Lecture 16: Advanced Use of Pointers

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology
Winter 2018

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro.
Some content taken from K.N. King's slides based on course text book]

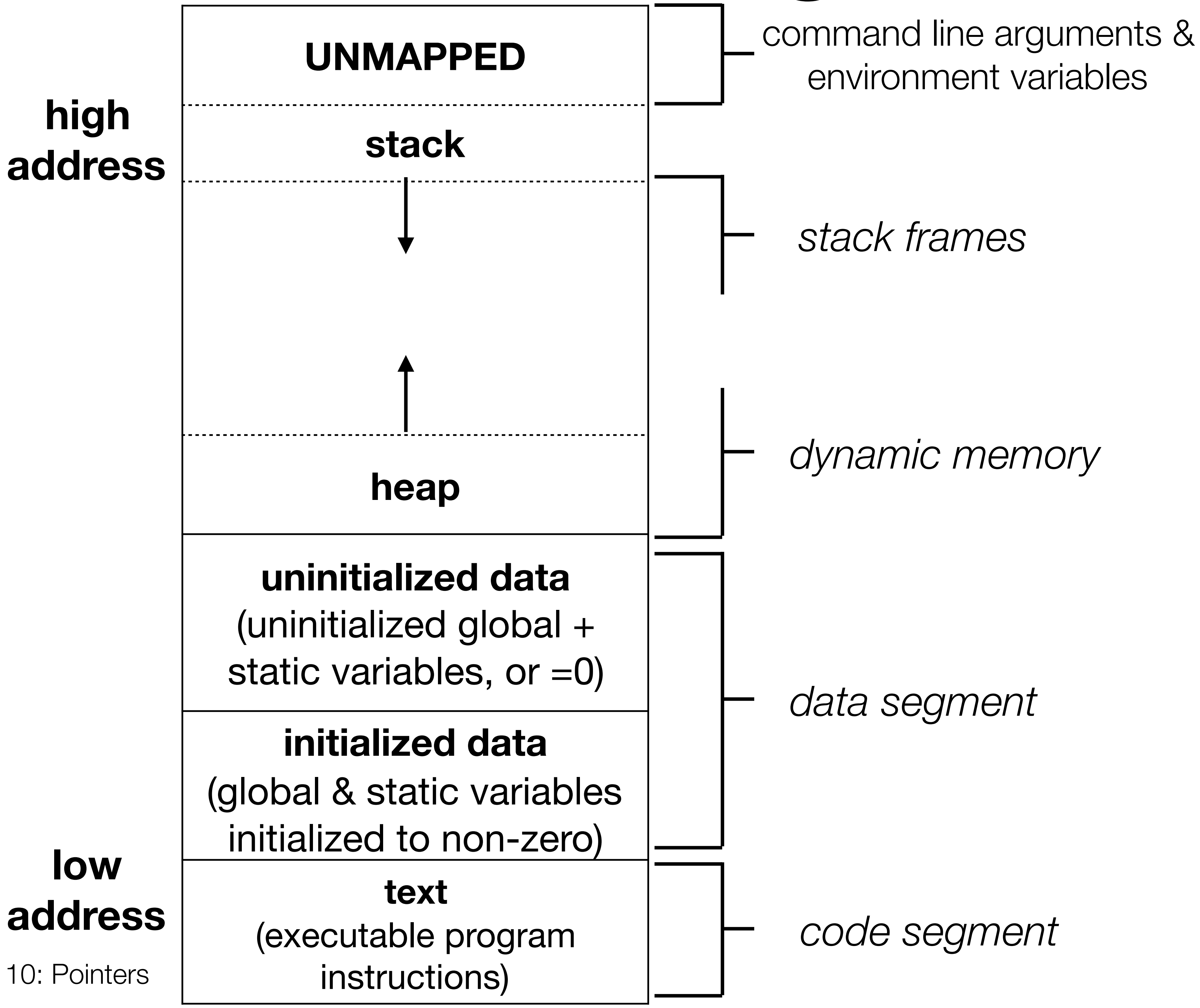
Agenda

- Dynamic storage allocation
- Dynamically allocated strings
- Deallocating storage
- Linked lists
- Pointers to pointers
- Pointers to functions

Readings

- Textbook Chapter 17.1-17.7

Recall: Memory Layout of a C Program


UNMAPPED
high address
stack
heap
uninitialized data
(uninitialized global + static variables, or =0)
initialized data
(global & static variables initialized to non-zero)
low address
text
(executable program instructions)
command line arguments & environment variables
stack frames
dynamic memory
data segment
code segment

Dynamic Storage Allocation

- So far, we have only seen memory that is allocated on the stack. Such memory is automatically allocated and deallocated by the compiler
- On the other hand, dynamic storage allocation allows the programmer to explicitly obtain blocks of memory as needed during execution
- Dynamically allocated memory is allocated on the heap instead of the stack and stays there until it is explicitly freed
- Since the programmer requests the allocation of this memory, the programmer must explicitly free it as well

Advantages of Dynamic Memory Allocation

- The size of certain data structures no longer needs to be fixed when the program is compiled, but can dynamically change during run-time
- Remember that for variable-length arrays in C99, even though the size of the array is determined at run-time, it remains fixed for the rest of the array’s lifetime
- Dynamic memory allocation gives the ability to allocate storage during program execution, allow data structures to grow or shrink in size as needed
- Dynamic memory allocation is used most often for strings, arrays, and structures

Memory Allocation Functions

- Three functions to allocate storage dynamically that are declared in `<stdlib.h>`
  - `malloc`: allocates a block of memory but doesn’t initialize it
  - `calloc`: allocates a block of memory and clears it
  - `realloc`: resizes a previously allocated block of memory
- All three functions return a generic pointer type: `void *`
- `malloc` is the most used and is more efficient that `calloc`, because it does not need to clear the allocated memory

Null Pointers

- If any of the previous memory allocation functions is not able to allocate a block of memory large enough to satisfy the request, it will return a null pointer

- A *null pointer* is a pointer to “nothing”, which is a special value that can be distinguished from all other valid pointers, denoted by the `NULL` keyword

- **It is the programmer’s responsibility to test the return value of any memory allocation function and take appropriate action if it is a null pointer.** The effect of attempting to access memory through a null pointer is undefined: the program may crash or behave unpredictably

Using `malloc` to Allocate Memory for a String

```
void* malloc(size_t size);
```

Using `malloc` to Allocate Memory for a String

```
void* malloc(size_t size);
```

```
char *p = malloc(n+1);
```

Using `malloc` to Allocate Memory for a String

```
void* malloc(size_t size);
```

```
char *p = malloc(n+1);
```

note how we have an extra char to account for the null character

Using `malloc` to Allocate Memory for a String

```
void* malloc(size_t size);
```

```
char *p = malloc(n+1);
```

note how we have an extra char to account for the null character

note how we did not cast the returned pointer by malloc, because the compiler automatically does the cast for us: a pointer of type void* can be assigned to a variable of any pointer type

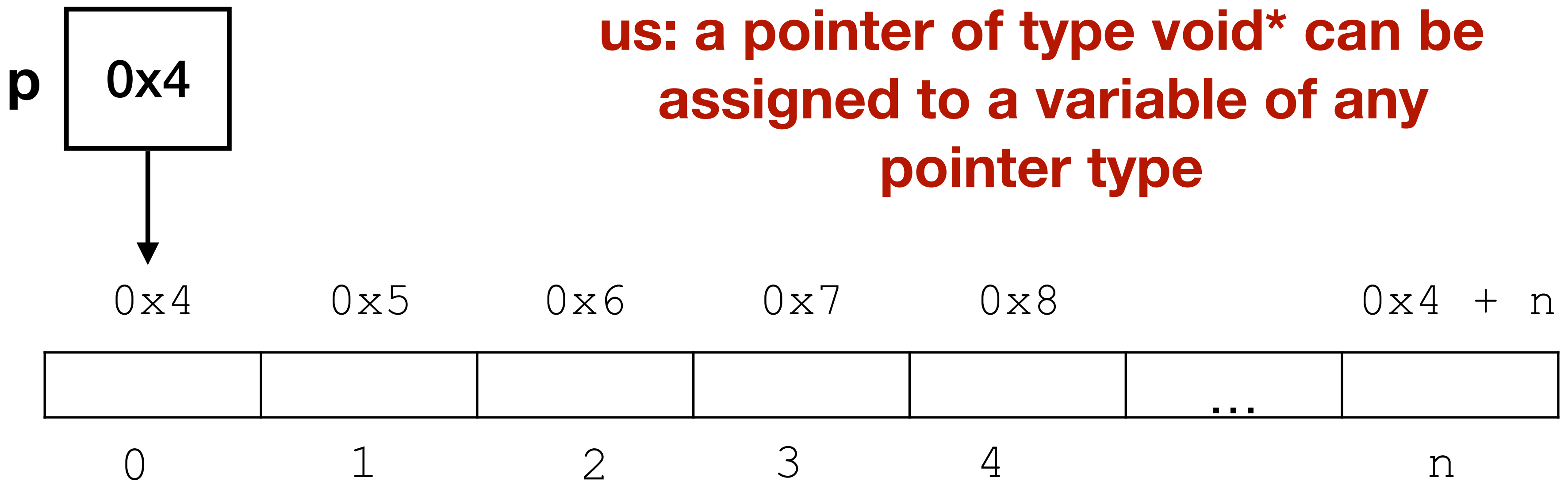
Using `malloc` to Allocate Memory for a String

```
void* malloc(size_t size);
```

```
char *p = malloc(n+1);
```

note how we have an extra char to account for the null character

note how we did not cast the returned pointer by malloc, because the compiler automatically does the cast for us: a pointer of type void* can be assigned to a variable of any pointer type

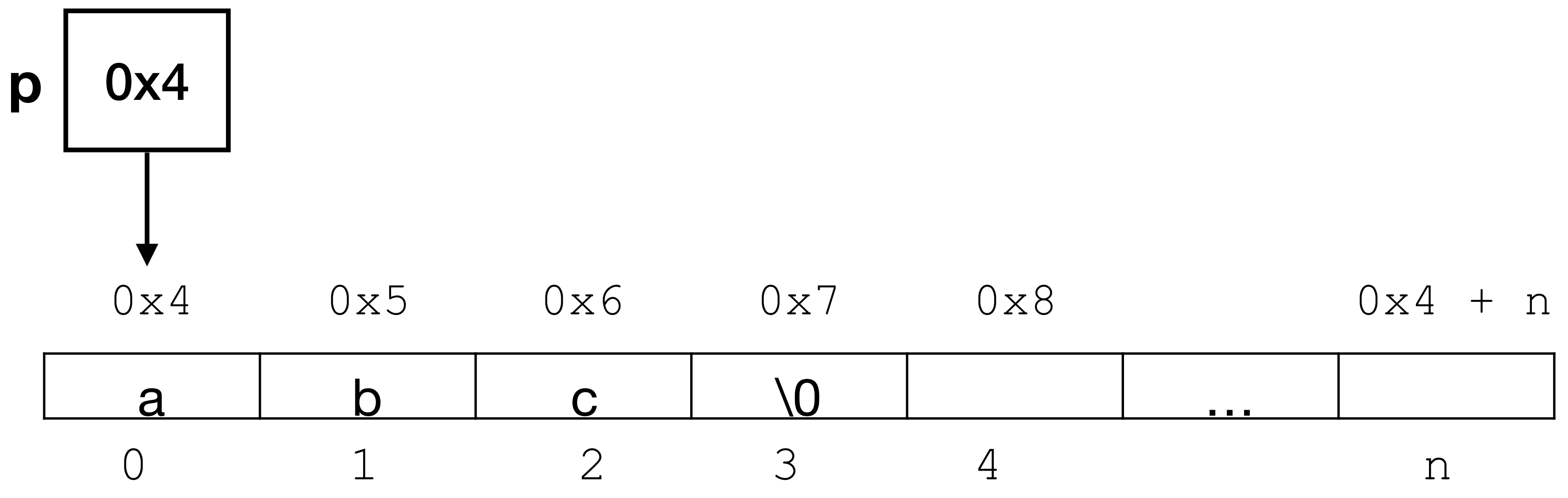


Initializing a String allocated with malloc

```
char *p = malloc(n+1);
strcpy(p, “abc”);
```

Initializing a String allocated with malloc

```
char *p = malloc(n+1);
strcpy(p, “abc”);
```

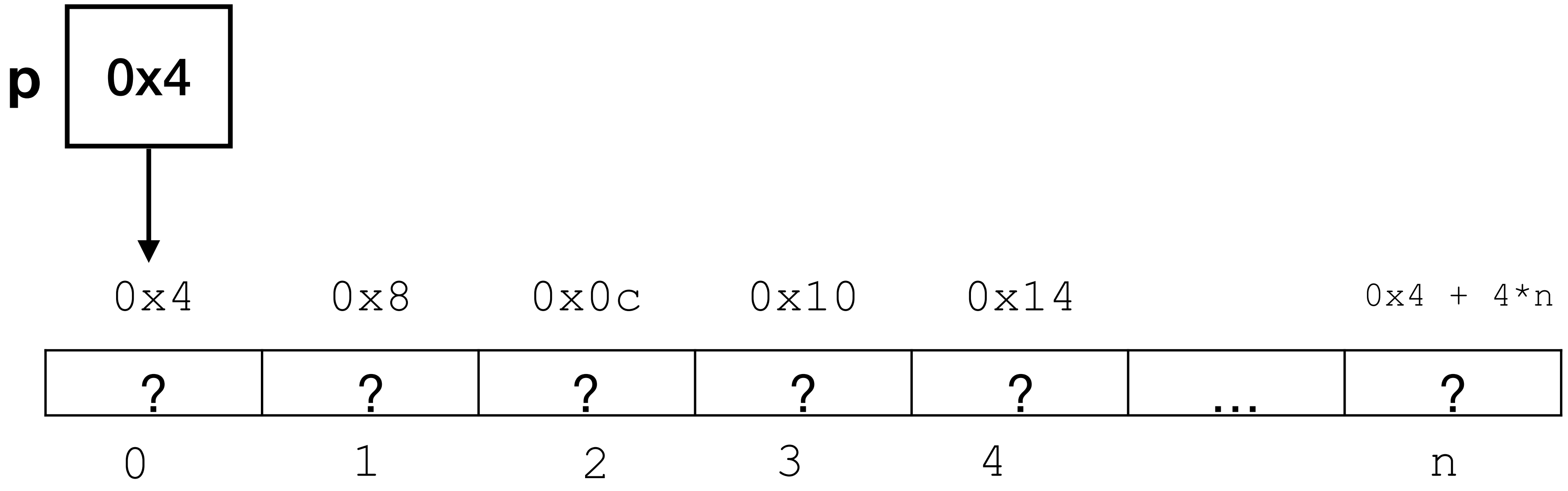


Using `malloc` to Create an `int` array

```
int *p = malloc(n * sizeof(int));
```

Using `malloc` to Create an `int` array

```
int *p = malloc(n * sizeof(int));
```

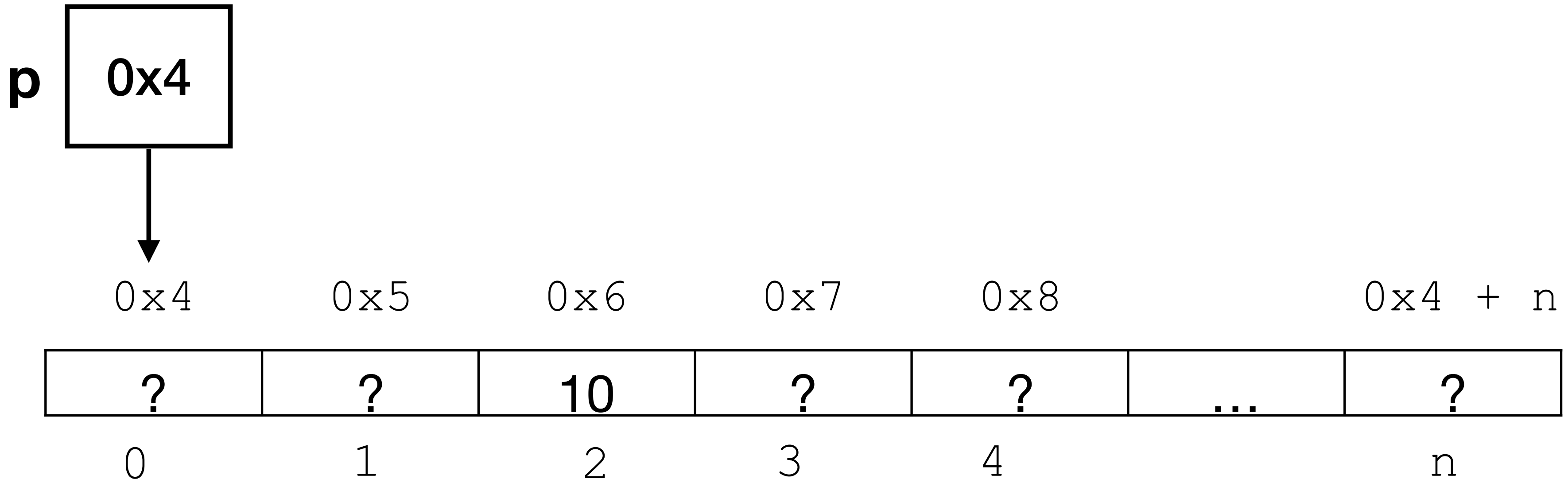


Accessing Elements of `int` Array Allocated with `malloc`

```
int *p = malloc(n * sizeof(int));
p[2] = 10;
```

Accessing Elements of `int` Array Allocated with `malloc`

```
int *p = malloc(n * sizeof(int));
p[2] = 10;
```



Creating a Function that Returns "New" Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Creating a Function that Returns “New” Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

```
char* concat(const char *s1, const char *s2){
 int length = strlen(s1) + strlen(s2) + 1;
 char result[length];
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}
```

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Creating a Function that Returns “New” Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

```
char* concat(const char *s1, const char *s2){
 int length = strlen(s1) + strlen(s2) + 1;
 char result[length];
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}
```

WRONGGGGG!!!!!!!!!

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Creating a Function that Returns “New” Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

```
char* concat(const char *s1, const char *s2){
 int length = strlen(s1) + strlen(s2) + 1;
 char result[length];
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}
```

WRONGGGGG!!!!!!!!!

Remember that result is a local variable with AUTOMATIC STORAGE DURATION. This means that it exists ONLY in the concat function, and the memory allocated for it will be deallocated once execution returns from concat. Therefore, its caller will not be able to correctly access the values in it. Worse, it may result in segmentation faults, because this memory may now be used for something else.

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Using Dynamic Memory Allocation to Return “New” Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Using Dynamic Memory Allocation to Return “New” Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

```
char* concat(const char *s1, const char *s2){
 int length = strlen(s1) + strlen(s2) + 1;
 char *result = malloc(length);
 if (result == NULL){
  fprintf(stderr, “ERROR: malloc failed in concat\n”);
  exit(EXIT_FAILURE);
 }
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}
```

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Using Dynamic Memory Allocation to Return “New” Variables

- Suppose we want to create a function that concatenates two strings without changing either one of them

```
char* concat(const char *s1, const char *s2){
 int length = strlen(s1) + strlen(s2) + 1;
 char *result = malloc(length);
 if (result == NULL){
  fprintf(stderr, “ERROR: malloc failed in concat\n”);
  exit(EXIT_FAILURE);
 }
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}
```

This works because the memory for result is DYNAMICALLY ALLOCATED. Dynamically allocated memory stays on the heap until explicitly freed. Therefore, the caller of concat would still be able to access this memory (and the expected values in it) after execution returns from concat.

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Deallocating Storage

Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

- More so, a program may allocate blocks of memory and then loose track of them

Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

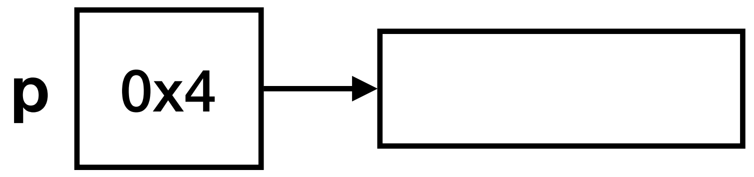
- More so, a program may allocate blocks of memory and then loose track of them

```
p = malloc(…);
q = malloc(…);
p = q;
```

Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

- More so, a program may allocate blocks of memory and then loose track of them

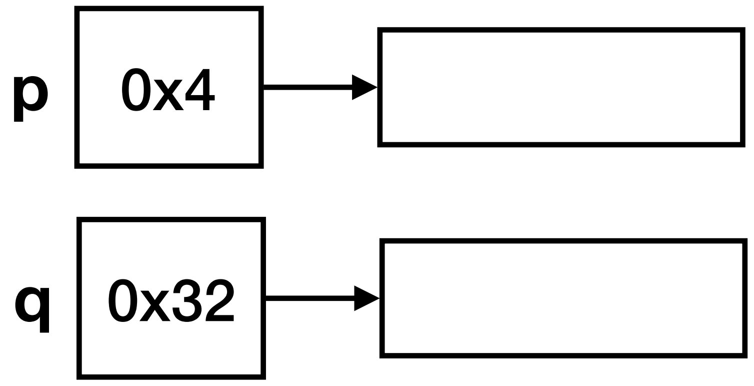


```
p = malloc(…);
q = malloc(…);
p = q;
```

Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

- More so, a program may allocate blocks of memory and then loose track of them



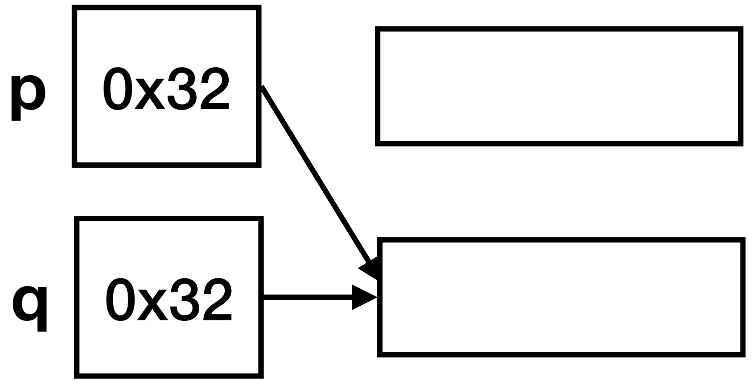
```
p = malloc(…);
q = malloc(…);
p = q;
```

Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

- More so, a program may allocate blocks of memory and then loose track of them

```
p = malloc(…);
q = malloc(…);
p = q;
```

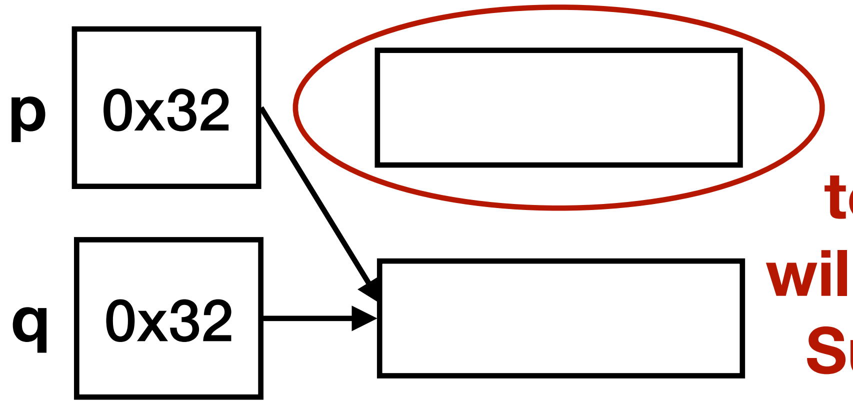


Deallocating Storage

- Dynamically allocated memory is allocated on the heap. If you do not free this memory, and call dynamic allocation functions too often in your code, then you may run out of space

- More so, a program may allocate blocks of memory and then loose track of them

```
p = malloc(…);
q = malloc(…);
p = q;
```



we no longer have access to this block of memory so we will never be able too use it again. Such memory blocks are called *garbage*

Garbage Collection

- A program that leaves garbage behind is said to have a *memory leak*
- Some languages (e.g., Java) provide a *garbage collector* that automatically locates and recycles garbage
- C does not have such a garbage collector. Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory

The `free` Function

```
void free(void *ptr);
```

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

The `free` Function

```
void free(void *ptr);
```

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

The argument to `free` must be a pointer that was previously returned by a memory allocation function. Passing `free` a pointer to any other object (such as a variable or array element) causes undefined behavior.

Reallocating Memory

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

- Once we have allocated memory for an array, we may later find it is too large or too small. The `realloc` function can resize the array as applicable

- When realloc is called, `ptr` MUST point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`. Otherwise, the behavior is undefined

- The `size` parameter represents the size of the new block, which may be larger or smaller than the original size

demo: dynalloc.c

Rules about `realloc` Behavior

- Doesn't initialize the bytes that are added to the block (i.e., doesn't initialize the extra memory it may allocate)
- If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged
- If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`
- If realloc is called with 0 as its second argument, it frees the memory block

Rules about `realloc`

Cont'd

- **Once realloc returns, be sure to update all pointers to the memory block, since it is possible that realloc has moved the block elsewhere**

```
int *temp;
int *myArray = malloc(5 * sizeof(int));
myArray[0] = 10; //etc.
temp = myArray;
temp[3] += 2; // we are using temp to access elements of the array
myArray = realloc(10*sizeof(int));

temp=myArray; /* if we want to use temp again to access the
          first element or process the array, we need to update it
          again bec. realloc may have changed the memory location*/
```

Revisiting the remind Program

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused“
 24 Susan's birthday
 26 Movie - "Chinatown"
```

Revisiting the remind Program *Cont'd*

- Original solution:

	5	S	a	t	u	r	d	a	y		c	l	a	s	s	\0								
1	2	S	a	t	u	r	d	a	y		c	l	a	s	s	\0								
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..

lots of wasted space!

Implementing the remind Program Using Dynamic Memory Allocation

demo: remind2.c

Implementing the remind Program Using Dynamic Memory Allocation

- First we can use a ragged array: `char *reminders[MAX_REMIND]`

demo: remind2.c

Implementing the remind Program Using Dynamic Memory Allocation

- First we can use a ragged array: `char *reminders[MAX_REMIND]`
    - This is a 1D array of character pointers. In other words, each element of the array is a character pointer and this character pointer points to an array of characters (i.e., it is a string). This means we can only use as much characters as needed

demo: remind2.c

Implementing the remind Program Using Dynamic Memory Allocation

- First we can use a ragged array: `char *reminders[MAX_REMIND]`
    - This is a 1D array of character pointers. In other words, each element of the array is a character pointer and this character pointer points to an array of characters (i.e., it is a string). This means we can only use as much characters as needed
- Then, we will dynamically allocate the strings for each element in the array

demo: remind2.c

Implementing the remind Program Using Dynamic Memory Allocation

- First we can use a ragged array: `char *reminders[MAX_REMIND]`
    - This is a 1D array of character pointers. In other words, each element of the array is a character pointer and this character pointer points to an array of characters (i.e., it is a string). This means we can only use as much characters as needed
- Then, we will dynamically allocate the strings for each element in the array
    - When we used a 1D array of char pointers before, we initialized the elements of the array right away, which meant that we needed to know all the strings at the time of initialization and these strings were string literals that cannot be changed later. Using dynamic memory allocation allows us to create string variables after the array initialization

demo: remind2.c

Implementing the remind Program Using Dynamic Memory Allocation

- First we can use a ragged array: `char *reminders[MAX_REMIND]`
  - This is a 1D array of character pointers. In other words, each element of the array is a character pointer and this character pointer points to an array of characters (i.e., it is a string). This means we can only use as much characters as needed
- Then, we will dynamically allocate the strings for each element in the array
  - When we used a 1D array of char pointers before, we initialized the elements of the array right away, which meant that we needed to know all the strings at the time of initialization and these strings were string literals that cannot be changed later. Using dynamic memory allocation allows us to create string variables after the array initialization
  - Additionally, this makes it easier to move existing reminders since we can only change the pointers, rather than use `strcpy`

demo: remind2.c

The `calloc` Function

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

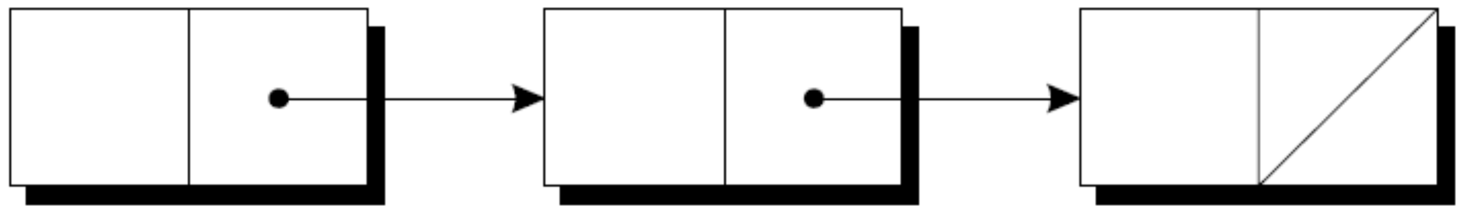
- `calloc` allocates space for an array with `nmemb` elements, each of which is `size` bytes long.
- It returns a null pointer when the requested space is not available
- After allocating the memory, `calloc` initializes this memory by setting all bits to 0
- Note that `malloc` can allocate memory for any variable type, not just arrays, while `calloc` is specifically for arrays. While `malloc` can be used to allocate memory for arrays, remember that it does not initialize that memory for you (and is thus faster)

Linked Lists

- Dynamic storage is especially useful for building lists, trees, graphs, and other linked data structures

- A *linked list* contains a chain of structures (called *nodes),* with each node containing a pointer to the next node

- The pointer in the last node in the list is a null pointer



Properties of Linked Lists

- Linked lists are more flexible than arrays: we can easily insert and delete nodes in a linked list allowing it to grow and shrink as needed

- However, with linked lists, you loose the "random access" capability of an array:

  - any element of an array can be accessed in the same amount of time

  - accessing a node in a linked list depends on where it is in the list (close to the beginning is fast while close to the end is slow)

How to Create a Linked List

- First, you need a structure that represents a single node
- That structure can contain any members you want, but you need to have a pointer to the next node in the list

```
struct node {
  int value;          /* data stored in the node  */
  struct node *next;  /* pointer to the next node */
};
```

- Note that you need to declare the node using the `struct` tag, not a `typedef`. Otherwise, you will not be able to declare the type of `next`.

How to Create a Linked List *Cont'd*

- Now that we have the node structure, we will need a variable that always points to the first node

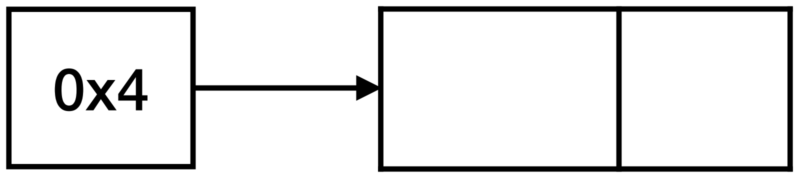
```
struct node *first = NULL;
```

- By setting first to `NULL`, we indicate that the list is initially empty

What does a Pointer to a Struct Mean?

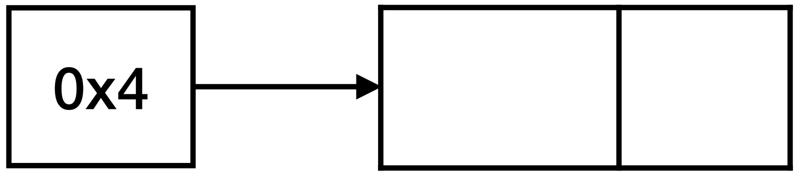
What does a Pointer to a Struct Mean?

```
struct node* test_node = malloc(sizeof(struct node));
```

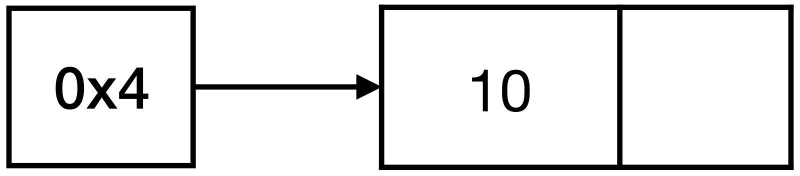


What does a Pointer to a Struct Mean?

```
struct node* test_node = malloc(sizeof(struct node));
```

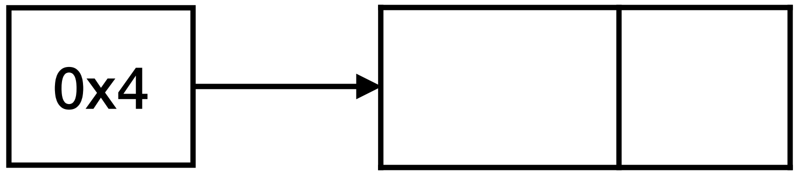


```
(*test_node).value = 10;
```

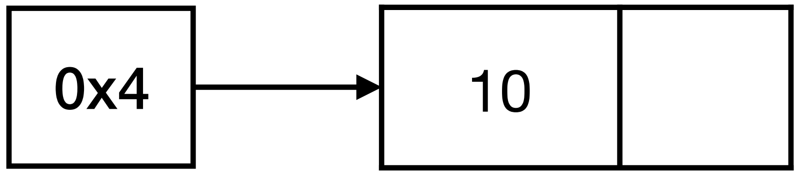


What does a Pointer to a Struct Mean?

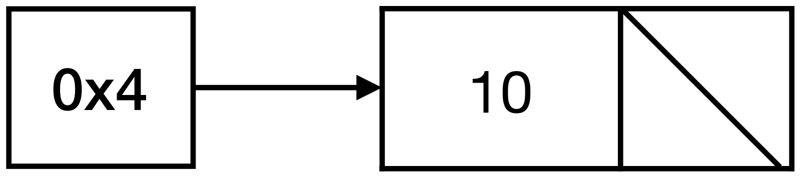
```
struct node* test_node = malloc(sizeof(struct node));
```



```
(*test_node).value = 10;
```



```
(*test_node).next = NULL;
```



The `->` Operator

- It is a bit cumbersome to keep dereferencing the struct pointer and then using the . operator to access its members
- Since accessing a member of a struct using a pointer is so common, C provides a special operator for this
- This operator is known as the *right arrow selection* (I just like to use arrow for short). It is basically a - followed by >

```
(*test_node).value = 10;     <=>     test_node->value = 10;
```

Adding a Node to the List

```
 struct node* add_to_list(struct node* first, int value){
   struct node *newNode;
   newNode = malloc (sizeof(struct node)); //step 1: allocate memory for new node
   if (newNode != NULL){
      newNode->value = 50; //step 2: update value(s) in the new node
      new_node->next = first;//step 3: insert the node at the beg. of the list
   }else{
      fprintf(stderr, "…");
      exit(EXIT_FAILURE);
   }

   return newNode;
 }

int main(){
  struct node *first = NULL;
  int n;
  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n != 0)
        first = add_to_list(first, n);
    else
       break;
  }
}
```

Adding a Node to the List

```
struct node* add_to_list(struct node* first, int value){
  struct node *newNode;
  newNode = malloc (sizeof(struct node)); //step 1: allocate memory for new node
  if (newNode != NULL){
     newNode->value = 50; //step 2: update value(s) in the new node
     new_node->next = first;//step 3: insert the node at the beg. of the list
  }else{
     fprintf(stderr, "…");
     exit(EXIT_FAILURE);
  }

  return newNode;
}

int main(){
  struct node *first = NULL;
  int n;
  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n != 0)
         first = add_to_list(first, n);
    else
       break;
  }
}
```

Note that we return a pointer to the newly created node, which is now at the beginning of the list. We will see in a bit how we can directly update `first`.

Adding a Node to the List

```
struct node* add_to_list(struct node* first, int value){
  struct node *newNode;
  newNode = malloc (sizeof(struct node)); //step 1: allocate memory for new node
  if (newNode != NULL){
      newNode->value = 50; //step 2: update value(s) in the new node
      new_node->next = first;//step 3: insert the node at the beg. of the list
  }else{
      fprintf(stderr, "…");
      exit(EXIT_FAILURE);
  }

  return newNode;
}
```

Note that we return a pointer to the newly created node, which is now at the beginning of the list. We will see in a bit how we can directly update `first`.

```
int main(){
  struct node *first = NULL;
  int n;
  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n != 0)
         first = add_to_list(first, n);
    else
        break;
  }
}
```

Note that numbers will be added in reverse order to the list

Visualization of Adding a Node to the Beginning of a List

(hand-written notes)

How can we search a linked list for a given number `n`?

Searching a Linked List (Option 1)

```
struct node *search_list(struct node *list, int n) {
    struct node *p;
    for (p = list; p != NULL; p = p->next)
      if (p->value == n)
        return p;


    return NULL;
}
```

Searching a Linked List (Option 2)

```
struct node *search_list(struct node *list, int n) {
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;

  return NULL;
}
```

Searching a Linked List (Option 2)

```
struct node *search_list(struct node *list, int n) {
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;

  return NULL;
}
```

Remember that the local variable list is a copy of the original list pointer so there is no harm in modifying it. Modifying it will not change the original list pointer in the caller function (e.g., `main`) in any way.

Searching a Linked List (Option 3)

```
struct node *search_list(struct node *list, int n) {
    for (; list != NULL && list->value != n;
           list = list->next);
    return list;
}
```

Since list is `NULL` if we reach the end of the list, returning `list` is correct even if we don't find `n`

Searching a Linked List (Option 4)

```
struct node *search_list(struct node *list, int n) {
  while (list != NULL && list->value != n)
    list = list->next;

  return list;
}
```

How can we delete a node from a linked list? What are the different cases we need to account for? (hand-written notes)

Deleting a Node from a List

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;
  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next);

   if (cur == NULL)
      return list; /* n was not found */

   if (prev == NULL)
      list = list->next; /* n is in the first node */
   else
     prev->next = cur->next;  /* n is another node in the list */

   free(cur); /* DON'T FORGET TO FREE THE MEMORY!! */

   return list;
}
```

Ordered Lists

- So far, we have seen how to add nodes at the beginning of the list. What if we want to keep an ordered list where the list is sorted by the data in each node
- This means we need to search for the right place to insert the node in

Write a function

```
struct node* add_ordered(struct node* list, int value);
```

that adds nodes into an ordered list

demo: check for memory leaks using valgrind on sorted_list.c

(Command: `valgrind --leak-check=yes myprog arg1 arg2`)

Ref: http://valgrind.org/docs/manual/quick-start.html)

Database Parts Program

- Check out the inventory2.c program in the book

Pointers to Pointers

- Since all function arguments are passed by value, we previously saw that if we want a function to be able to modify the value of a passed argument (e.g., an `int`), then we need to change the function such that it accepts a pointer to that variable (e.g., `int`*). In that case, instead of passing the value of the integer, we would pass the memory address of that integer variable
- Similarly, if we want a function (E.g., `add_to_list`) to be able to modify the value of a passed pointer (e.g., the pointer to the first node in the list), then we need to change it to accept a *pointer to a pointer*
- Remember that we previously saw the concept of a pointer to a pointer (char**) when we covered arrays of `char*`

Modifying `add_to_list` to directly modify the passed list

```
void add_to_list(struct node **list, int n){
    struct node *new_node;
    new_node = malloc(sizeof(struct node));
    if (new_node == NULL) {
       printf("Error: malloc failed in add_to_list\n");
       exit(EXIT_FAILURE);
    }

    new_node->value = n;
    new_node->next = *list;
    *list = new_node;
}
```

Pointers to Functions

- Pointers do not necessarily have to point to data
- C allows pointers to also point to functions
- Functions occupy memory locations, so every function has an address
- We can use function pointers in the same way we use pointers to data
- Passing a function pointer as an argument is fairly common

Simple Example

- Suppose we want to create a function `operate` that performs some specified operation on two operands
- To keep `operate` as general as possible, we can pass a function pointer to the operation we want to do:

```
int operate( int (*f)(int, int), int a, int b);
```

demo: fn-ptr.c

Simple Example

- Suppose we want to create a function `operate` that performs some specified operation on two operands

- To keep `operate` as general as possible, we can pass a function pointer to the operation we want to do:

```
int operate( int (*f)(int, int), int a, int b);
```

parentheses necessary too indicate that `f` is a pointer too a function, not a function that returns a pointer

demo: fn-ptr.c

Simple Example

- Suppose we want to create a function `operate` that performs some specified operation on two operands
- To keep `operate` as general as possible, we can pass a function pointer to the operation we want to do:

```
int operate( int (*f)(int, int), int a, int b);
```

parentheses necessary too indicate that `f` is a pointer too a function, not a function that returns a pointer

the function that is passed must take two `int` parameters and return an `int`

demo: fn-ptr.c

The `qsort` Function

- `qsort` is a useful function in the <stdlib.h> header that makes use of function pointers
- `qsort` is used to sort arrays off any type
- to be able to be that flexible, `qsort` needs to be told what the comparison criteria is. In other words, given two elements `a` and `b`, how does it tell if `a` is less than/equal/greater than `b`?
- The comparison criteria is provided to sort through a *comparison function* that must follow the following guidelines. When given two pointers p and q to array elements, the comparison function must:
  - return an integer that is negative if `*p` is “less than” `*q`
  - return 0 if `*p` is “equal” t `*q`
  - return an integer that is positive if `*p` is “greater than” `*q`

The qsort Function

```
void qsort(void *base, size_t nmemb, size_t size,
               int (*compar)(const void*, const void*));
```

- `base` must point to the first element of the array. If we want to sort the whole array, we simply pass the name of the array. If we want to sort a portion of the array, we pass a pointer to the first element in that portion
- `nmemb` is the number of elements to be sorted (not necessarily equal to the number of elements in the array)
- `size` is the size of each array element, measured in bytes
- `compar` is a pointer to the comparison function that will be internally used for the comparison

demo: sort-array.c