APS105 Lecture 28
Dynamic Memory Allocation

Dan Zingaro

March 31, 2010

Feedback from Reading Quiz

- Assume that `struct student` has been declared. Write one line of code that sets variable `s` to a block of memory that can be used to hold a `struct student`.
  - (1) `struct student *s= (struct student*) malloc(sizeof(struct student*))`
  - (2) `struct student *s =malloc(sizeof(struct));`
  - (3) `struct student *s= malloc(size of struct)`
  - (4) `struct student *s = malloc (s * sizeof(int))`
- "If we do not use `free` right after using `malloc` then will the computer show an error?"
  - No, but if you never `free` memory, you'll eventually run out

Automatic Memory Allocation

- Imagine we want to write a function `concat` that takes two strings, and returns their concatenation as a new string
  - It will **not** modify one of the existing strings, so it behaves differently from `strcat`
- We could call it like this

```
char *s;
s = concat ("abc", "def");
printf ("%s\n", s); /*abcdef*/
```

Automatic Memory Allocation...

Here is an attempt at writing this function.

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

Automatic Memory Allocation...

- The problem is that the `result` array goes out of scope when the function returns
- The memory for `result` exists only while `concat` is running, after which we lose access to that memory
- What we have to be able to do is allocate memory that will not be lost when `concat` finishes
- In other words, we require manual control of memory allocation

Dynamic Memory Allocation

- We want a function `concat` that takes two strings and concatenates them into a new string
- On each call of `concat`, we must allocate memory for the new string
- When the program is being compiled, we can't possibly know how many times `concat` will be called
- We do not know how much memory to reserve at compile-time
- We must dynamically allocate memory while the program is running

Malloc

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

- We use `malloc` to dynamically allocate memory
- `malloc` returns a pointer to the newly acquired memory, or `NULL` if there is not enough available memory
- To allocate memory for a string of `n` characters, we ask `malloc` for one extra character to store the `'\0'`

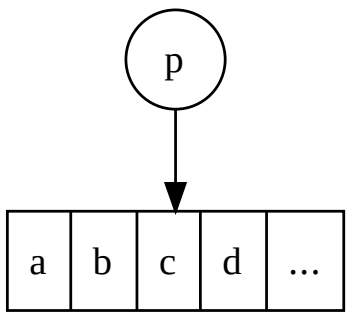
```
char *p;
p = malloc (n + 1);
```

Malloc...

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

- `p` is now a pointer to an uninitialized `n+1`-byte block of memory
- We can use it like any character array (i.e. string)

```
strcpy (p, "abcd");
```



A Correct Concat

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

char *concat(const char *s1, const char *s2) {
  char *result;
  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

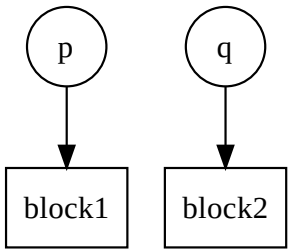
Freeing Memory

- Once we use `malloc` to ask for memory, we're responsible for releasing it when it's no longer required
  - When we don't use `malloc`, memory is freed automatically when it goes out of scope
- So, when we're finished with the concatenated string, we should release its memory
- If we keep calling `malloc` without freeing any of the memory, we'll eventually run out

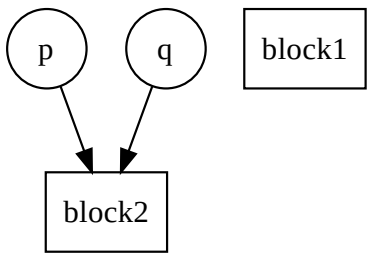
```
free (p);
```

Lost Memory

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




```
p = q;
```



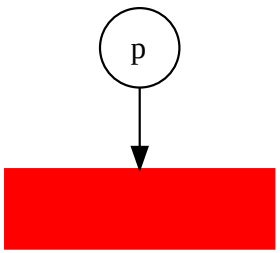
- Since p now points to q's block, we have no way of accessing p's old block
- This is called a memory leak

Dangling Pointers

```
char *p = malloc(5);
...
free(p);
```

- p is now a pointer to memory it does not own; it is an error to access memory through p

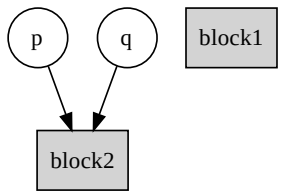
```
strcpy (p, "abcd"); /*Bad!*/
```

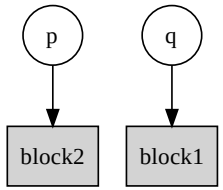


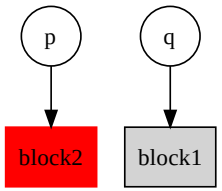
ConcepTest

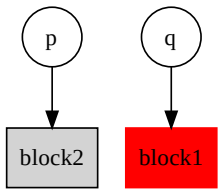
What is the result of the following code? (A red box indicates a block of memory that has been freed.)

```
char *p = malloc(5);
char *q = p;
p = malloc(5);
```

- A. 

- B. 

- C. 

- D. 

sizeof

- The size of many types of data varies depending on the computer being used
- C includes the `sizeof` operator that tells us how many bytes a type or an expression requires

```
#include <stdio.h>

int main(void) {
  int i;
  double dbl;
  printf ("Size of i: %d.\n", sizeof (i));
  printf ("Size of dbl: %d.\n", sizeof (dbl));
  printf ("Size of  an int: %d.\n", sizeof (int));
  return 0;
}
```

ConcepTest

```
struct student {
  char firstName[20];
  char lastName[20];
  int year;
};
```

What is the best way to dynamically allocate a `struct student`?

- A. `struct student *s = malloc(struct student);`
- B. `struct student *s = malloc(sizeof(struct student));`
- C. `struct student *s = malloc(44);`