SOLVING PROBLEMS WITH POINTERS AND DYNAMIC MEMORY

Pointer Fundamentals

```
int * intPointer;
int * variable1, variable2;
int *variable1 = &variable2;
*variable1 = 12;
intPointer = variable1;
double * doublePointer = (int*)malloc(sizeof(double));
*doublePointer = 35.4;

double localDouble = *doublePointer;

free(doublePointer);
```

Benefits of Pointers

- Runtime-sized data structures
- Resizable data structures
- Memory sharing

Unnecessary Pointer

```
void compute(int input, int* output);
int num1 = 10;
int* num2 = (int*)malloc(sizeof(int));
compute(num1, num2);

int num1 = 10;
int num2;
compute(num1, &num2);
```

Memory Matters

- All programmers must eventually understand how memory systems work in a modern computer, and C forces you to face this issue head-on.

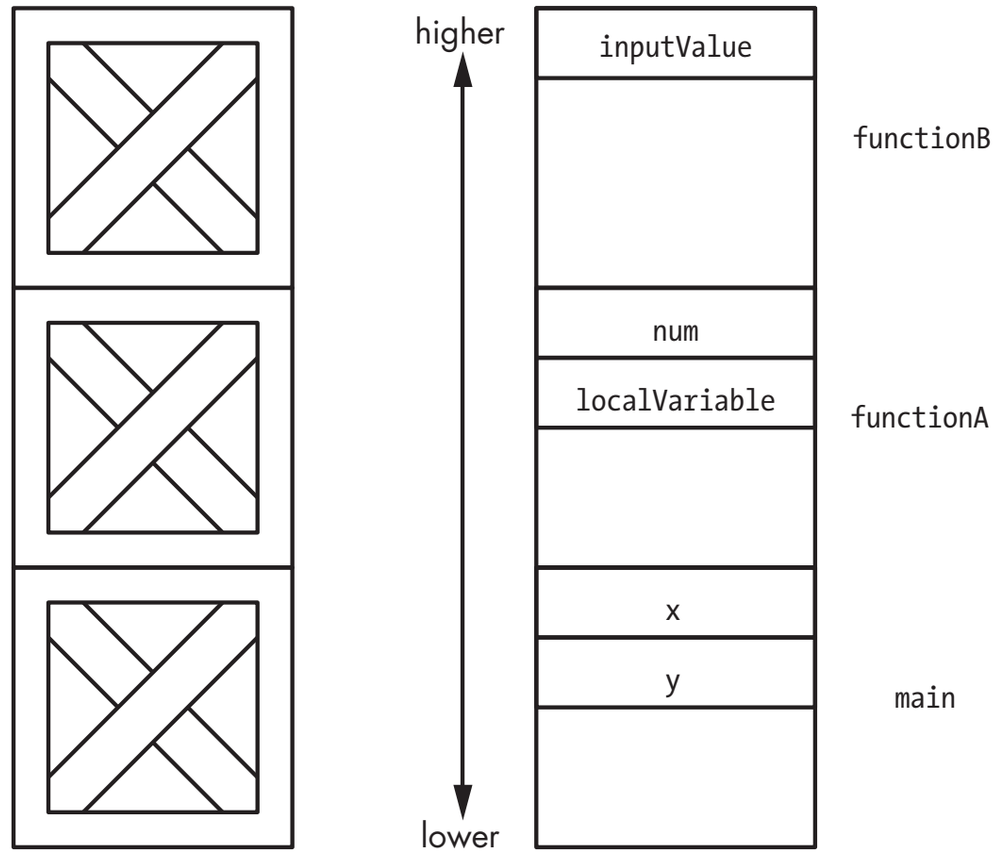
The Stack and the Heap

- C allocates memory in two places: the stack and the heap.

- As the names imply, the stack is organized and neat, and the heap is disjointed and messy.

Stack

- When you have a crate to store, you place it on the top of the stack. To remove a particular crate from the stack, you have to first remove all the crates that are on top of it.



Run-Time Stack

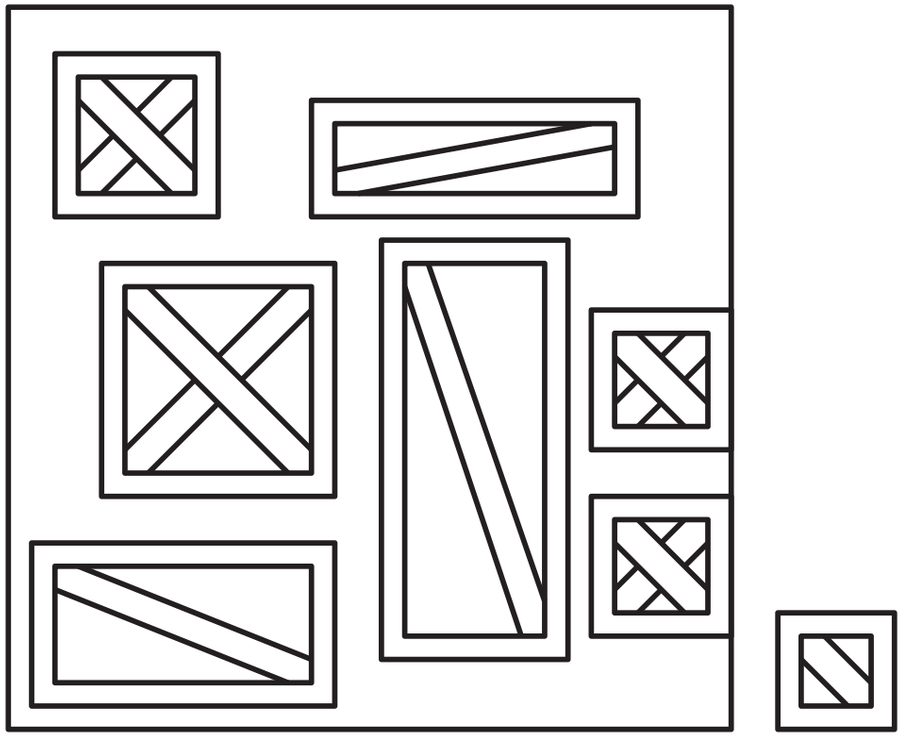
- Every time a function is called (and this includes the main function), a block of memory is allocated on the top of the runtime stack. This block of memory is called an activation record.

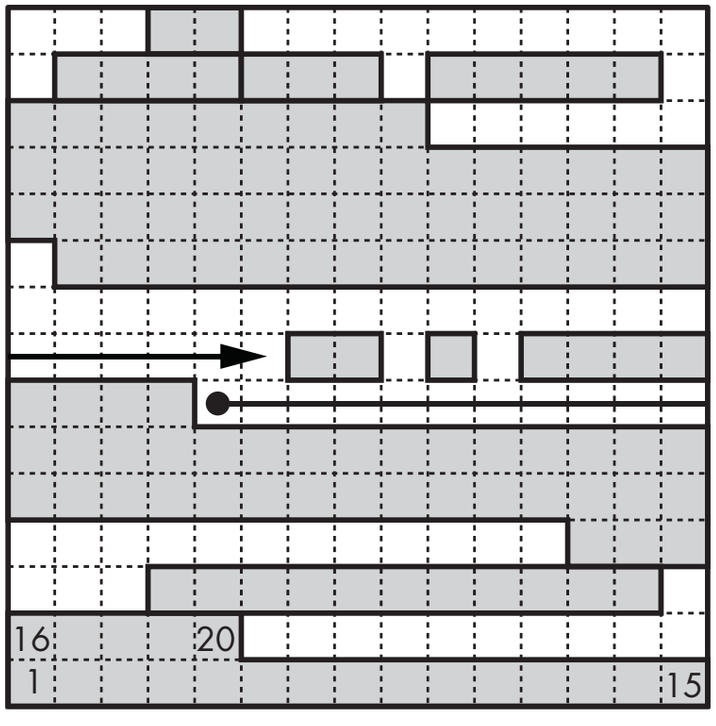
```
int functionB(int inputValue) {
        return inputValue - 10;
}
int functionA(int num) {
    int localVariable = functionB(num * 10);
    return localVariable;
}
int main() {
    int x = 12;
    int y = functionA(x);
    return 0;
}
```

Heap

- this storage system is flexible and allows you to get to the contents of any crate at any time.

- the room is going to quickly become a mess.





Memory Size

- Programs need to use memory efficiently to avoid overall system slowdown. In a multitasking operating system, every byte of memory wasted by one program pushes the system as a whole toward the point where the set of currently running programs doesn't have enough memory to run.

- *How to test your memory size in stack and heap?*

Lifetime

- The lifetime of a variable is the time span between allocation and deallocation.

- With a stack-based variable, meaning either a local variable or a parameter, the lifetime is handled implicitly.

- With a heap-based variable, meaning a variable dynamically allocated using *malloc*, the lifetime is in our hands.

Memory Leak

- The most obvious issue is the dreaded memory leak, a situation in which memory is allocated from the heap but never deallocated and not referenced by any pointer.

```
int *intPtr = (int*)malloc(sizeof(int));
intPtr = NULL;
```

deallocate the same memory twice

- Sometimes, instead of memory that never gets deallocated, we have the opposite problem, attempting to deallocate the same memory twice, which produces a runtime error. This might seem like an easy problem to avoid: Just don't call *free* twice on the same variable. What makes this situation tricky is that we may have multiple variables pointing to the same memory. If multiple variables point to the same memory and we call *free* on any of those variables, we have effectively deallocated the memory for all of the variables. If we don't explicitly clear the variables to NULL, they will be known as dangling references, and calling *free* on any of them will produce a runtime error.

Variable-Length Strings

- Write heap-based implementations for three required string functions:
  - *append* This function takes a string and a character and appends the character to the end of the string.
  - *concatenate* This function takes two strings and appends the characters of the second string onto the first.
  - *characterAt* This function takes a string and a number and returns the character at that position in the string (with the first character in the string numbered zero).
- Write the code with the assumption that characterAt will be called frequently, while the other two functions will be called relatively seldom. The relative efficiency of the operations should reflect the calling frequency.

Bad Style

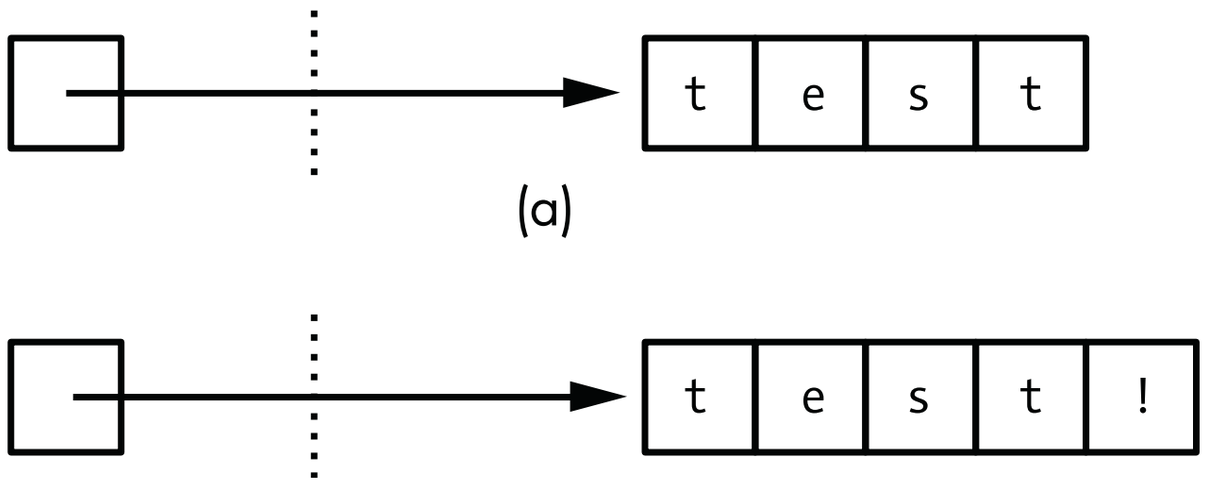
```
typedef char * arrayString;

char characterAt(arrayString s, int position) {
    return s[position];
}
```

characterAt

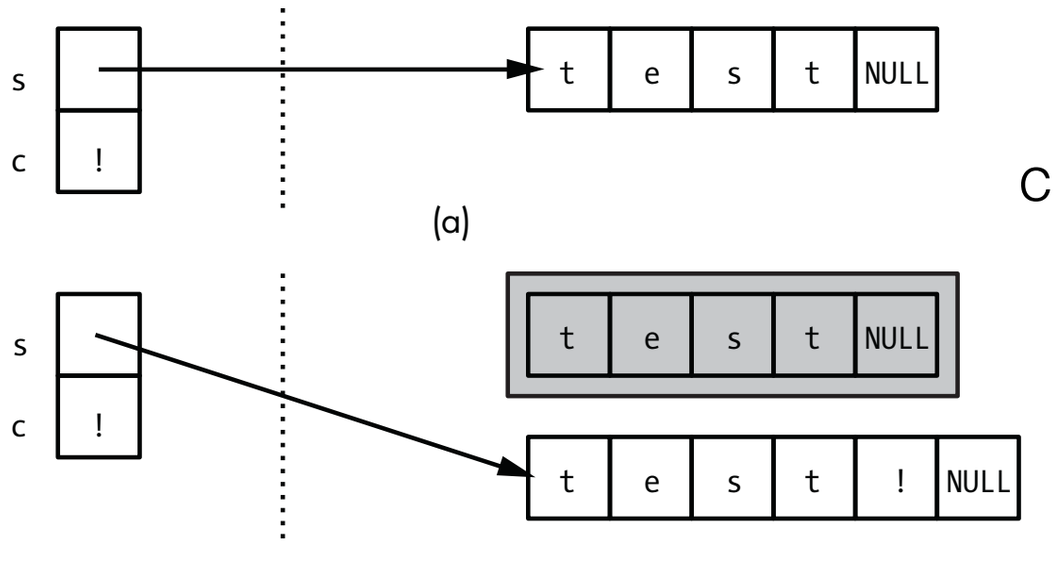
```
char characterAt(char* s, int position) {
        return s[position];
}
```

append



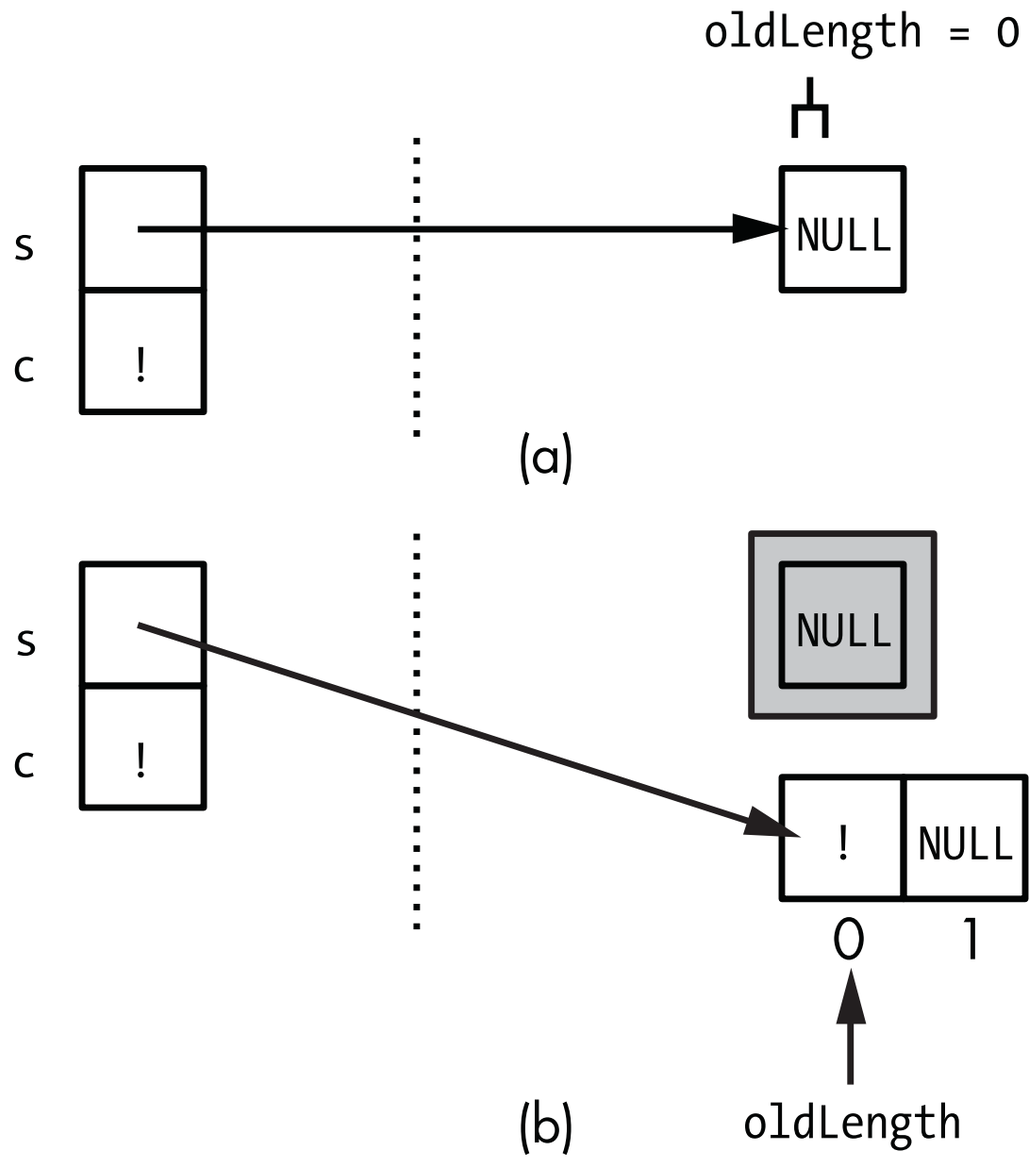
- How do we specify the length/end of the string?

append




```
char* append(char *s, char c) {
    int oldLength = 0;
      while (s[oldLength] != 0) {
            oldLength++;
      }
      char* newS = (char*)malloc(oldLength + 2);
      for (int i = 0; i < oldLength; i++) {
            newS[i] = s[i];
      }
      newS[oldLength] = c;
      newS[oldLength + 1] = 0;
      free(s);
      return newS;
}
```

Checking for Special Cases



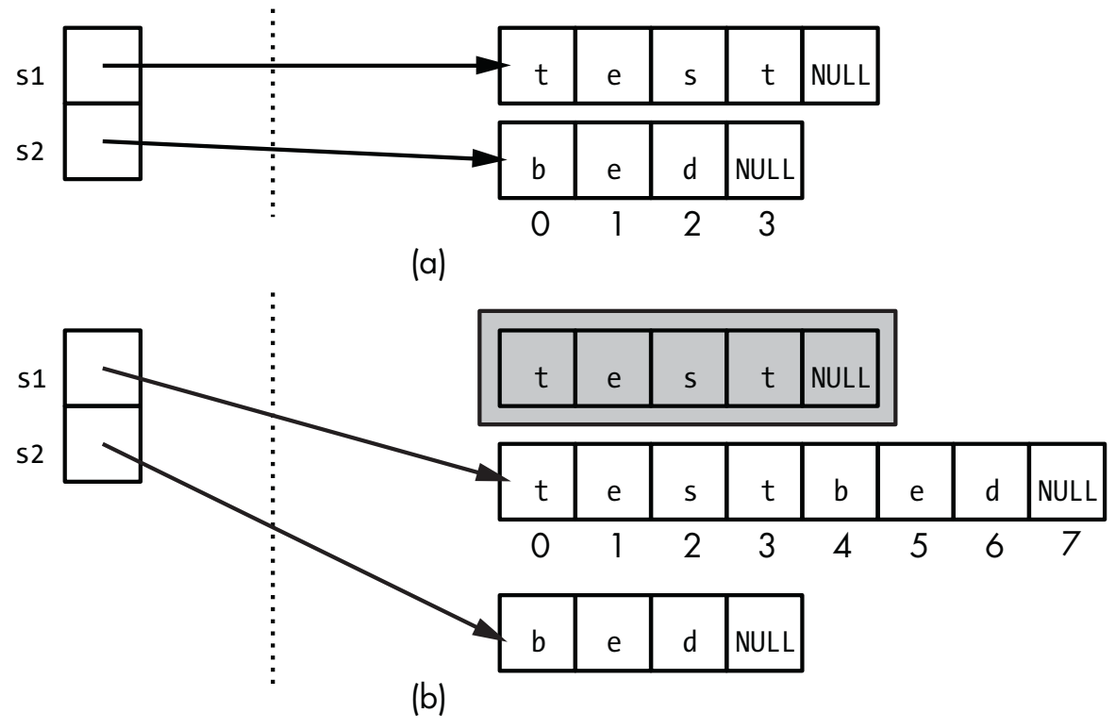
Optimization for Future

```
int length(char* s) {
        int count = 0;
        while (s[count] != 0) {
            count++;
        }
    return count;
}
```

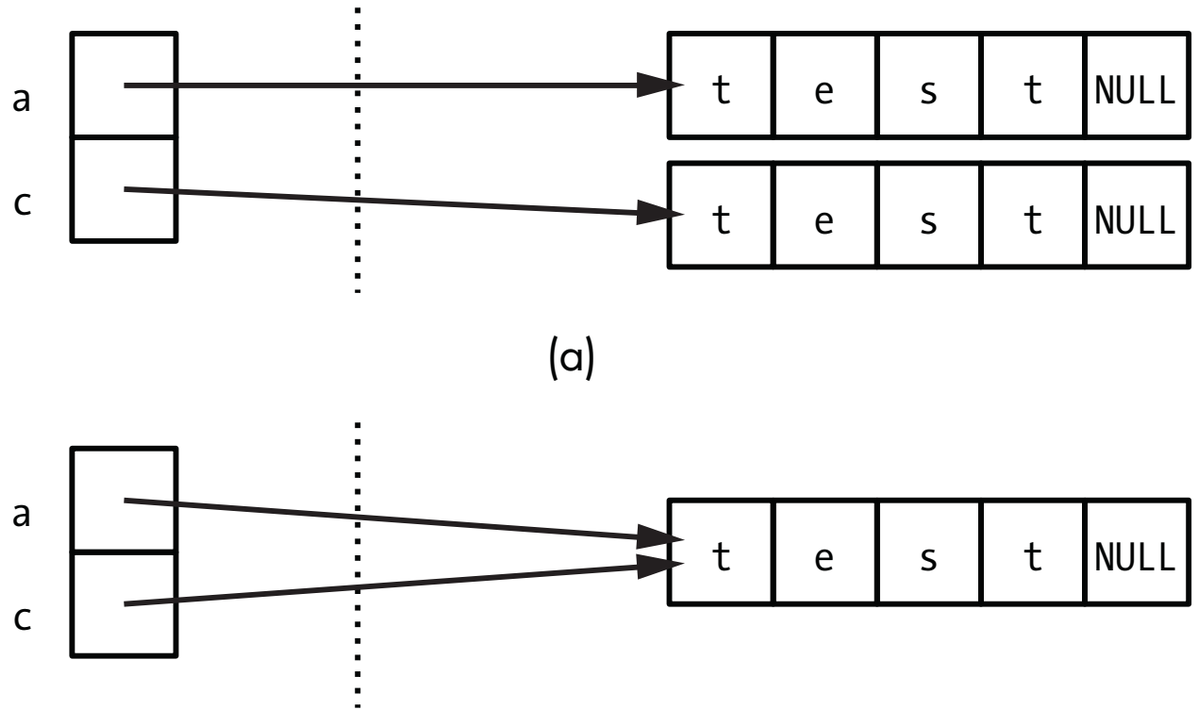
```
char* append(char *s, char c) {
    int oldLength = length(s);
    char* newS = (char*)malloc(oldLength + 2);
    for (int i = 0; i < oldLength; i++) {
        newS[i] = s[i];
    }
    newS[oldLength] = c;
    newS[oldLength + 1] = 0;
    free(s);
    return newS;
}
```

concatenate

```
char* concatenate(char* s1, char* s2) {
    int s1_OldLength = length(s1);
    int s2_Length = length(s2);
    int s1_NewLength = s1_OldLength + s2_Length;
    arrayString newS = (char*)malloc(s1_NewLength + 1);
    for(int i = 0; i < s1_OldLength; i++) {
        newS[i] = s1[i];
    }
    for(int i = 0; i < s2_Length; i++) {
        newS[s1_OldLength + i] = s2[i];
    }
    newS[s1_NewLength] = 0;
    free(s1);
    return newS;
}
```



Special Cases



Pascal String