COMP9024: Data Structures and Algorithms

Week 1: Abstract Data Types and Pointers

Contents

Abstract Data Types (1/4)

- A data type is a set of values, and a set of operations on those values
- An ADT (Abstract Data Type) is a mathematical model for data types
    - An approach to implementing data types
    - Separates interface from implementation
- Users of an ADT see only the interface
- Builders of the ADT provide an implementation

Abstract Data Types (2/4)

An 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

An ADT implementation gives

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

Abstract Data Types (3/4)

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

Abstract Data Types (4/4)

Typical operations with ADTs

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

Collection ADTs (1/4)

A collection consist of a group of items where each item may be a simple type or an ADT.

Items are typically of the same type and often have a key (to identify them)

Collections may be categorised by …

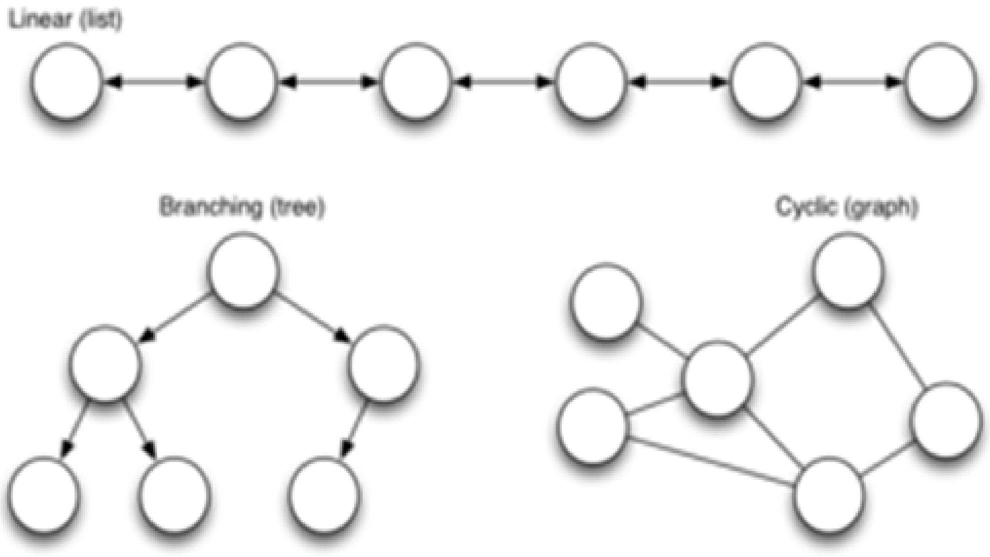
- structure:

  linear (array, linked list), branching (tree), cyclic (graph)
- usage:

  matrix, stack, queue, set, search-tree, dictionary, map, …

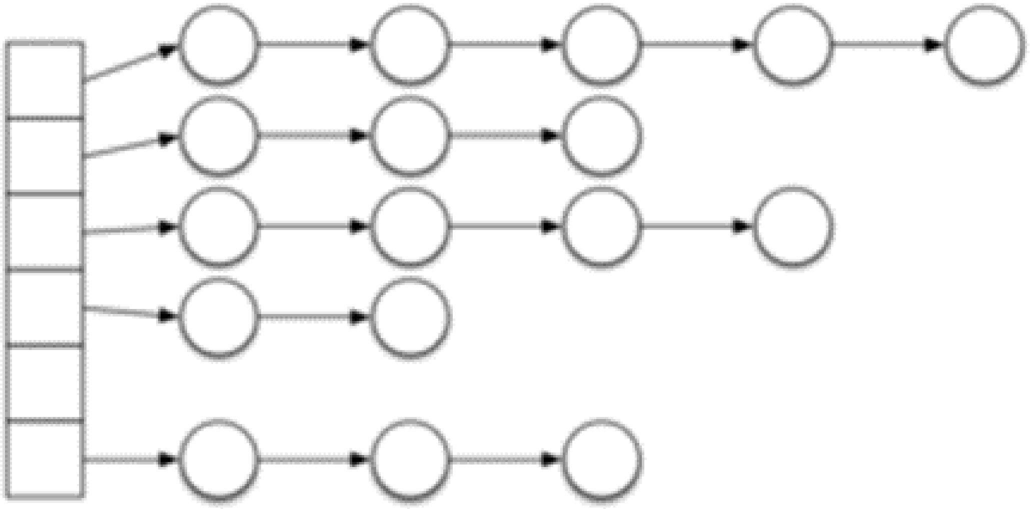
Collections (2/4)

Collection structures:



Collections (3/4)

- Or even a hybrid structure like:



Collection (4/4)

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, …)

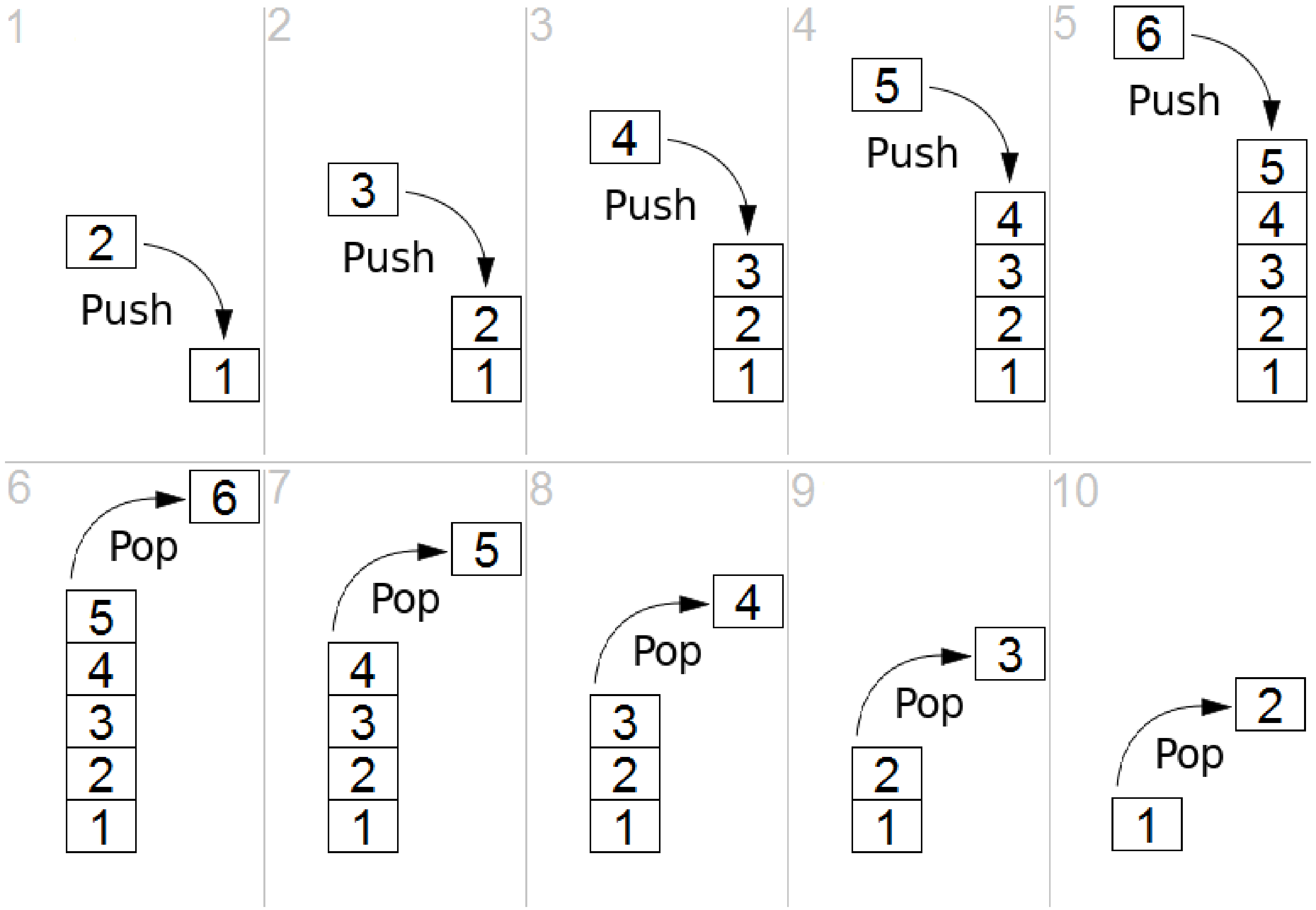
Stack ADT (1/2)

A stack is an abstract data type that serves as a collection of elements, with the following operations:

- createStack(), which creates an empty stack.
- push(element), which adds an element to the collection, and
- pop(), which removes the top element from the stack.
- peek(), which returns the top element without modifying the stack.
- isEmpty(), which checks if the stack is empty.

Elements come off a stack following LIFO (Last In First Out) order.

Stack ADT (2/2)



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

An Implementation of Stack (1/3)

Implementation issues:

- A data structure to store all the elements
    - Different data structures (array, linked lists, …) can be used
- A stack pointer to point to the stack top

Note that there is a hardware stack in each processor, and the processor provides

- push and pop instructions, and
- a register serving as a stack pointer

An Implementation of Stack (2/3)

Interface (a file named Stack.h)

```
// Stack header file

void stackInit();
int  isEmpty();
void push(char);
char pop();
```

An Implementation of Stack (3/3)

```c
#include "Stack.h"

#define MAXITEMS 10
static struct {
  char item[MAXITEMS];
  int  top;
} stackObject;

 void stackInit() {
   stackObject.top = -1;
 }

 int isEmpty() {
   return (stackObject.top < 0);
 }
```

```c
void push(char ch) {
  assert(stackObject.top < MAXITEMS-1);
  stackObject.top++;
  int i = stackObject.top;
  stackObject.item[i] = ch;
}

char pop() {
  assert(stackObject.top > -1);
  int i = stackObject.top;
  char ch = stackObject.item[i];
  stackObject.top--;
  return ch;
}
```

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Bracket Matching (1/5)

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

Which of the following expressions are correct?

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

Bracket Matching (2/5)

Algorithm

```
#include "Stack.h"

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
|   |   |   else
|   |   |      pop top of stack
|   |   |      if brackets do not match then
|   |   |         return FALSE
|   |   |      end if
|   |   |   end if
|   |   end if
|   end for
|   if stack is not empty then return FALSE
|   return TRUE
```

Bracket Matching (3/5)

Execution trace of client on sample input:

([{ }])

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

Bracket Matching (4/5)

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); }
}
```

Bracket Matching (5/5)

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

Queue ADT (1/4)

A queue consists of a linear sequence of an arbitrary number of items with the following major operations:

- enqueue(element): add a new element at the end of the queue
- dequeue(): remove the element at the front of the queue
- Other auxiliary operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): indicates whether no elements are stored

All the elements are removed from the queue following FIFO (First In First Out) order.

Queue ADT (2/4)

Operation		***Output***	*Q*
enqueue(5)		–	(5)
enqueue(3)		–	(5, 3)
dequeue()	*5*	(3)	
enqueue(7)		–	(3, 7)
dequeue()	*3*	(7)	
front()		*7*	(7)
dequeue()	*7*	()	
dequeue()	*"error"*	()	
isEmpty()		*true*	()
enqueue(9)		–	(9)
enqueue(7)		–	(9, 7)
size()		*2*	(9, 7)
enqueue(3)		–	(9, 7, 3)
enqueue(5)		–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)	

Queue ADT (3/4)

Applications of queues

- Direct applications
- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming
- Indirect applications
- Auxiliary data structure for algorithms
- Component of other data structures

Queue ADT (4/4)

- A queue can be implemented using an array or a linked list
- Two variables keep track of the front and rear

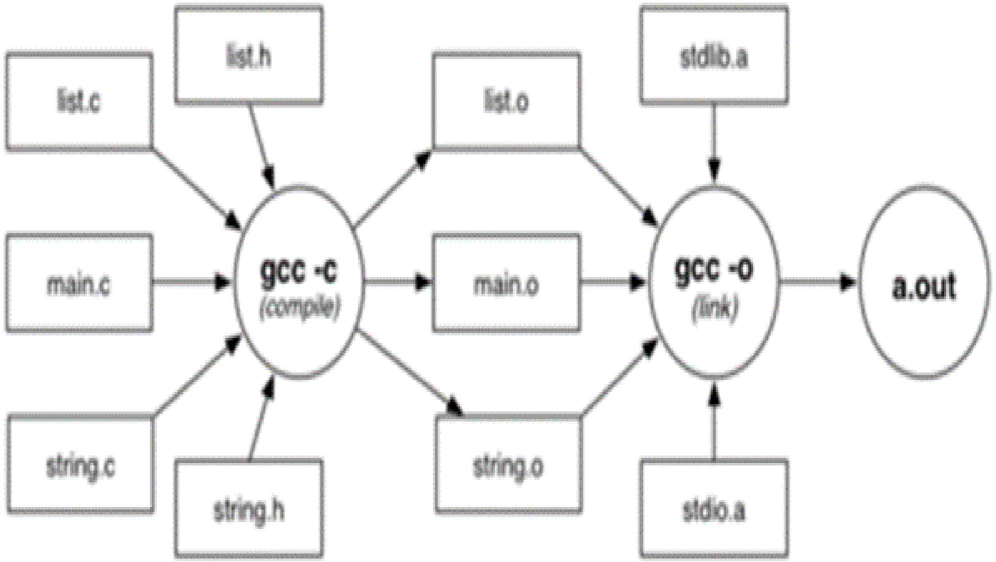
Compilation and Makefiles (1/7)

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*



Compilation and Makefiles (2/7)

Compilation/linking with gcc:

```
gcc -c Stack.c

gcc -c bracket.c

gcc -o rbt bracket.o Stack.o
```

gcc is a multi-purpose tool

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

Compilation and Makefiles (3/7)

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

Compilation and Makefiles (4/7)

Example: multi-module program

main.c

```
#include <stdio.h>
#include "world.h"
#include "graphics.h"

int main(void)
{
   ...
   drawPlayer(p);
   spin(...);
}
```

world.h

```
typedef ... Ob;
typedef ... Pl;

extern addObject(Ob);
extern remObject(Ob);
extern movePlayer(Pl);
```

world.c

```
#include <stdlib.h>

addObject(...)
{ ... }

remObject(...)
{ ... }

movePlayer(...)
{ ... }
```

graphics.h

```
extern drawObject(Ob);
extern drawPlayer(Pl);
extern spin(...);
```

graphics.c

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

drawObject(Ob o);
{ ... }

drawPlayer(Pl p)
{ ... }

spin(...)
{ ... }
```

Compilation and Makefiles (5/7)

make is driven by dependencies given in a Makefile

A dependency specifies

target : source1 source2 …

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*

Compilation and Makefiles (6/7)

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 -c main.c

graphics.o : graphics.c world.h
	gcc -Wall -Werror -c graphics.c

world.o : world.c
	gcc -Wall -Werror -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

Compilation and Makefiles (7/7)

If make arguments are targets, build just those targets:

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

If no args, build first target in the Makefile.

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

Memory (1/3)

Computer memory … large array of consecutive data cells or bytes

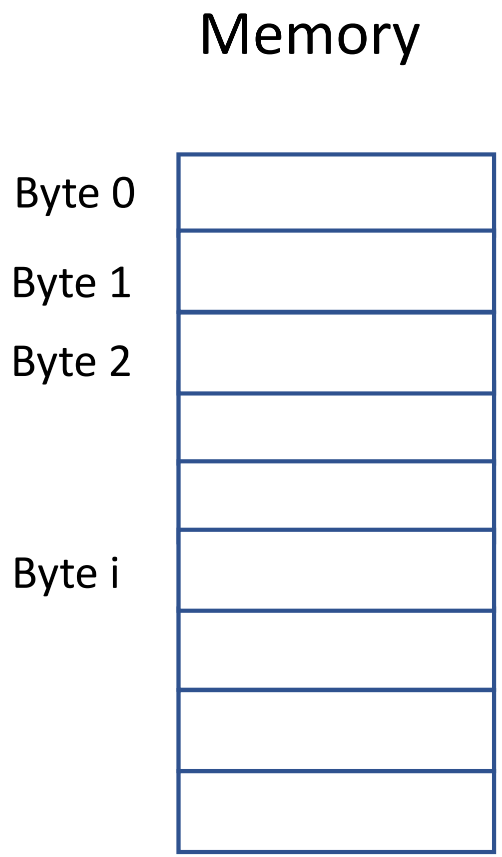
- char … 1 byte   int, float … 4 bytes   double … 8 bytes

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



Memory (2/3)

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 (3/3)

When an array is declared, the elements of the array are 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
```

Pointers (1/4)

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

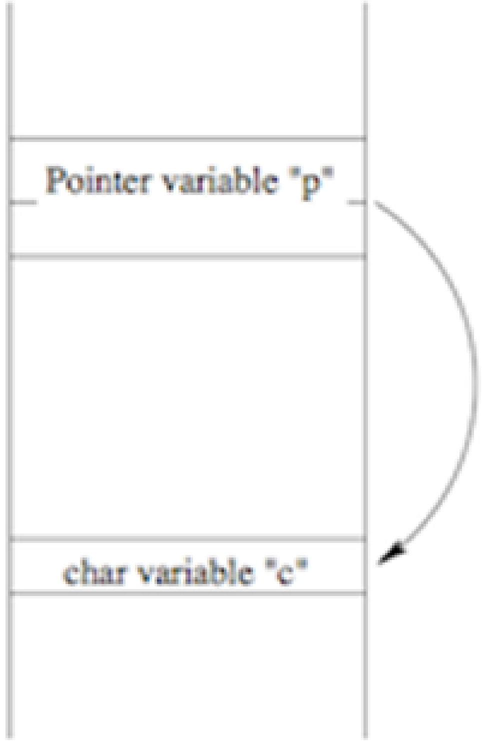
The number of memory cells needed for a pointer depends on the computer's architecture:

- 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 (2/4)

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 (3/4)

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';
```

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

Pointers (4/4)

Things to note:

- all pointers constrained to point to a particular type of object
-
- `char *s;`
-
-
- `int *p;`

if pointer `p` is pointing to an integer variable `x`

⇒ `*p` can occur in any context that `x` could

Examples of Pointers (1/5)

```
int *p; int *q;
int a[5];
int x = 10, y;

p = &x;
*p = 20;
y = *p;
p = &a[2];
q = p;
```

Examples of Pointers (2/5)

What is the output of the following program?

```c
#include <stdio.h>

int main(void) {
   int *ptr1, *ptr2;
   int i = 10, j = 20;

   ptr1 = &i;
   ptr2 = &j;

   *ptr1 = *ptr1 + *ptr2;
   ptr2 = ptr1;
   *ptr2 = 2 * (*ptr2);
   printf("Val = %d\n", *ptr1 + *ptr2);
   return 0;
}
```

```
Val = 120
```

Examples of Pointers (3/5)

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

The *wrong* way:

```
void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

int main(void) {
   int a = 5, b = 7;
   swap(a, b);
   printf("a = %d, b = %d\n", a, b);
   return 0;
}
```

Examples of Pointers (4/5)

Recall that in C, scalar parameters are passed "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

Examples of Pointers (5/5)

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

The *right* way:

```
void swap(int *p, int *q) {
   int temp = *p;
   *p = *q;
   *q = temp;
}

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

Pointers and Arrays (1/3)

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 (2/3)

Pointer-based scan written in more typical style

address of first element

address of last element + 1

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

pointer arithmetic
(move to next element)

access current element

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

Pointers and Arrays (3/3)

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], …

Pointer Arithmetic (1/7)

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];
int *p;
p = &a[0];
p = p + 1;
```

Pointer Arithmetic (2/7)

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];                char s[10];
int *p;                      char *q;
p = &a[0];               q = &s[0];
p = p + 2;               q++;
```

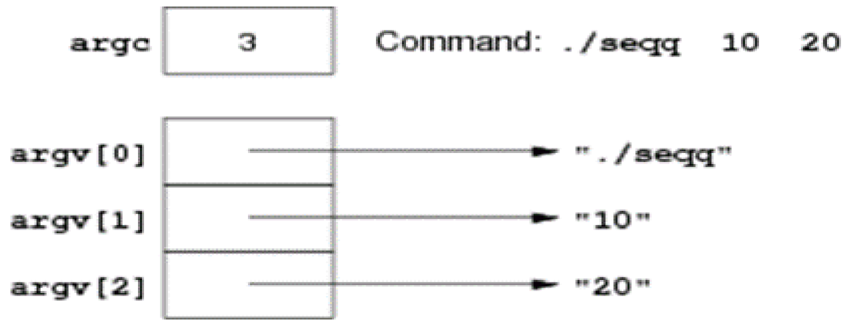
Pointer Arithmetic (3/7)

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

- These are 0 or more strings specified when a program is run
- If you run this command in a terminal:

  prompt$ ./seqq 10 20

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



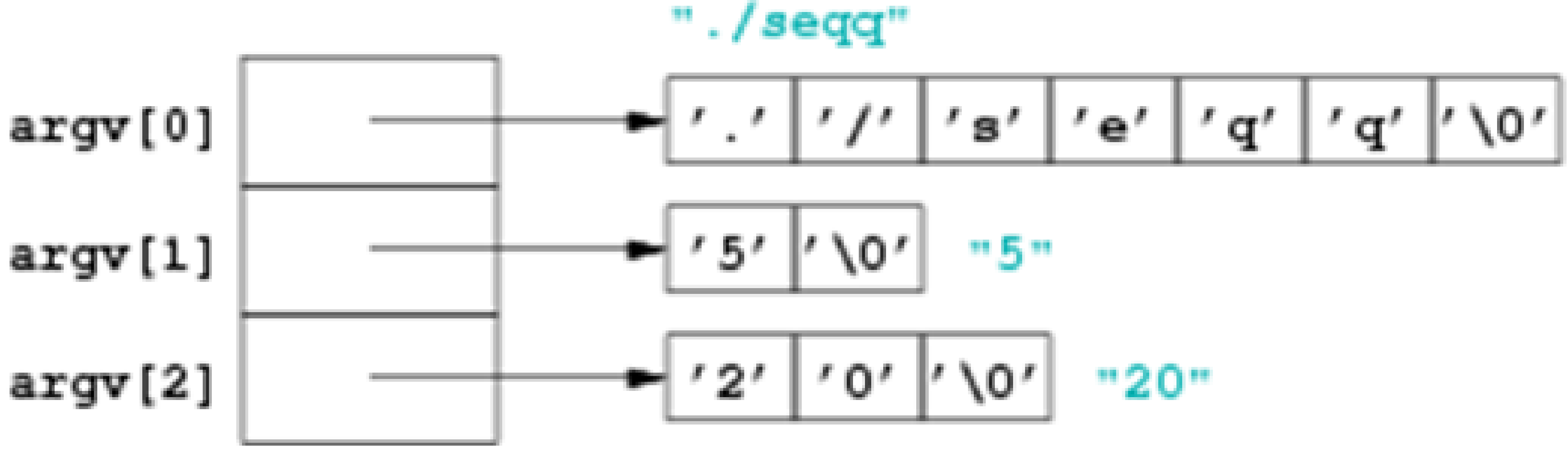
Each element of argv[] is

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

Pointer Arithmetic (4/7)

More detail on how argv is represented:

prompt$ ./seqq 5 20



Pointer Arithmetic (5/7)

`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)

Pointer Arithmetic (6/7)

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 the following Collatz's process, until 1 is reached:
  - If n is even, set n to n/2
  - If n is odd, set n to 3*n+1

Pointer Arithmetic (7/7)

```
#include <stdio.h>
#include <stdlib.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(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;
}
```

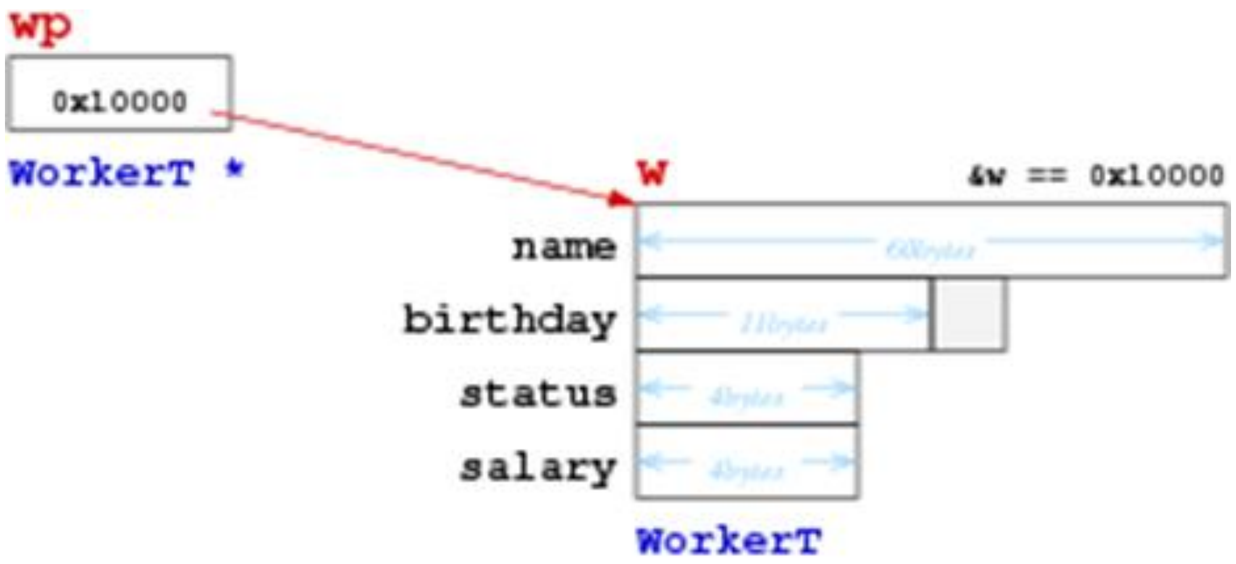
Pointers and Structures (1/3)

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

```
typedef char Date[11];
typedef struct {
    char  name[60];
    Date  birthday;
    int   status;
    float salary;
} WorkerT;
WorkerT w;  WorkerT *wp;
wp = &w;
*wp.salary = 125000.00;
w.salary = 125000.00;
*(wp.salary) = 125000.00;
(*wp).salary = 125000.00;
// wp->salary = 125000.00;
```

Pointers and Structures (2/3)

Diagram of scenario from program above:



Pointers and Structures (3/3)

General principle …

If we have:

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

then the following are all equivalent:

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

Summary

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