Dynamic Memory Allocation in C

CS 2060

Prof. Jonathan Ventura

Static memory allocation

- So far, we have only dealt with *static* memory allocation:

```
int value;  // 4 bytes
char string[10];    // 10 bytes
struct Point { int x, int y };  // 8 bytes
```

- In each of these cases, the size of the variable is known at compile-time.
- We know exactly how much memory is needed to run the program.

Static memory allocation

- What if we don't know how much space we need until run-time?

```
// read number of values to be entered
int count;
scanf("%d", &count);

// now we need to store count integers somewhere...
```

Static memory allocation

- One option is to allocate a fixed-size buffer. What are the disadvantages of this?

```
// allocate fixed-size buffer
int input[1024];

// read values into buffer
for ( int i = 0; i < count; i++ ) {
    scanf("%d",&input[i]);
}
```

Dynamic memory allocation

```c
// allocate fixed-size buffer
int input[1024];

// read values into buffer
for ( int i = 0; i < count; i++ ) {
    scanf("%d",&input[i]);
}
```

- A fixed-size buffer may waste space (if `count < 1024`)
- Or, we may not have enough space (if `count > 1024`)

Dynamic memory allocation

- With *dynamic memory allocation* we can *request* an arbitrary of bytes at run time.

```
void * malloc( size_t size ); // request size bytes
```

- `malloc` returns a pointer to `size` bytes which are allocated for our usage.

Dynamic memory allocation example

```c
// read number of values to be entered
int count;
scanf("%d", &count);

// allocate memory
int *input = malloc( sizeof(int)*count );

// read values
for ( int i = 0; i < count; i++ ) scanf("%d",&input[i]);
```

Dynamic memory allocation

- We don't need to allocate an array, necessarily.
- We can allocate space for a single `struct` or even a single `int`.

```
struct Point *pt = malloc( sizeof(struct Point) );

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

Dynamic memory allocation

- Dynamic memory allocation is useful when we need data to *persist* beyond the life of a function call.
- Similar to global variables, dynamically allocated memory can be accessed anywhere in a program.
- We just need a pointer to the allocated block of memory.

Dynamic memory allocation example

```
struct Point { int x, int y };

struct Point * makePoint( int x, int y )
{
    // allocate and initialize a struct
    struct Point *pt = malloc( sizeof(struct Point) );
    pt->x = x;  pt->y = y;

    // return a pointer
    return pt;
}
```

Dynamic memory allocation example

```
int main()
{
    // create ten Point pointers
    struct Point *pts[10];

    for ( int i = 0; i < 10; i++ ) {
        pts[i] = makePoint( 0, 0 );
    }
}
```

Dynamic memory allocation example

```
struct Message {
    int length; // number of bytes in data
    char *data;
};
```

Dynamic memory allocation example

```
struct Message * makeMessage(
            int length,
            const char *data )
{
    struct Message *message =
    malloc( sizeof( struct Message ) );

    // How do I copy the provided data into the message?

    return message;
};
```

Dynamic memory allocation example

```
struct Message * makeMessage(
            int length,
            const char *data )
{
    struct Message *message =
    malloc( sizeof( struct Message ) );

    // Does this work?
    memcpy( message->data, data, length );
    message->length = length;

    return message;
};
```

Dynamic memory allocation example

- In this case, `struct Message` contains a pointer to data, not the data itself.

```
struct Message {
    int length; // number of bytes in data
    char *data;
};
```

- What is `sizeof(struct Message)` ?

Dynamic memory allocation example

- In this case, `struct Message` contains a pointer to data, not the data itself.

```
struct Message {
    int length; // 4-byte integer
    char *data; // 8-byte pointer
};
```

- What is `sizeof(struct Message)` ?

  $= 4 + 8 = 12$ bytes

Dynamic memory allocation example

```
struct Message * makeMessage(
            int length,
            const char *data )
{
    struct Message *message =
    malloc( sizeof( struct Message ) );

    // Does this work?
    memcpy( message->data, data, length );
    message->length = length;

    return message;
};
```

Dynamic memory allocation example

- Allocating the struct will not allocate space for the data, just a pointer.
- The `data` will be uninitialized.

```
// copy to uninitialized pointer -- bad
memcpy( message->data, data, length );
message->length = length;
```

- How can we fix it?

Dynamic memory allocation example

- We need to allocate the approriate amount of memory to store the data.

```
// allocate space
message->data = malloc( length );

// copy data
memcpy( message->data, data, length );

// set length
message->length = length;
```

Dynamic memory allocation example

```
struct Message * makeMessage(
            int length,
            const char *data )
{
    struct Message *message =
    malloc( sizeof( struct Message ) );

    message->data = malloc( length );
    memcpy( message->data, data, length );
    message->length = length;

    return message;
};
```

Dynamic memory allocation example

- How would this be different?

```
message->data = data;
message->length = length;
```

Dynamic memory allocation example

- This copies the *pointer*, not the data.

```
message->data = data;
message->length = length;
```

- This is a *shallow copy*.
- We now have two pointers to the data, not two copies of the data itself.

Dynamic memory allocation example

```
struct DictionaryEntry {
    char *word;
    char *definition;
};
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition )
{
    // allocate a new struct

    // copy the word and definition to it

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    // copy the word and definition to it

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    // need to allocate space for strings
    // how big to make them?

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    // is this right?
    entry->word = malloc( strlen(word) );
    entry->definition = malloc( strlen(definition) );

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    // we need one extra space for
    // the null terminator
    entry->word = malloc( strlen(word) + 1 );
    entry->definition = malloc( strlen(definition) + 1 );

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    entry->word = malloc( strlen(word) + 1 );
    entry->definition = malloc( strlen(definition) + 1 );

    // how to copy strings to entry?

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    entry->word = malloc( strlen(word) + 1 );
    entry->definition = malloc( strlen(definition) + 1 );

    memcpy( entry->word, word, strlen(word)+1 );
    memcpy( entry->definition, definition,
        strlen(definition)+1 );

    return entry;
}
```

Dynamic memory allocation example

```
struct DictionaryEntry *
makeEntry( const char *word, char *definition ) {
    struct DictionaryEntry *entry =
    malloc( sizeof(struct DictionaryEntry) );

    entry->word = malloc( strlen(word) + 1 );
    entry->definition = malloc( strlen(definition) + 1 );

    // or just use strcpy:
    strcpy( entry->word, word );
    strcpy( entry->definition, definition );

    return entry;
}
```

Releasing memory

- We should *release* allocated bytes when the memory is no longer needed.

```
void free( void *ptr ); // release bytes
```

- `free` releases the memory allocated at `ptr` so other processes can use it.

Releasing memory example

```
// calculate average
float average = 0;
for ( int i = 0; i < count; i++ ) average += input[i];
average /= count;

// release memory
free( input );
```

Releasing memory example

```
void freeMessage( struct Message *message )
{
    // release message data
    free( message->data );

    // release message
    free( message );
}
```

Releasing memory example

```
void freeEntry( struct DictionaryEntry *entry )
{
    // release entry strings
    free( entry->word );
    free( entry->definition );

    // release entry
    free( entry );
}
```

Releasing memory

- Once the memory has been released (`freed`), we should not try to access it.

```
free(input);
input[0] = 10; // bad access
```

Releasing memory

- It is recommended to set a pointer to `NULL` after `free`ing it. Why?

```
free(input);
input = NULL;
```

Releasing memory

- It is easy to check if a pointer is `NULL`.
- However, there is no way to tell if a non-`NULL` pointer points to properly allocated memory.

```
// returns 1 upon success
int processData( int *array ) {
    // guard against NULL pointer
    if ( array == NULL ) return 0;

    // do data processing...

    // return success
    return 1;
}
```

Releasing memory

- `free` will do nothing if `NULL` is passed to it.

```
free(NULL); // nothing happens (no error)
```

Releasing memory

- However, trying to `free` an uninitialized pointer will cause an error.
- Same for calling `free` on an already-freed memory block.

```
char *ptr;
free(ptr);  // error

ptr = malloc(10);
free(ptr);
free(ptr);  // error
```

Releasing memory

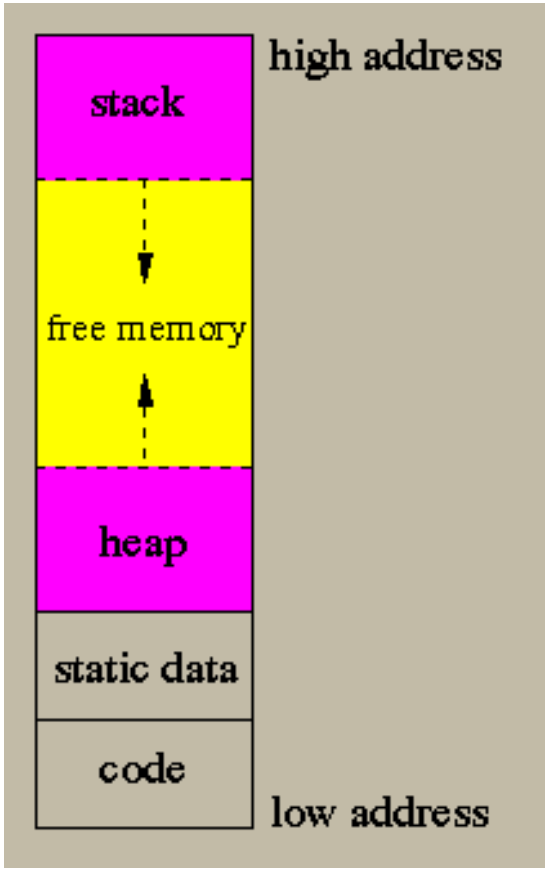
- This is another reason why pointers should be initialized to NULL and set to NULL after they are freed.

```
char *ptr = NULL;
free(ptr);  // no error

ptr = malloc(10);
free(ptr); ptr = NULL;
free(ptr);  // no error
```

Stack and heap

- Remember the stack and the heap?
- Function call frames are pushed onto the **stack**.
  - The stack grows downward.
  - Stack frames have static size determined at compile-time.
- Dynamic memory allocations take place in the **heap**.
  - The heap grows upward.
  - Dynamic memory allocation sizes are determined at run-time.



Stack and heap

- Here is an illustration of three `malloc` calls on the heap:

```
char *block1 = malloc(10);
char *block2 = malloc(10);
char *block3 = malloc(10);
```

Byte number	Name	Size
20	`block3`	10
10	`block2`	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- What happens when we free `block2`?

```
char *block1 = malloc(10);
char *block2 = malloc(10);
char *block3 = malloc(10);
free(block2);
```

Byte number	Name	Size
20	`block3`	10
10	`block2`	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- Now we have empty space in the middle of the heap.

```
char *block1 = malloc(10);
char *block2 = malloc(10);
char *block3 = malloc(10);
free(block2);
```

Byte number	Name	Size
20	`block3`	10
10	*empty*	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- This is called **memory fragmentation**. Why is it a problem?

Byte number	Name	Size
20	`block3`	10
10	*empty*	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- What if we try to allocate a block of 11 bytes?

```
free(block2);
block2 = malloc(11);
```

Byte number	Name	Size
20	`block3`	10
10	*empty*	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- We can't fit 11 bytes into the 10-byte hole in the heap; we have to put it on top.

```
free(block2);
block2 = malloc(11);
```

Byte number	Name	Size
30	`block2`	11
20	`block3`	10
10	*empty*	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- Why can't we re-organize the heap?
- Move `block3` down so that it takes up the hole left by `block2` ?

Byte number	Name	Size
20	`block3`	10
10	*empty*	10
0	`block1`	10

Table: Memory allocations on the heap

Memory fragmentation

- The problem is our program is holding pointers to `block1` and `block3`.
- We would need to update those memory addresses everywhere they are stored in the program – this is not feasible.

```
char *block1 = malloc(10); // = 0
char *block2 = malloc(10); // = 10
char *block3 = malloc(10); // = 20
free(block2);
```

Byte number	Name	Size
20	`block3`	10
10	*empty*	10
0	`block1`	10

Table: Memory allocations on the heap

Pros/cons

- Upsides of dynamic memory allocation:
  - Don't need to know size of memory needed at compile time
  - Allocated memory persists beyond function call
  - Useful for implementing dynamic data structures (next lecture)

Pros/cons

- Upsides of dynamic memory allocation:
  - Don't need to know size of memory needed at compile time
  - Allocated memory persists beyond function call
  - Useful for implementing dynamic data structures (next lecture)
- Downsides of dynamic memory allocation:
  - Overhead of `malloc` and `free` calls
  - Memory fragmentation
  - Introduces bugs, vulnerabilities

Pros/cons

- Upsides of dynamic memory allocation:
  - Don't need to know size of memory needed at compile time
  - Allocated memory persists beyond function call
  - Useful for implementing dynamic data structures (next lecture)
- Downsides of dynamic memory allocation:
  - Overhead of `malloc` and `free` calls
  - Memory fragmentation
  - Introduces bugs, vulnerabilities
- Modern operating systems handle dynamic memory allocations very well – virtual memory, sandboxing

Pros/cons

- Upsides of dynamic memory allocation:
  - Don't need to know size of memory needed at compile time
  - Allocated memory persists beyond function call
  - Useful for implementing dynamic data structures (next lecture)
- Downsides of dynamic memory allocation:
  - Overhead of `malloc` and `free` calls
  - Memory fragmentation
  - Introduces bugs, vulnerabilities
- Modern operating systems handle dynamic memory allocations very well – virtual memory, sandboxing
- Dynamic memory allocation is discouraged on embedded systems (like Arduino)

Allocating an array

- There are a couple more memory allocation functions to be aware of.
- `calloc` allocates an array of chunks of the same size.

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

- This effectively allocates `count * size` bytes of memory.

Allocating an array

- `calloc` is like a dynamic memory replacement for arrays:

```
// allocate array on stack
int array1[100];

// allocate array on heap
int *array2 = calloc( 100, sizeof(int) );
```

Resizing a block of memory

- `realloc` resizes a block of pre-allocated memory.

```
void * realloc( void *ptr, size_t size );
```

- `ptr` should point to a block of memory allocated with `malloc` or `calloc`.
- `size` is the new size of the block.

Resizing a block of memory

- The values in the block are preserved:
  - If the new block is bigger, the first bytes will be the same as before.
  - If the new block is smaller, the last bytes will be truncated.

Resizing a block of memory

```c
// allocate 100 bytes
char *string = malloc( 100 );

// copy in string
strcpy( string, "hello!" );

// make buffer bigger
string = realloc( string, 200 );

// print string -- will retain "hello!"
printf("%s\n", string );
```

Next time

- HW8 due Wednesday night
- Quiz on file I/O – remember: open book, open notes
- Data structures in C: linked list, maybe stack