program with multiple kinds of stacks
  - E.g., a stack of books, and a stack of pancakes
  - But, can you can only define Item_T once
- Solution in C, though it is a bit clumsy
  - Don’t define Item_T (i.e., let it be a “void *”)
  - Good flexibility, but you lose the C type checking

```
typedef struct Item *Item_T;
typedef struct Stack *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, void *item);
extern void *Stack_pop(Stack_T stk);
```

# Conclusions

- Heap
  - Memory allocated and deallocated by the programmer
  - Useful for making efficient use of memory
  - Useful when storage requirements aren't known in advance
- Abstract Data Types (ADTs)
  - Separation of interface and implementation
  - Don't even allow the client to manipulate the data directly
  - Example of a stack
    - Implementation #1: array
    - Implementation #2: linked list
  - Backup slides on void pointers follow…

# Backup Slides on Void Opaque Pointers

# stack.h (with void*)

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

typedef struct Item *Item_T;
typedef struct Stack *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, void *item);
extern void *Stack_pop(Stack_T stk);

/* It's a checked runtime error to pass a NULL Stack_T to any
   routine, or call Stack_pop with an empty stack
*/

#endif
```

# Stack Implementation (with void*)

## stack.c

```
#include <assert.h>
#include <stdlib.h>
#include “stack.h”

struct Stack {struct List *head;};
struct List {void *val; struct List *next;};

Stack_T Stack_new(void) {
  Stack_T stk = malloc(sizeof(*stk));
  assert(stk);
  stk->head = NULL;
  return stk;
}
```

# `stack.c` (with `void*`) continued

```
int Stack_empty(Stack_T stk) {
  assert(stk != NULL);
  return stk->head == NULL;
}

void Stack_push(Stack_T stk, void *item) {
  Stack_T t = malloc(sizeof(*t));
  assert(t != NULL);
  assert(stk != NULL);
  t->val = item;
  t->next = stk->head;
  stk->head = t;
}
```

# stack.c (with void*) continued

```
void *Stack_pop(Stack_T stk) {
  void *x;
  struct List *p;
  assert(stk != NULL);
  assert(stk->head != NULL);
  x = stk->head->val;
  p = stk->head;
  stk->head = stk->head->next;
  free(p);
  return x;
}
```

# Client Program (With Void)

## `client.c` (with void*)

```
#include <stdio.h>
#include <stdlib.h>
#include “item.h”
#include “stack.h”

int main(int argc, char *argv[]) {
   int i;
   Stack_T s = Stack_new();
   for (i = 1; i < argc; i++)
      Stack_push(s, Item_new(argv[i]));
   while (!Stack_empty(s))
      printf(“%s\n”,Stack_pop(s));
   return 0;
}
```

# Structural Equality Testing

Suppose we want to test two stacks for equality:

```
int Stack_equal(Stack_T s1, Stack_T s2);
```

How can this be implemented?

```
int Stack_equal(Stack_T s1, Stack_T s2) {
  return (s1 == s2);
}
```

We want to test whether two stacks are equivalent stacks, not whether they are the <u>*same*</u> stack.

# Almost, But Not Quite...

How about this:

```
int Stack_equal(Stack_T s1, Stack_T s2) {
  struct List *p, *q;
  for (p=s1->head, q=s2->head;  p && q;
              p=p->next, q=q->next)
      if (p->val != q->val)
         return 0;
  return p==NULL && q==NULL;
}
```

This is better, but what we want to test whether **`s1->val`** is *equivalent* to **`s2->val`**, not whether it is the *same*.

# Item ADT Provides Equal Test

How about this:

```
int Stack_equal(Stack_T s1, Stack_T s2) {
  struct List *p, *q;
  for (p=s1->head, q=s2->head;  p && q;
            p=p->next, q=q->next)
      if ( ! Item_equal(p->val, q->val))
         return 0;
  return p==NULL && q==NULL;
}
```

This is good for the “Item_T” version of stacks (provided the Item interface has an Item_equal function), but what about the void* version of stacks?

# Function Pointers

How about this:

```
int Stack_equal(Stack_T s1, Stack_T s2,
                int (*equal)(void *, void *)) {
  struct List *p, *q;
  for (p=s1->head, q=s2->head;  p && q;
             p=p->next, q=q->next)
      if ( ! equal((void*)p->val, (void*) q->val))
         return 0;
  return p==NULL && q==NULL;
}
```

The client must pass an equality-tester function to Stack_equal.

# Passing a Function Pointer

```
int Stack_equal(Stack_T s1, Stack_T s2,
                int (*equal)(void *, void *)) {
  struct List *p, *q;
  for (p=s1->head, q=s2->head;  p && q;
              p=p->next, q=q->next)
      if ( ! equal((void*)p->val, (void*) q->val))
         return 0;
  return p==NULL && q==NULL;
}
```

Client:

```
int char_equal (char *a, char *b) {
   return (!strcmp(a,b));
}
int string_stacks_equal(Stack_T st1, Stack_T st2) {
   return Stack_equal(st1, st2,
                (int (*)(void*, void*)) char_equal);
}
```

cast