

Week 03: Data Abstraction

Data Abstraction

Abstract Data Types

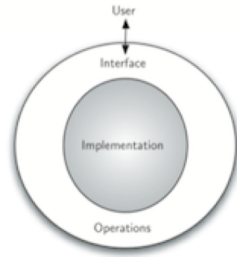
2/73

A *data type* is ...

- a set of *values* (atomic or structured values) e.g. *integer stacks*
- a collection of *operations* on those values e.g. *push, pop, isEmpty?*

An *abstract data type* ...

- is a logical description of how we view the data and operations
- without regard to how they will be implemented
- creates an *encapsulation* around the data
- is a form of *information hiding*



... Abstract Data Types

3/73

Users of the ADT see only the *interface*

Builders of the ADT provide an *implementation*

ADT *interface* provides

- a user-view of the data structure
- function signatures (prototypes) for all operations
- semantics of operations (via documentation)
- \Rightarrow a "contract" between ADT and its clients

ADT *implementation* gives

- concrete definition of the data structures
- function implementations for all operations

... Abstract Data Types

4/73

ADT interfaces are *opaque*

- clients *cannot* see the implementation via the interface

ADTs are important because ...

- facilitate decomposition of complex programs
- make implementation changes invisible to clients
- improve readability and structuring of software

... Abstract Data Types

5/73

Typical operations with ADTs

- *create* a value of the type
- *modify* one variable of the type
- *combine* two values of the type

Collections

6/73

Common ADTs ...

- consist of a *collection* of *items*
- where each item may be a simple type or an ADT
- and items often have a *key* (to identify them)

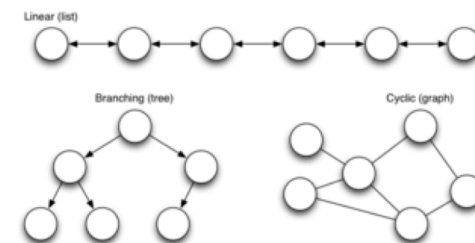
Collections may be categorised by ...

- *structure*:
 - linear (array, linked list), branching (tree), cyclic (graph)
- *usage*:
 - matrix, stack, queue, set, search-tree, dictionary, map, ...

... Collections

7/73

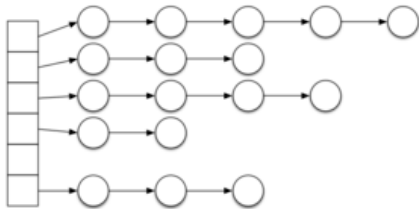
Collection structures:



... Collections

8/73

Or even a hybrid structure like:



... Collections

9/73

For a given collection type

- many different data representations are possible

For a given operation and data representation

- several different algorithms are possible
- efficiency of algorithms may vary widely

Generally,

- there is no overall "best" representation/implementation
- cost depends on the mix of operations
(e.g. proportion of inserts, searches, deletions, ...)

ADOs and ADTs

10/73

We want to distinguish ...

- ADO = *abstract data object*
- ADT = *abstract data type*

Warning: Sedgewick's first few examples are ADOs, not ADTs.

Example: Abstract Stack Data Object

11/73

Stack, aka *pushdown stack* or *LIFO data structure*

Assume (for the time being) stacks of `char` values

Operations:

- *create* an empty stack
- insert (*push*) an item onto stack
- remove (*pop*) most recently pushed item
- check whether stack *is empty*

... Example: Abstract Stack Data Object

12/73

Example of use:

Stack	Operation	Return value
?	create	-
-	isempty	true
-	push a	-
a	push b	-
a b	push c	-
a b c	pop	c
a b	isempty	false

Exercise #1: Stack vs Queue

13/73

Consider the previous example but with a queue instead of a stack.

Which element would have been taken out ("dequeued") first?

a

Stack as ADO

15/73

Interface (a file named `Stack.h`)

```
// Stack ADO header file

#define MAXITEMS 10

void StackInit();           // set up empty stack
int  StackIsEmpty();        // check whether stack is empty
void StackPush(char);       // insert char on top of stack
char StackPop();            // remove char from top of stack
```

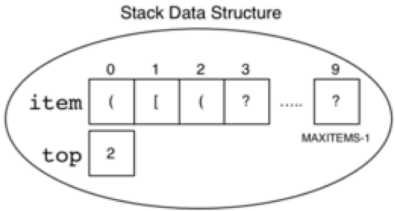
Note:

- no explicit reference to Stack object
- this makes it an *Abstract Data Object (ADO)*

... Stack as ADO

16/73

Implementation may use the following data structure:



... Stack as ADO

17/73

Implementation (in a file named Stack.c):

```
#include "Stack.h"
#include <assert.h>

// define the Data Structure
typedef struct {
    char item[MAXITEMS];
    int top;
} stackRep;

// define the Data Object
static stackRep stackObject;

// set up empty stack
void StackInit() {
    stackObject.top = -1;
}

// check whether stack is empty
int StackIsEmpty() {
    return (stackObject.top < 0);
}

// insert char on top of stack
void StackPush(char ch) {
    assert(stackObject.top < MAXITEMS-1);
    stackObject.top++;
    int i = stackObject.top;
    stackObject.item[i] = ch;
}

// remove char from top of stack
char StackPop() {
    assert(stackObject.top > -1);
    int i = stackObject.top;
    char ch = stackObject.item[i];
    stackObject.top--;
    return ch;
}
```

- `assert(test)` terminates program with error message if `test` fails
- `static Type Var` declares `Var` as *local* to `Stack.c`

Exercise #2: Bracket Matching

18/73

Bracket matching ... check whether all opening brackets such as '(', '[', '{' have matching closing brackets ')', ']', '}'

Which of the following expressions are balanced?

1. `(a+b) * c`
2. `a[i]+b[j]*c[k]`
3. `(a[i]+b[j])*c[k]`
4. `a(a+b)*c`
5. `void f(char a[], int n) {int i; for(i=0;i<n;i++) { a[i] = (a[i]*a[i])*(i+1); }}`

6. `a(a+b * c`

1. balanced
2. not balanced (case 1: an opening bracket is missing)
3. balanced
4. not balanced (case 2: closing bracket doesn't match opening bracket)
5. balanced
6. not balanced (case 3: missing closing bracket)

... Stack as ADO

20/73

Bracket matching algorithm, to be implemented as a *client* for Stack ADO:

```
bracketMatching(s):
    Input  stream s of characters
    Output true if parentheses in s balanced, false otherwise

    for each ch in s do
        if ch = open bracket then
            push ch onto stack
        else if ch = closing bracket then
            if stack is empty then
                return false // opening bracket missing (case 1)
            else
                pop top of stack
                if brackets do not match then
                    return false // wrong closing bracket (case 2)
                end if
            end if
        end if
    end for
    if stack is not empty then return false // some brackets unmatched (case 3)
    else return true
```

... Stack as ADO

21/73

Execution trace of client on sample input:

([{ }])

Next char	Stack	Check
-	empty	-
((-
[([-
{	([{	-
}	([{ vs } ✓
]	([vs] ✓

)	empty	(vs) ✓
eof	empty	-

Exercise #3: Bracket Matching Algorithm

22/73

Trace the algorithm on the input

```
void f(char a[], int n) {
    int i;
    for(i=0;i<n;i++) { a[i] = a[i]*a[i]}*(i+1); }
}
```

Next bracket	Stack	Check
start	empty	-
((-
[([-
]	(✓
)	empty	✓
{	{	-
({ (-
)	{	✓
{	{ {	-
[{ { [-
]	{ {	✓
[{ { [-
]	{ {	✓
[{ { [-
]	{ {	✓
)	{	false

Exercise #4: Implement Bracket Matching Algorithm in C

24/73

- Use Stack ADT

```
#include "Stack.h"
```

- Sidetrack: Character I/O Functions in C* (requires <stdio.h>)

```
int getchar(void);
```

- returns character read from standard input as an `int`, or returns **EOF** on end of file (keyboard: CTRL-D on Unix, CTRL-Z on Windows)

```
int putchar(int ch);
```

- writes the character `ch` to standard output
- returns the character written, or EOF on error

Managing Abstract Data Structures in C

Compilers

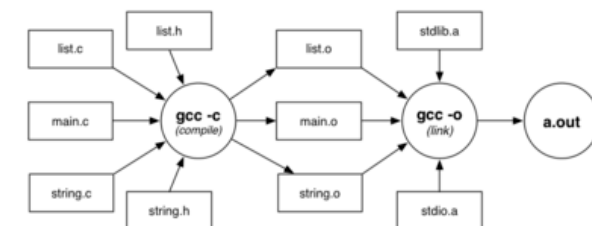
26/73

Compilers are programs that

- convert program source code to executable form
- "executable" might be machine code or bytecode

The Gnu C compiler (**gcc**)

- applies source-to-source transformation (pre-processor)
- compiles *source code* to produce *object files*
- links object files and *libraries* to produce *executables*



... Compilers

27/73

Compilation/linking with gcc

```
gcc -c Stack.c
produces Stack.o, from Stack.c and Stack.h
```

```
gcc -c brackets.c
produces brackets.o, from brackets.c and Stack.h
```

```
gcc -o rbt brackets.o Stack.o
links brackets.o, Stack.o and libraries
producing executable program called rbt
```

Note that `stdio`, `assert` included implicitly.

gcc is a multi-purpose tool

- compiles (`-c`), links, makes executables (`-o`)

Make/Makefiles

28/73

Compilation process is complex for large systems.

How much to compile?

- ideally, what's changed since last compile
- practically, recompile everything, to be sure

The **make** command assists by allowing

- programmers to document *dependencies* in code
- minimal re-compilation, based on dependencies

... Make/Makefiles

29/73

Example multi-module program ...



... Make/Makefiles

30/73

make is driven by dependencies given in a **Makefile**

A *dependency* specifies

target : *source*₁ *source*₂ ...
 commands to build target from sources

e.g.

```
game : main.o graphics.o world.o
      gcc -o game main.o graphics.o world.o
```

Rule: *target* is rebuilt if older than any *source*_{*i*}

... Make/Makefiles

31/73

A **Makefile** for the example program:

```
game : main.o graphics.o world.o
      gcc -o game main.o graphics.o world.o

main.o : main.c graphics.h world.h
      gcc -Wall -Werror -std=c11 -c main.c

graphics.o : graphics.c world.h
      gcc -Wall -Werror -std=c11 -c graphics.c

world.o : world.c
      gcc -Wall -Werror -std=c11 -c world.c
```

Things to note:

- A *target* (`game`, `main.o`, ...) is on a newline
 - followed by a **:**
 - then followed by the files that the target is dependent on
- The *action* (`gcc ...`) is always on a newline
 - and must be indented with a *TAB*

... Make/Makefiles

32/73

If `make` arguments are targets, build just those targets:

```
prompt$ make world.o
gcc -Wall -Werror -std=c11 -c world.c
```

If no args, build first target in the Makefile.

```
prompt$ make
gcc -Wall -Werror -std=c11 -c main.c
gcc -Wall -Werror -std=c11 -c graphics.c
gcc -Wall -Werror -std=c11 -c world.c
gcc -o game main.o graphics.o world.o
```

Exercise #5: Makefile

33/73

Write a Makefile for the bracket matching program.

From ADOs to ADTs

34/73

- `Stack.c` provides a single abstract object **stackObject**

Abstract Data *Types*

- allow clients to create and manipulate arbitrarily many data objects of an abstract type
- ... without revealing the implementation to a client

In C, ADTs are implemented using *pointers* and *dynamic memory allocation*

Pointers

Sidetrack: Numeral Systems

36/73

Numeral system ... system for representing numbers using digits or other symbols.

- Most cultures have developed a *decimal* system (based on 10)
- For computers it is convenient to use a *binary* (base 2) or a *hexadecimal* (base 16) system

... Sidetrack: Numeral Systems

37/73

Decimal representation

- The **base** is 10; digits 0 - 9
- Example: decimal number 4705 can be interpreted as
 $4 \cdot 10^3 + 7 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0$
- Place values:

...	1000	100	10	1
...	10^3	10^2	10^1	10^0

... Sidetrack: Numeral Systems

38/73

Binary representation

- The **base** is 2; digits 0 and 1
- Example: binary number 1101 can be interpreted as
 $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- Place values:

...	8	4	2	1
...	2^3	2^2	2^1	2^0

- Write number as **0b1101** (= 13)

... Sidetrack: Numeral Systems

39/73

Hexadecimal representation

- The **base** is 16; digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Example: hexadecimal number 3AF1 can be interpreted as
 $3 \cdot 16^3 + 10 \cdot 16^2 + 15 \cdot 16^1 + 1 \cdot 16^0$
- Place values:

...	4096	256	16	1
...	16^3	16^2	16^1	16^0

- Write number as **0x3AF1** (= 15089)

Exercise #6: Conversion Between Different Numeral Systems

40/73

1. Convert 74 to base 2
2. Convert 0x2D to base 10
3. Convert 0b1011111000101001 to base 16
 - Hint: **1011111000101001**
4. Convert 0x12D to base 2

1. **0b1001010**
2. **45**
3. **0xBE29**
4. **0b100101101**

Memory

42/73

Computer memory ... large array of consecutive data cells or bytes

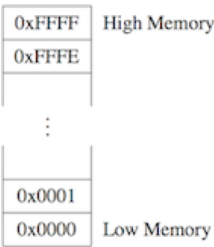
- `char` ... 1 byte `int, float` ... 4 bytes `double` ... 8 bytes

When a variable is declared, the operating system finds a place in memory to store the appropriate number of bytes.

If we declare a variable called `k` ...

- the place where `k` is stored is denoted by **&k**
- also called the **address** of `k`

It is convenient to print memory addresses in Hexadecimal notation



43/73

... Memory

Example:

```
int k;
int m;

printf("address of k is %p\n", &k);
printf("address of m is %p\n", &m);
```

```
address of k is BFFFFB80
address of m is BFFFFB84
```

This means that

- k occupies the four bytes from BFFFFB80 to BFFFFB83
- m occupies the four bytes from BFFFFB84 to BFFFFB87

Note the use of `%p` as placeholder for an address ("pointer" value)

... Memory

44/73

When an array is declared, the elements of the array are guaranteed to be stored in consecutive memory locations:

```
int array[5];

for (i = 0; i < 5; i++) {
    printf("address of array[%d] is %p\n", i, &array[i]);
}
```

```
address of array[0] is BFFFFB60
address of array[1] is BFFFFB64
address of array[2] is BFFFFB68
address of array[3] is BFFFFB6C
address of array[4] is BFFFFB70
```

Application: Input Using scanf()

45/73

Standard I/O function `scanf()` requires the *address* of a variable as argument

- `scanf()` uses a format string like `printf()`
- use `%d` to read an integer value

```
#include <stdio.h>
...
int answer;
printf("Enter your answer: ");
scanf("%d", &answer);
```

- use `%f` to read a floating point value (`%lf` for double)

```
float e;
printf("Enter e: ");
scanf("%f", &e);
```

- `scanf()` returns a value — the number of items read
 - use this value to determine if `scanf()` successfully read a number
 - `scanf()` could fail e.g. if the user enters letters

Exercise #7: Using scanf

46/73

Write a program that

- asks the user for a number
- checks that it is positive
- applies Collatz's process (Exercise 4, Problem Set Week 2) to the number

```
#include <stdio.h>
```

```
void collatz(int n) {
    printf("%d\n", n);
    while (n != 1) {
        if (n % 2 == 0)
            n = n / 2;
        else
            n = 3*n + 1;
        printf("%d\n", n);
    }
}
```

```
int main(void) {
    int n;
    printf("Enter a positive number: ");
    if (scanf("%d", &n) == 1 && (n > 0)) /* test if scanf successful
                                         and returns positive number */
        collatz(n);
    return 0;
}
```

Pointers

48/73

A *pointer* ...

- is a special type of variable
- storing the **address** (memory location) of another variable

A pointer occupies space in memory, just like any other variable of a certain type

The number of memory cells needed for a pointer depends on the computer's architecture:

- Old computer, or hand-held device with only 64KB of addressable memory:
 - 2 memory cells (i.e. 16 bits) to hold any address from 0x0000 to 0xFFFF (= 65535)
- Desktop machine with 4GB of addressable memory

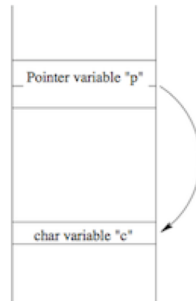
- 4 memory cells (i.e. 32 bits) to hold any address from 0x00000000 to 0xFFFFFFFF (= 4294967295)
- Modern 64-bit computer
 - 8 memory cells (can address 2^{64} bytes, but in practice the amount of memory is limited by the CPU)

... Pointers

49/73

Suppose we have a pointer **p** that "points to" a `char` variable `c`.

Assuming that the pointer **p** requires 2 bytes to store the address of `c`, here is what the memory map might look like:



... Pointers

50/73

Now that we have assigned to `p` the address of variable `c` ...

- need to be able to reference the data in that memory location

Operator `*` is used to access the object the pointer points to

- e.g. to change the value of `c` using the pointer `p`:

```
*p = 'T'; // sets the value of c to 'T'
```

The `*` operator is sometimes described as "*dereferencing*" the pointer, to access the underlying variable

... Pointers

51/73

Things to note:

- all pointers constrained to point to a particular type of object

```
// a potential pointer to any object of type char
char *s;
```

```
// a potential pointer to any object of type int
int *p;
```

- if pointer `p` is pointing to an integer variable `x`
⇒ `*p` can occur in any context that `x` could

Examples of Pointers

52/73

```
int *p; int *q; // this is how pointers are declared
int a[5];
int x = 10, y;
```

```
p = &x;          // p now points to x
*p = 20;         // whatever p points to is now equal to 20
y = *p;          // y is now equal to whatever p points to
p = &a[2];        // p points to an element of array a[]
q = p;           // q and p now point to the same thing
```

Exercise #8: Pointers

53/73

What is the output of the following program?

```
1  #include <stdio.h>
2
3  int main(void) {
4      int *ptr1, *ptr2;
5      int i = 10, j = 20;
6
7      ptr1 = &i;
8      ptr2 = &j;
9
10     *ptr1 = *ptr1 + *ptr2;
11     ptr2 = ptr1;
12     *ptr2 = 2 * (*ptr2);
13     printf("Val = %d\n", *ptr1 + *ptr2);
14     return 0;
15 }
```

Val = 120

... Examples of Pointers

55/73

Can we write a function to "swap" two variables?

The *wrong* way:

```
void swap(int a, int b) {
    int temp = a;           // only local "copies" of a and b will swap
    a = b;
```



```

    b = temp;
}

int main(void) {
    int a = 5, b = 7;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b); // a and b still have their original values
    return 0;
}

```

... Examples of Pointers

56/73

In C, parameters are "call-by-value"

- changes made to the value of a parameter do not affect the original
- function `swap()` tries to swap the values of `a` and `b`, but fails because it only swaps the copies, not the "real" variables in `main()`

We can achieve "simulated call-by-reference" by passing pointers as parameters

- this allows the function to change the "actual" value of the variables

... Examples of Pointers

57/73

Can we write a function to "swap" two variables?

The *right* way:

```

void swap(int *p, int *q) {
    int temp = *p;           // change the actual values of a and b
    *p = *q;
    *q = temp;
}

int main(void) {
    int a = 5, b = 7;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b); // a and b now successfully swapped
    return 0;
}

```

Pointers and Arrays

58/73

An alternative approach to iteration through an array:

- determine the **address of the first element** in the array
- determine the **address of the last element** in the array
- set a pointer variable to refer to the first element
- use **pointer arithmetic** to move from element to element
- terminate loop when address exceeds that of last element

Example:

```

int a[6];
int *p = &a[0];
while (p <= &a[5]) {
    printf("%2d ", *p);
    p++;
}

```

... Pointers and Arrays

59/73

Pointer-based scan written in more typical style

```

int *p;
int a[6];
for (p = &a[0]; p < &a[6]; p++)
    printf("%2d ", *p);

```

Annotations:

- address of first element**: points to `&a[0]`
- address of last element + 1**: points to `&a[6]`
- access current element**: points to `*p`
- pointer arithmetic (move to next element)**: points to `p++`

Note: because of pointer/array connection `a[i] == *(a+i)`

Pointer Arithmetic

60/73

A *pointer* variable holds a value which is an *address*.

C knows what type of object is being pointed to

- it knows the `sizeof` that object
- it can compute where the next/previous object is located

Example:

```

int a[6]; // assume array starts at address 0x1000
int *p;
p = &a[0]; // p contains 0x1000
p = p + 1; // p now contains 0x1004

```

... Pointer Arithmetic

61/73

For a pointer declared as `T *p;` (where `T` is a type)

- if the pointer initially contains address `A`
 - executing `p = p + k;` (where `k` is a constant)
 - changes the value in `p` to `A + k*sizeof(T)`

The value of `k` can be positive or negative.

Example:

```
int a[6];      (addr 0x1000)      char s[10];    (addr 0x2000)
int *p;        (p == ?)          char *q;        (q == ?)
p = &a[0];      (p == 0x1000)      q = &s[0];      (q == 0x2000)
p = p + 2;      (p == 0x1008)      q++;              (q == 0x2001)
```

Arrays of Strings

62/73

One common type of pointer/array combination are the *command line arguments*

- These are 0 or more strings specified when program is run
- Suppose you have an executable program named `seqq`. If you run this command in a terminal:

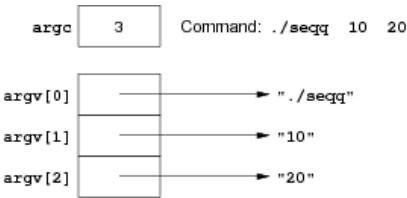
```
prompt$ ./seqq 10 20

then seqq will be given 2 command-line arguments: "10", "20"
```

... Arrays of Strings

63/73

```
prompt$ ./seqq 10 20
```



Each element of `argv[]` is

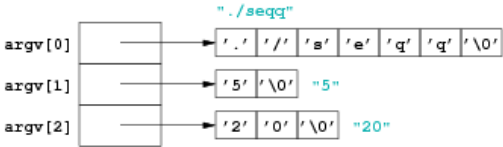
- a pointer to the start of a character array (`char *`)
 - containing a `\0`-terminated string

... Arrays of Strings

64/73

More detail on how `argv` is represented:

```
prompt$ ./seqq 5 20
```



... Arrays of Strings

65/73

`main()` needs different prototype if you want to access command-line arguments:

```
int main(int argc, char *argv[]) { ...
```

- `argc` ... stores the number of command-line arguments + 1
 - `argc == 1` if no command-line arguments
- `argv[]` ... stores program name + command-line arguments
 - `argv[0]` always contains the program name
 - `argv[1], argv[2], ...` are the command-line arguments if supplied

`<stdlib.h>` defines useful functions to convert strings:

- `atoi(char *s)` converts string to int
- `atof(char *s)` converts string to double (can also be assigned to `float` variable)

Exercise #9: Command Line Arguments

66/73

Write a program that

- checks for a single command line argument
 - if not, outputs a usage message and exits with failure
- converts this argument to a number and checks that it is positive
- applies Collatz's process (Exercise 4, Problem Set Week 2) to the number

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

void collatz(int n) {
    ...
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s number\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (n > 0)
        collatz(n);
    return 0;
}
```

... Arrays of Strings

68/73

`argv` can also be viewed as *double pointer* (a pointer to a pointer)

⇒ Alternative prototype for `main()`:

```
int main(int argc, char **argv) { ...
```

Can still use argv[0], argv[1],...

Pointers and Structures

Like any object, we can get the address of a struct via &.

```
typedef char Date[11]; // e.g. "03-08-2017"
typedef struct {
    char name[60];
    Date birthday;
    int status; // e.g. 1 (= full time)
    float salary;
} WorkerT;

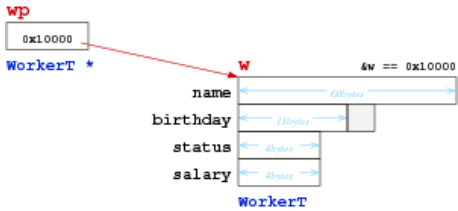
WorkerT w; WorkerT *wp;
wp = &w;
// a problem ...
*wp.salary = 125000.00;
// does not have the same effect as
w.salary = 125000.00;
// because it is interpreted as
*(wp.salary) = 125000.00;

// to achieve the correct effect, we need
(*wp).salary = 125000.00;
// a simpler alternative is normally used in C
wp->salary = 125000.00;
```

Learn this well; we will frequently use it in this course.

... Pointers and Structures

Diagram of scenario from program above:



... Pointers and Structures

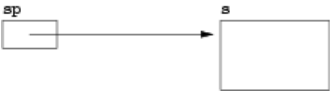
General principle ...

If we have:

```
SomeStructType s, *sp = &s;
```

then the following are all equivalent:

```
s.SomeElem    sp->SomeElem    (*sp).SomeElem
```



Tips for Week 3 Problem Set

Main themes: *Abstract data objects; pointers*

- Redefine char stack ADO to integer stack ADO, integer queue ADO
 - Develop clients for integer stack ADO
 - read numbers from stdin
 - read command line argument(s) and convert to integer
 - use stack to convert decimal number to binary:
- ```
prompt$./binary 13
1101
```
- write Makefile to build executable from IntStack.h, IntStack.c, binary.c
  - Exercise 5: check your understanding of pointers for arrays and structs; pointer arithmetic
  - Challenge Exercise: wrack your brain — do not use any string functions

## Summary

- Introduction to ADOs and ADTs
  - Compilation and **Makefiles**
  - Pointers
- Suggested reading:
  - introduction to ADTs ... Sedgewick, Ch.4.1-4.3
  - pointers ... Moffat, Ch.6.6-6.7