Lecture Notes for Lecture 11 of CS 5001 (Foundations of CS) for the Fall, 2018 session at the Northeastern University Silicon Valley Campus.

Memory Management

Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 10 Review

- Defining types as aggregations of heterogeneous values allows modeling complex, real-world data about an entity.
- C provides a struct for creating aggregated types that are comprised of fields specified by variable declarations.
- Struct types have their own name space and require using ‘struct’ qualifier wherever instances of the type are defined
- Using typedef with struct eliminates the need for a ‘struct’ qualifier by putting the enum in the regular type name space.
- Initialize structs using positional or designated initializer lists.
- C makes “shallow copy” of a struct on assignment, when passing as a parameter, or returning from a function; avoid by using pointers.
- A struct can be embedded in another struct, or it can be shared by pointing to it from multiple other structs.

Memory Management

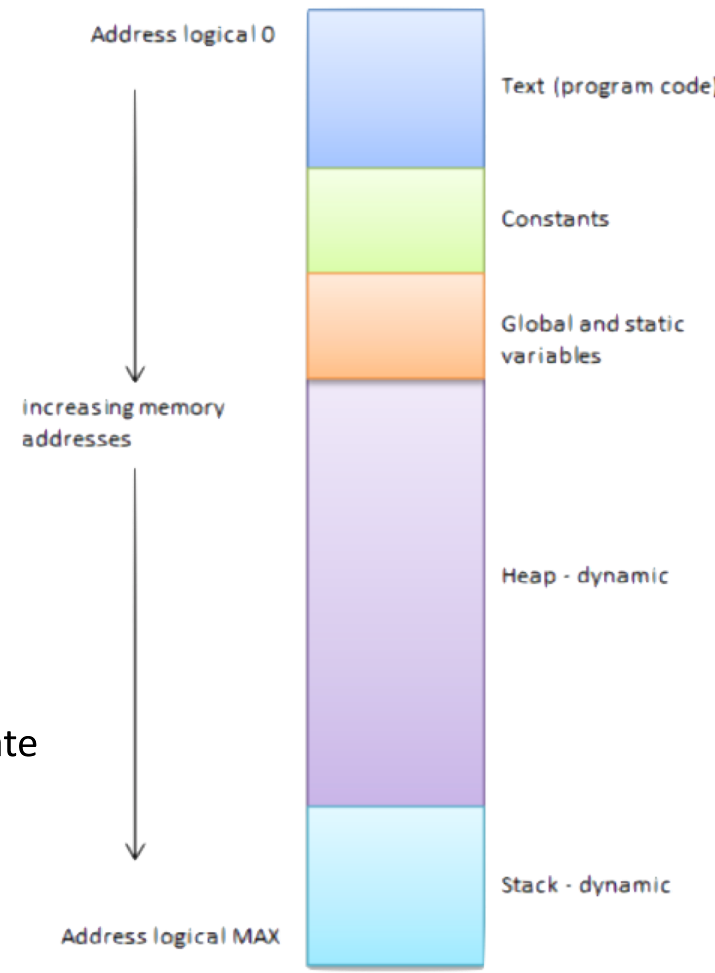
- In this lecture, we will learn about how memory is managed in a C program, including the local and global variables, constants, and the program code itself.
- We will also learn about functions that manage a special region of memory that is independent of global storage and the local storage within a given function.
- Finally, we will learn techniques for using this memory to create data, including arrays and structs, whose lifetime is under control of the programmer.

Memory Management

C Memory Management

- When a C program is loaded, it is organized into areas of memory, called segments:
  - Program Code
    - Compiled code for the program
  - Constants
    - Literal strings and other fixed constants
  - Global and static variables
    - Variables declared globally or statically
  - Dynamic memory (heap)
    - A pool of memory programmers can allocate
  - Local variables ("stack")
    - Storage for local variables in functions

C Program Memory Map



Memory Management

C Memory Management

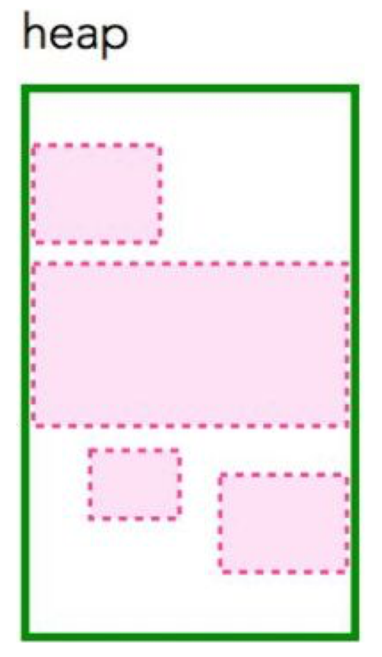
- Compiled program occupies memory in the *text segment*, and we can even have a pointer to a function in that area of memory.
- Literal strings and other constants are in a *literal segment* that cannot be modified, such as a pointer to a string literal.
- Global and static variables are automatically created in the *global segment*, which exist for as long as the program is running
- Local variables are automatically created in the *stack segment* when execution enters a function, and the storage automatically goes away when the function returns.
- The last area of memory is the *dynamic segment,* where programmers can allocate storage. What is it and how is it used?

Memory Management

What is Dynamic Memory?

- *Dynamic memory* is the storage in the dynamic segment. It is accessed through functions that manage the storage pool, sometimes referred to as the *heap*.

- A program can allocate a block of memory from the pool at runtime, use it for as long as it is needed, and then return the storage to the pool for reuse.



- The lifetime of local storage is a function call, and global storage lasts for the lifetime of the program, but the lifetime of dynamic memory is under program control.

Memory Management

Why Use Dynamic Memory?

- Local and global storage are useful when a fixed numbers of a type are required, whose lifetime is either the function or the lifetime of the program.
- With dynamic storage, arbitrary numbers of a type can be created whose lifetime transcends a function but is shorter than the lifetime of a program.
- Instances of a type can be created by a function at any point in a program, and then used by other functions until the instance is no longer needed.
- This makes creating instance of types in dynamic memory ideal when their lifetimes need to be under program control.

Memory Management

How is Dynamic Memory Used?

- Allocate dynamic memory using the standard C function *void *malloc(size_t nbytes)*, and free it using *void free(void *)*.
- Reference the allocated memory by assigning the pointer returned from *malloc* to a pointer of the type being allocated.
- Here is an example of allocating and, initializing the string, printing it, then returning the storage to the heap.

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

char *heapHello = malloc(strlen("hello")+1);            // allocate heap storage
strcpy(heapHello, "hello");                             // initialize allocated storage
printf("heapHello : \"%s\"\n", heapHello);              // print string
free(heapHello);                                        // return storage to heap
```

Memory Management

How is Dynamic Memory Used?

- Unlike local storage, heap storage remains active until it is returned to the heap by calling *free()*.
- The free() function can only be called with a pointer to storage that was allocated from the heap. A pointer to any other storage or to already freed storage is a runtime error.
- For convenience, the NULL pointer can be passed to free() and it performs no action. This is one of the few C functions with this property.

Memory Management

How is Dynamic Memory Used?

- Here is the standard C function *char *strdup(const char*)* that allocates, initializes, and returns a copy of a string.

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

/**
 * Allocates and initialize copy of str from dynamic memory.
 * @param str the string to copy
 * @return a copy of str from dynamic memory or NULL if str is NULL
 */
char *strdup(const char* str) {
    if (str == NULL) {
        return NULL;                    // special case NULL str
    }
    char *newstr = malloc(strlen(str)+1);     // allocate new string of same length
    return strcpy(newstr, str);               // copy to new string – strcpy returns newstr

    //    return str == NULL ? NULL : strcpy(malloc(strlen(str)+1), str);
}
```

Memory Management

How is Dynamic Memory Used?

- The storage for the string copy returned by *strdup()* must be returned to the heap by calling *free()* on the returned pointer.
- Once allocated storage is returned, it can no longer be accessed.

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

/** Test strdup function */
int main(void) {
    char *heapHello = strdup("hello");              // allocate and initialize string ”hello”
    printf("heapHello : \"%s\"\n", heapHello);      // print string
    free(heapHello);                                // return storage to heap
    char *heapNULL = strdup(NULL);                  // allocation and initialize NULL string
    printf("heapNULL : \"%s\"\n", heapNULL);        // print string
    free(heapNULL);                                 // return storage to heap
}
```

Memory Management

How is Dynamic Memory Used?

- The size of the memory allocated from the heap is only guaranteed to be the requested size. The standard C function *void* realloc(void*)* can be used to resize it.
- If necessary, the content of the previously allocated storage is copied to new storage and the previous storage is freed.

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

char *heapHello = malloc(strlen("hello")+1);          // allocate heap storage
strcpy(heapHello, "hello");                           // initialize allocated storage
printf("heapHello : \"%s\"\n", heapHello);            // print string

heapHello = realloc(strlen("hello world")+1);         // extend allocated storage
strcat(heapHello, " world");                          // append to extended storage
printf("heapHello : \"%s\"\n", heapHello);            // print string
free(heapHello);                                      // return storage to heap
```

Memory Management

Example: Allocating Journal Struct

- Any C type can allocated in dynamic memory. Here is an example of allocating a Journal struct from dynamic storage:

```
/** Typedef for ISSN: nnnn-nnnN */
typedef uint32_t Issn;

/** Struct that defines a Publisher */
typedef struct {        // “anonymous struct”
    char name[100];         // name of publisher
} Publisher;            // only known by its typedef name

/** Struct that defines a Journal */
typedef struct {        // “anonymous struct”
    char name[100];         // journal name
    Issn issn;              // defined type for the ISSN of journal
    Publisher publisher;    // defined type for the journal publisher
} Journal;              // only known by its typedef name
```

Memory Management

Example: Allocating Journal Struct

```
/**
 * Get string representation of ISSN.
 * @param issn the ISSN value
 * @param issnChars array to receive the ISSN characters
 * @return pointer to ISSN chars
 */
char *issnToString(Issn issn, char issnChars[]) {
    // make use of underlying uint32_t type of ISSN internally
    sprintf(issnChars, "%04x-%04x", issn >> 16, issn & 0xFFFF);
    if (issnChars[8] == 'a') {
        issnChars[8] = 'X';  // issn uses ‘X’ rather than ‘a’ for 10 for check digit
    }
    return issnChars;
}
```

Memory Management

Example: Allocating Journal Struct

```
/**
 * Print a Journal.
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString(journal->issn, issnChars);
    printf(”journal name: '%s'\nISSN: %s\npublisher name: '%s'\n",
            journal->name, issnString, journal->publisher.name);
}
```

Memory Management

Example: Allocating Journal Struct

```
/** Create a new Journal.
  * @param name the journal name
  * @param issn the journal issn
  * @param pubName the publisher name
  * @return the new journal
  */
Journal *newJournal(const char *name, Issn issn, const char *pubName) {
    Journal *journal = malloc(sizeof(Journal));
    strcpy(journal->name, name);
    journal->issn = issn;
    strcpy(journal->publisher.name, pubName);
    return journal;
}

/** Delete the journal
 * @param journal the journal to delete
 */
void deleteJournal(Journal *journal) {
    free(journal);
}
```

Memory Management

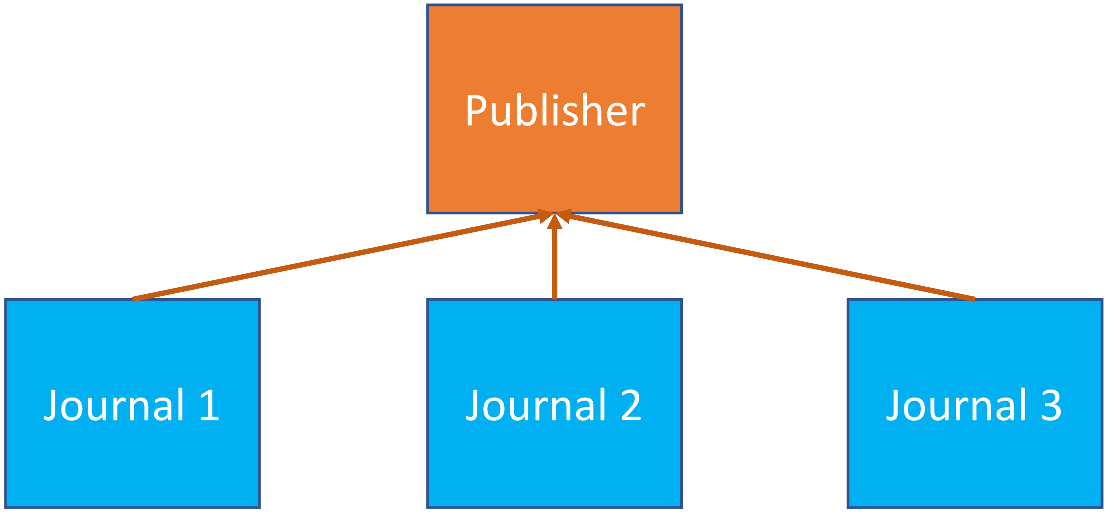
Example: Allocating Journal Struct

```
/** Test dynamic allocation of journal */
int main(void) {
    const char *ngs = "National Geographic Society";

    Journal *nationalGeographic =
        newJournal("National Geographic", 0x00279358, ngs);
    Journal *nationalGeographicExplorer =
        newJournal("National Geographic Explorer", 0x15413357, ngs);
    Journal *nationalGeographicKids =
        newJournal("National Geographic Kids", 0x15423042, ngs);

    printJournal(nationalGeographic);
    printJournal(nationalGeographicExplorer);
    printJournal(nationalGeographicKids);

    deleteJournal(nationalGeographic);
    deleteJournal(nationalGeographicExplorer);
    deleteJournal(nationalGeographicKids);
}
```

Memory Management

Example: One-To-Many Unidirectional Relationship

- We also looked at having the Publisher field point to its publisher, so that journals can share a publisher struct.
- This is *one-to-many relationship*, and is often notated as 1:* to show that one publisher can have 0 or more journals. It is *unidirectional* because the Publisher does not point back.



Memory Management

Example: One-To-Many Unidirectional Relationship

- We also looked at making the Publisher field point to its publisher, so that many journals can share a publisher struct.

```
/** Typedef for ISSN: nnnn-nnnN */
typedef uint32_t Issn;

/** Struct that defines a Publisher */
typedef struct {        // “anonymous struct”
    char name[100];         // name of publisher
} Publisher;            // only known by its typedef name

/** Struct that defines a Journal */
typedef struct {        // “anonymous struct”
    char name[100];         // journal name
    Issn issn;              // defined type for the ISSN of journal
    Publisher *publisher;   // defined type for the journal publisher
} Journal;              // only known by its typedef name
```

Memory Management

Example: One-To-Many Unidirectional Relationship

```
/**
 * Print a Publisher.
 * @param publisher the publisher to print
 */
void printPublisher(const Publisher *publisher) {
    printf(”publisher name: ’%s’\n”, publisher->name);
}

/**
 * Print a Journal.
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString(journal->issn, issnChars);
    printf(”journal name: '%s'\nISSN: %s\n",  journal->name, issnString);
    printPublisher(journal->publisher);  // journal publisher pointer field
}
```

Memory Management

Example: One-To-Many Unidirectional Relationship

```
/**
  * Create a new Publisher.
  * @param name the publisher name
  */
Publisher *newPublisher(const char *name) {
    Publisher *publisher = malloc(sizeof(Publisher));
    strcpy(publisher->name, name);
    return publisher;
}

/**
 * Delete the publisher
 * @param publisher the publisher to delete
 */
void deletePublisher(Publisher *publisher) {
    free(publisher);
}
```

Memory Management

Example: One-To-Many Unidirectional Relationship

```
/** Create a new Journal.
  * @param name the journal name
  * @param issn the journal issn
  * @param pubName the publisher name
  */
Journal *newJournal(const char *name, Issn issn, Publisher *publisher) {
    Journal *journal = malloc(sizeof(Journal));
    strcpy(journal->name, name);
    journal->issn = issn;
    journal->publisher = publisher;
    return journal;
}

/** Delete the journal
 * @param journal the journal to delete
 */
void deleteJournal(Journal *journal) {
    free(journal);
}
```

Memory Management

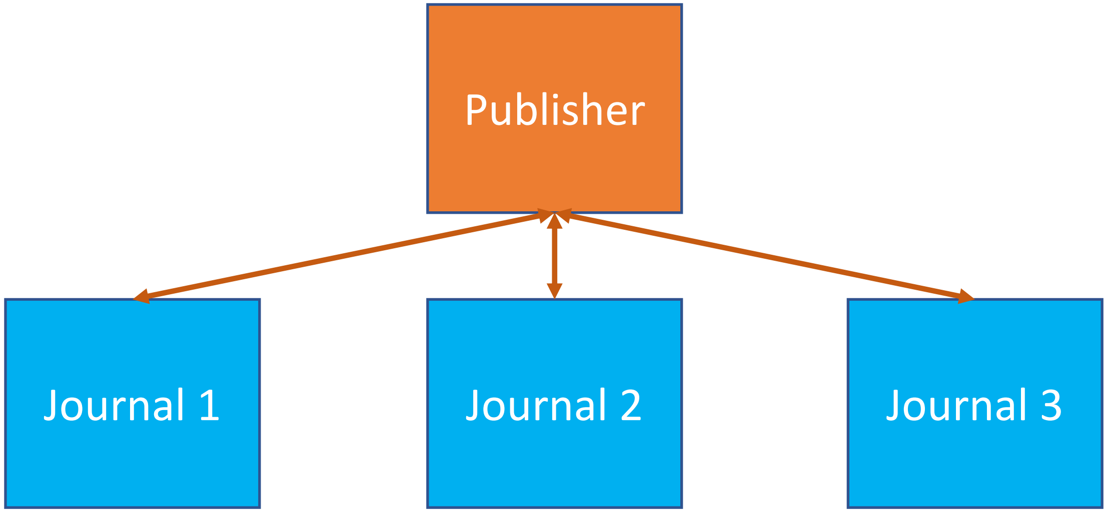
Example: One-To-Many Unidirectional Relationship

```
/** Test dynamic allocation of publisher and journal */
int main(void) {
    Publisher *ngs = newPublisher("National Geographic Society");

    Journal *nationalGeographic =
        newJournal("National Geographic", 0x00279358, ngs);
    Journal *nationalGeographicExplorer =
        newJournal("National Geographic Explorer", 0x15413357, ngs);
    Journal *nationalGeographicKids =
        newJournal("National Geographic Kids", 0x15423042, ngs);

    printJournal(nationalGeographic);
    printJournal(nationalGeographicExplorer);
    printJournal(nationalGeographicKids);

    deleteJournal(nationalGeographic);
    deleteJournal(nationalGeographicExplorer);
    deleteJournal(nationalGeographicKids);
    deletePublisher(ngs);
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

- We can make the relationship *bidirectional* by also having the Publisher also point to its Journals.
- Given a Journal, we can locate its Publisher, and given a Publisher, we can locate all of its Journals.



Memory Management

Example: One-To-Many Bidirectional Relationship

- The easiest way to point to the journals is to add an array of Journal pointers to the Publisher. When creating a Journal, we must also add a pointer to it in the *journals* array.
- How do we determine the *journals* array index? The easiest way is to also add a *nJournals* field with the current number of journals, and use it as the index of the next journal entry.

```
/** Struct that defines a Journal */
typedef struct {        // "anonymous struct"
    char name[100];             // name of publisher
    Journal *journals[10]       // up to 10 journals
    unsigned int nJournals;     // number of journals
} Publisher;            // only known by its typedef name
```

Memory Management

Example: One-To-Many Bidirectional Relationship

- Because Journal has a Publisher* field and Publisher now has a Journal* array field, the circular reference must be resolved.
- The solution is to add a *forward declaration* of the Journal struct and typedef before the Publisher definition. This works because Publisher points to rather than embeds a Journal.

```
/** Forward declaration of Journal struct and typedef */
typedef struct Journal Journal;      // cannot omit struct name
    …
/** Struct that defines Journal */
typedef struct Journal {   // struct required, but repeating typedef ok too
    ….
} Journal;
```

Memory Management

Example: One-To-Many Unidirectional Relationship

```
/**
  * Create a new Publisher.
  * @param name the publisher name
  */
Publisher *newPublisher(const char *name) {
    Publisher *publisher = malloc(sizeof(Publisher));
    strcpy(publisher->name, name);
    publisher->nJournals = 0;      // initially no journals
    return publisher;
}

/**
 * Delete the publisher.
 * @param publisher the publisher to delete
 */
void deletePublisher(Publisher *publisher) {
    free(publisher);
}
```

Memory Management

Example: One-To-Many Unidirectional Relationship

```
/**
 * Add a journal to a publisher.
 * @param publisher the publisher
 * @param journal the journal to add
 */
void addJournalToPublisher(Publisher *publisher, Journal *journal) {
    assert(publisher->nJournals < 10);  // program exits if too many journals
    publisher->journals[publisher->nJournals++] = journal;
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

- Prints only the publisher and the journal information.

```
/**
 * Print a Publisher.
 * @param publisher the publisher to print
 */
void printPublisherInfo(const Publisher *publisher) {
    printf(”publisher name: ’%s’\n”, publisher->name);
}
/**
 * Print a Journal only without publisher info.
 * @param journal the journal to print
 */
void printJournalInfo(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString(journal->issn, issnChars);
    printf("journal name: '%s'\nISSN: %s\n", journal->name, issnString);
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

- This function prints the journal with its publisher info.

```
/**
 * Print a Journal and its publisher
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    printJournalInfo(journal);
    printPublisherInfo(journal->publisher);
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

- This function prints the publisher with its journals info.

```
/**
 * Print a Publisher and its journals.
 * @param publisher the publisher to print
 */
void printPublisher(const Publisher *publisher) {
    printPublisherInfo(publisher);

    for (int jnl = 0; jnl < publisher->nJournals; jnl++) {
        printJournalInfo(publisher->journals[jnl]);
    }
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

```
/**
 * Create and initialize a journal for a publisher.
 * @param name the journal name
 * @param issn the journal issn
 * @param publisher the journal publisher
 * @return the journal
 */
Journal *newJournal(const char *name, Issn issn, Publisher *publisher) {
    Journal *journal = malloc(sizeof(Journal));
    strcpy(journal->name, name);
    journal->issn = issn;
    journal->publisher = publisher;

    addJournalToPublisher(publisher, journal);
    return journal;
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

```
/**
 * Delete the journal.
 * @param journal the journal to delete
 */
void deleteJournal(Journal *journal) {
    free(journal);
}
```

Memory Management

Example: One-To-Many Bidirectional Relationship

```
/** Test dynamic allocation of publisher and journal */
int main(void) {
    Publisher *ngs = newPublisher("National Geographic Society");

    Journal *nationalGeographic =
          newJournal("National Geographic", 0x00279358, ngs);
    Journal *nationalGeographicExplorer =
          newJournal("National Geographic Explorer", 0x15413357, ngs);
    Journal *nationalGeographicKids =
          newJournal("National Geographic Kids", 0x15423042, ngs);
```

Memory Management

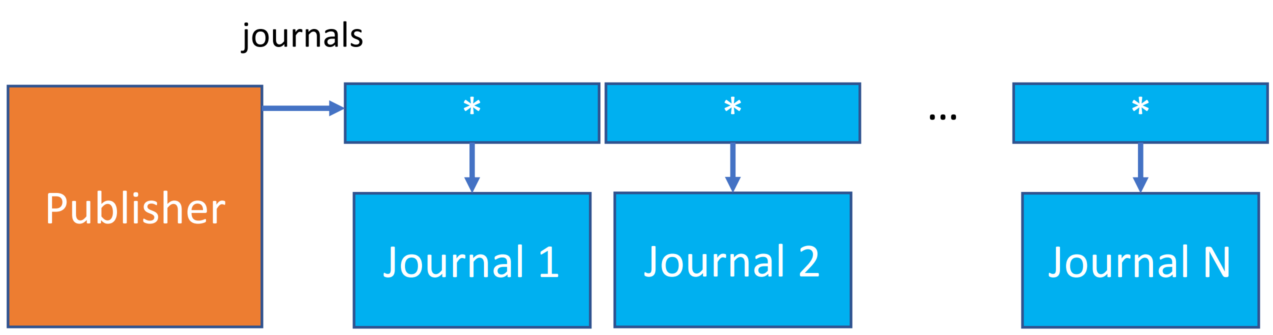
Example: One-To-Many Bidirectional Relationship

```
    printJournal(nationalGeographic);
    printJournal(nationalGeographicExplorer);
    printJournal(nationalGeographicKids);
    printJournalsForPublisher(ngs);

    deleteJournal(nationalGeographic);
    deleteJournal(nationalGeographicExplorer);
    deleteJournal(nationalGeographicKids);
    deletePublisher(ngs);
}
```

Memory Management

Example: Arbitrary Number of Journals

- One of the problems with our bidirectional implementation is that it limits the number of journals to a fixed number.
- Accommodating an arbitrary number of journals requires being able to resize the *journals* array in the Publisher struct.
- We do this by dynamically allocating the *journals* array, and reallocating it if more elements are required. We did something similar earlier with the char array used as a string.



Memory Management

Example: Arbitrary Number of Journals

- Here is the new definition of Publisher:

```
/** Struct that defines a Journal */
typedef struct {        // "anonymous struct"
    char name[100];              // name of publisher
    Journal **journals;          // array of journal pointers
    unsigned int nJournals;      // number of journals
    unsigned int maxJournals;    // maximum number of journals
} Publisher;            // only known by its typedef name
```

- The *journals* field now point to an array of Journal pointers that will be allocated dynamically.
- The *maxJournals* field keeps track of the array capacity. If the array is full, it must be resized before adding another journal.

Memory Management

Example: Arbitrary Number of Journals

- Here is how the Publisher is initialized.

```
/**
 * Create and initialize a publisher.
 * @param name the publisher name
 * @return the publisher
 */
Publisher *newPublisher(const char *name) {
    Publisher *publisher = malloc(sizeof(Publisher));
    strcpy(publisher->name, name);
    publisher->nJournals = 0;          // no journals
    publisher->maxJournals = 2;        // initially, two slots available
    publisher->journals = malloc(publisher->maxJournals * sizeof(Journal*));
    return publisher;
}
```

Memory Management

Example: Arbitrary Number of Journals

- Here is how the Publisher is deleted.

```
/**
 * Delete the publisher.
 * @param publisher the publisher to delete
 */
void deletePublisher(Publisher *publisher) {
    free(publisher->journals);  // first free dynamic array
    free(publisher);
}
```

Memory Management

Arbitrary Number of Journals

- Here is how a Journal is added to a Publisher.

```
/**
 * Add a journal to a publisher.
 * @param publisher the publisher
 * @param journal the journal to add
 */
void addJournalToPublisher(Publisher *publisher, Journal *journal) {
    if (publisher->nJournals >= publisher->maxJournals) {  // out of space
        publisher->maxJournals *= 2;       // double available size
        publisher->journals =              // grow storage to new available size
            realloc(publisher->journals, publisher->maxJournals * sizeof(Journal*));
    }
    publisher->journals[publisher->nJournals++] = journal;
}
```

Memory Management

Example: Arbitrary Number of Journals

```
/** Test dynamic allocation of publisher and journal */
int main(void) {
    Publisher *ngs = newPublisher("National Geographic Society");

    Journal *nationalGeographic =
        newJournal("National Geographic", 0x00279358, ngs);
    Journal *nationalGeographicExplorer =
        newJournal("National Geographic Explorer", 0x15413357, ngs);
    Journal *nationalGeographicKids =
        newJournal("National Geographic Kids", 0x15423042, ngs);
```

Memory Management

Example: Arbitrary Number of Journals

```
    printJournal(nationalGeographic);
    printJournal(nationalGeographicExplorer);
    printJournal(nationalGeographicKids);
    printJournalsForPublisher(ngs);

    deleteJournal(nationalGeographic);
    deleteJournal(nationalGeographicExplorer);
    deleteJournal(nationalGeographicKids);
    deletePublisher(ngs);
}
```