

(13-2) Dynamic Data Structures I

H&K Chapter 13

Instructor – Beiyu Lin

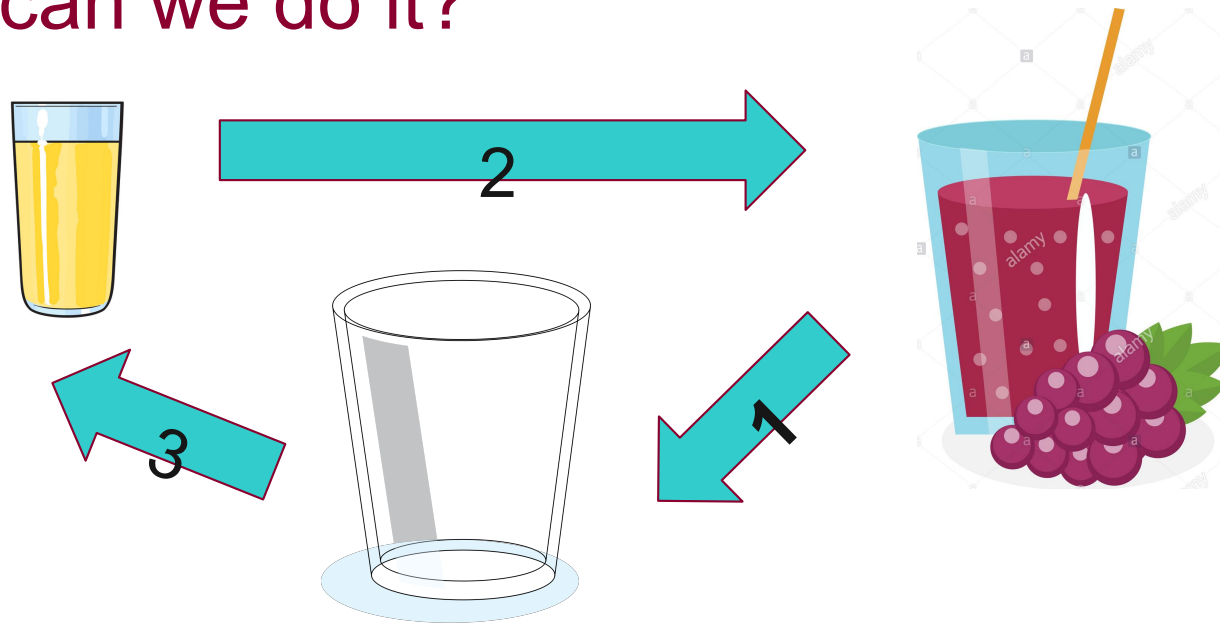
CptS 121 (Jan. 6th, 2019)

Washington State University



Review – Swap two items

- If we want to swap the juices in the cup, how can we do it?



Graphs are from:

<https://www.vectorstock.com/royalty-free-vector/cartoon-glass-cup-of-lemon-fruit-juice-vector-17979355>

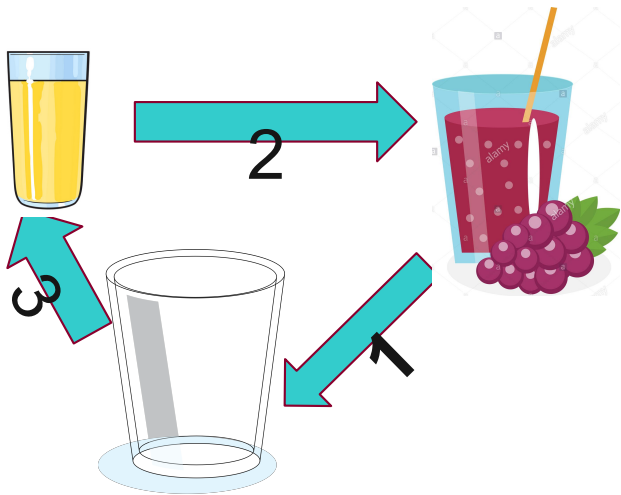
<https://www.alamy.com/stock-photo-grape-juice-in-a-glass-fresh-isolated-on-white-background-fruit-and-125939240.html>

<http://clipart-library.com/clipart/1971889.htm>



Review – Swap two items

- If we want to swap two integer in an array, how can we do it?



```
int temp;  
temp = int_arr[i];  
int_arr[i] = int_arr[j];  
int_arr[j] = temp;
```

Graphs are from:

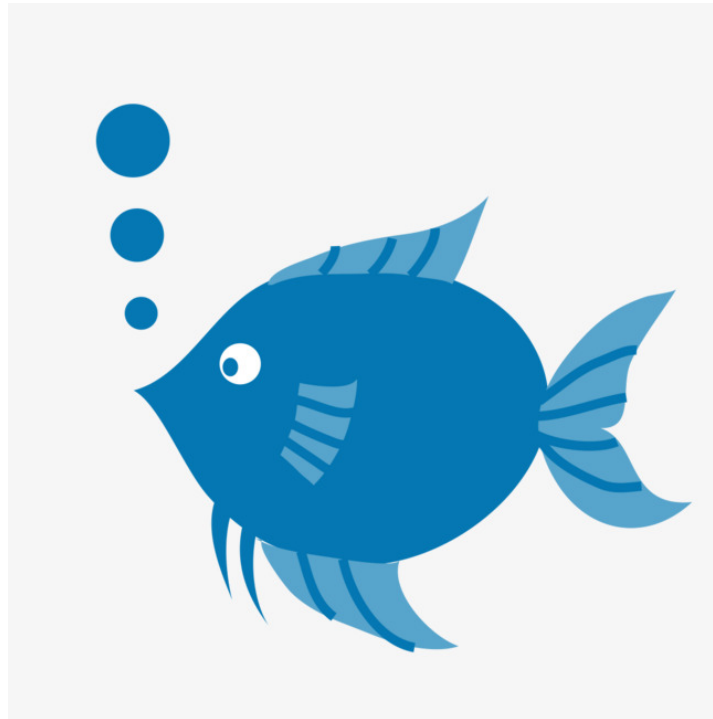
<https://www.vectorstock.com/royalty-free-vector/cartoon-glass-cup-of-lemon-fruit-juice-vector-17979355>

<https://www.alamy.com/stock-photo-grape-juice-in-a-glass-fresh-isolated-on-white-background-fruit-and-125939240.html>

<http://clipart-library.com/clipart/1971889.htm>



Review – Bubble Sort



Graphs are from:
https://pngtree.com/freepng/blue-fish-blowing-bubbles_395239.html



Review – Bubble Sort

Given 9, 5, 7, 2, 4, sort it in an increasing order.

Steps:

0. set the starting index as 0.
1. set the temp = the value of index 0;
2. find the minimum value from index 1-index 4.
3. swap the value of index 0 and minimum value
4. Increase the starting index, repeat steps 1-3.

Code:

```
int int_arr[] = {9, 5, 7, 2, 4};
int k, j, temp;
For(k=0;k<5; k++)
{ temp = int_arr[k];
  for (j = i+1; j< 5; j++)
  { if ( int_arr[j] < temp)
    { int_arr[j] = temp;
      int_arr[k] = int_arr[j];
    }
  }
}
```



Review – Bubble Sort

Given 9, 5, 7, 2, 4, sort it in an increasing order.

Code:

```
int int_arr[] = {9, 5, 7, 2, 4};
int k, j, temp;
For(k=0;k<5; k++)
{ temp = int_arr[k];
  for (j = i+1; j< 5; j++)
  { if ( int_arr[j] < temp)
    { int_arr[j] = temp;
      int_arr[k] = int_arr[j];
    }
  }
}
```

Swap two strings

Code:

```
/* swap two strings*/
char str1[20] = "today";
char str2[20] = "tomorrow";
char temp[20];
strcpy(temp, str1);
strcpy(str1, str2);
strcpy(str2, temp);
printf("%s\n", str1);
```



Dynamic Data Structures (1)

- Structures that expand and contract as a program executes
- Memory is allocated as needed (dynamically)
- We can use dynamic memory allocation to dynamically allocate an array at runtime
 - Thus, there's no need to constrain the maximum size of an array at compile time



Dynamic Data Structures (2)

- Memory can be allocated to store any of the simple or structured data types described throughout this course (i.e. *structs*, *ints*, *doubles*, etc.)
- Separately allocated structured blocks are called *nodes*
- These nodes can be used to form composite structures that expand and contract while the program executes



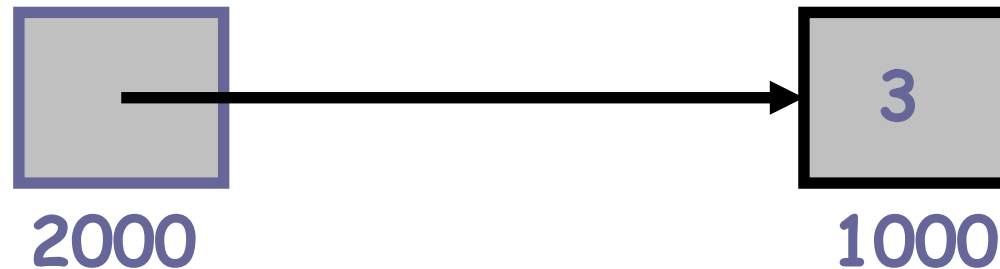
Pointer Review (1)

- A pointer variable contains the address of another cell containing a data value
- Note that a pointer is “useless” unless we make sure that it points somewhere:

Recall functions with pointers.

```
- int num = 3, int *nump = &num;
```

nump points to *num*



- The *direct* value of *num* is 3, while the *direct* value of *nump* is the address (1000) of the memory cell which holds the 3



Pointer Review (2)

- The integer 3 is the *indirect* value of *nump*, this value can be accessed by following the pointer stored in *nump*
- If the indirection, dereferencing, or “pointer-following” operator is applied to a pointer variable, the indirect value of the pointer variable is accessed
- That is, if we apply **nump*, we are able to access the integer value 3
- The next slide summarizes...



Pointer Review (3)



Reference	Explanation	Value
<code>num</code>	Direct value of <code>num</code>	3
<code>nump</code>	Direct value of <code>nump</code>	1000
<code>*nump</code>	Indirect value of <code>nump</code>	3
<code>&nump</code>	Address of <code>nump</code>	2000



Pointers as Function Parameters (1)

- Recall that we define an output parameter to a function by passing the address (&) of the variable to the function
- The output parameter is defined as a pointer in the formal parameter list
- Also, recall that output parameters allow us to return more than one value from a function
- The next slide shows a long division function which uses *quotientp* and *remainderp* as pointers



Pointers as Function Parameters (2)

- Function with Pointers as Output Parameters

```
#include <stdio.h>

void long_division (int dividend, int divisor, int *quotientp, int *remainderp);

int main (void)
{
    int quot, rem;

    long_division (40, 3, &quot, &rem);
    printf ("40 divided by 3 yields quotient %d ", quot);
    printf ("and remainder %d\n", rem);

    return 0;
}

void long_division (int dividend, int divisor, int *quotientp, int *remainderp)
{
    *quotientp = dividend / divisor;
    *remainderp = dividend % divisor;
}
```



Pointers Representing Arrays and Strings (1)

- Consider representing two arrays as follows:
 - `double list_of_nums[20];`
 - `char your_name[40];`
- When we pass either of these arrays to functions, we use the array name without a subscript
- The array name itself represents the **address** of the initial array element



Pointers Representing Arrays and Strings (2)

- Hence, when we pass the array name, we are actually passing the entire array as a pointer
- So, the formal parameter for the string *name* may be declared in two ways:
 - `char name[]`
 - `char *name`
- Note that, in general, it is a good idea to pass the maximum size of the array to the function, e.g.:
 - `void func (char *name, int size);`



Pointers to Structures

- Recall that when we have a pointer to a structure, we can use the indirect component selection operator `->` to access components within the structure

```
typedef struct
{
    double x;
    double y;
} Point;

int main (void)
{
    Point p1, *struct_ptr;
    p1.x = 12.3;
    p1.y = 2.5;

    struct_ptr = &p1;

    struct_ptr->x; /* Access the x component in Point, i.e. 12.3 */
    .
    .
    .
}
```



Dynamic Memory Allocation

- Sometimes we need to allocate memory according to an explicit program request
- The memory must be allocated dynamically
- To do this, we can use the function `malloc()` found in `<stdlib.h>`



malloc Function (1)

- `malloc()` has the following prototype:
 - `void *malloc (size_t size);`
- The argument to *malloc* () is a number indicating the amount of memory space needed to be allocated
- We can apply the `sizeof` operator to a data type to determine the number (the number represents the number of bytes needed)



malloc Function (2)

- The statement:

```
malloc (sizeof (int))
```

allocates exactly enough space to hold one integer

- A call to `malloc` returns a pointer to the block of memory allocated (if no memory can be allocated, `NULL` is returned)
- The return type is `void *` which is generic, thus we need to assign the pointer to a specific type by explicitly typecasting:

```
int *nump = (int *) malloc (sizeof (int));
```



malloc Function (3)

- The area in which the dynamic memory is allocated is called the *heap*
- The *stack* is the area that stores function data when functions are entered
 - Remember that the memory allocated for a given function's data (on the stack) goes away when the function is exited



Accessing a Component of Dynamically Allocated Structure

- Accessing a component of a dynamically allocated structure is done in the same way as accessing a component of a statically allocated structure:

```
Point *struct_ptr =  
    (Point *) malloc (sizeof (Point));
```

```
struct_ptr->x
```

is equivalent to

```
(*struct_ptr).x
```



Dynamic Array Allocation with `calloc`

- `calloc()` may be used to allocate contiguous blocks of memory for array elements
- The function is also found in `<stdlib.h>`
- The function prototype for `calloc()` is as follows:

```
void *calloc (size_t nitems,  
              size_t size);
```



calloc

- The two arguments to `calloc()` are the number of array elements needed and the size of one element, respectively
- As with `malloc`, `calloc` also returns a `void *` to the beginning of the allocated blocks of memory
- Thus, we can allocate 10 blocks of contiguous memory as follows:

```
Point *struct_ptr = (Point *) calloc (10, sizeof  
    (Point));
```

- Note the above statement is similar to

```
Point array[10];
```

except that it is allocated dynamically, instead of statically



Applying Array Notation to Pointers

- If we executed the `calloc` statement on the previous slide, we could then use array notation to access each element in the block of dynamically allocated memory:
 - `struct_ptr[0]` would access the first element in the array and `struct_ptr[9]` would access the last element in the array



Freeing Allocated Memory

- We need to be able to give up memory after we are done using it
- We can use the function `free()` to return the allocated memory to the heap
- For example:

```
free (struct_ptr);
```

returns the dynamically-allocated memory block pointed to by `struct_ptr`
- "Memory leaks" occur if we do not free memory before the local variable that accesses the allocated memory goes out of scope
- Memory leaks are a common source of run-time program bugs; they can also slow down a program's execution



References

- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (8th Ed.)*, Addison-Wesley, 2016.
- P.J. Deitel & H.M. Deitel, *C How to Program (7th Ed.)*, Pearson Education , Inc., 2013.



Collaborators

- Chris Hundhausen
- Andrew O'Fallon

