Dynamic memory in C

- Scope of names in C
- Dynamic memory
  - malloc and calloc
  - free
- C programming idioms

Scope of Names

- The scope of a variable determines the region over which you can access the variable by its name.

- C provides four types of scope:
  - **Program** scope
  - **File** scope
  - **Function** scope
  - **Block** scope

Program Scope

- The variable exists for the program's lifetime and can be accessed from any file comprising the program.
  - To define a global variable, omit the "extern" keyword, and include an initializer (needed if you want a value other than 0).
  - To link to a global variable, include the extern keyword but omit an initializer
- Example:
  - Variable with program scope is declared and referenced in file 1.
  - Variable with program scope is referenced in file 2.

```
/*
 * file 1
 */

int ticks = 1;

void tick_tock() {
    ticks += 1;
}
```

```
/*
 * file 2
 */

extern int ticks;

int read_clock() {
    return ticks * TICKS_PER_SECOND;
}
```

File Scope

- The variable is visible from its point of declaration to the end of the source file.
- To give a variable file scope, define it outside a function with the **static** keyword

```
/*
 * file 3
 */

static long long int boot_time = 0;

void at_boot(void) {
    boot_time = get_clock();
}

...

int main(void) {
    printf("%i\n", boot_time);
}
```

```
/*
 * file 4
 */

/* THE LINE BELOW WILL FAIL
 * when the executable is constructed.
*/

extern long long int boot_time = 0;
```

Function Scope

- The name is visible from the beginning to the end of a function.
- According to the ANSI standard, the scope of function arguments is the same as the scope of variables defined at the outmost scope of a function. Shadowing of function arguments is not allowed.
- (Shadowing of global variables is permitted, however.)

```
/*
 * file 5
 */

void function_f(int x)
{
    ... = x + ...;
}
```

```
/*
 * file 6
 */

/* The variable declaration within the
 * function below will cause a compiler
 * error.
 */
void function_g (int x)
{
   int x;  /* Not possible. */
}
```

```
/*
 * file 7
 */
int sum = 0;

void function_h(int x)
{
    int sum = init_sum(); /* different! */
}
```

Block Scope

- The variable is visible from its point of declaration to the end of the block. A block is any series of statements enclosed by braces.

```
/*
 * file 7
 */

int sum;

void function_y (int X[], int n) {
    int j;

    {
        /* Start of a nested scope */

        int j;
        for (j = 0, sum = 0; j < n; j += 1) {
            sum += X[j];
        }

        /* End of a nested scope */
    }
}
```

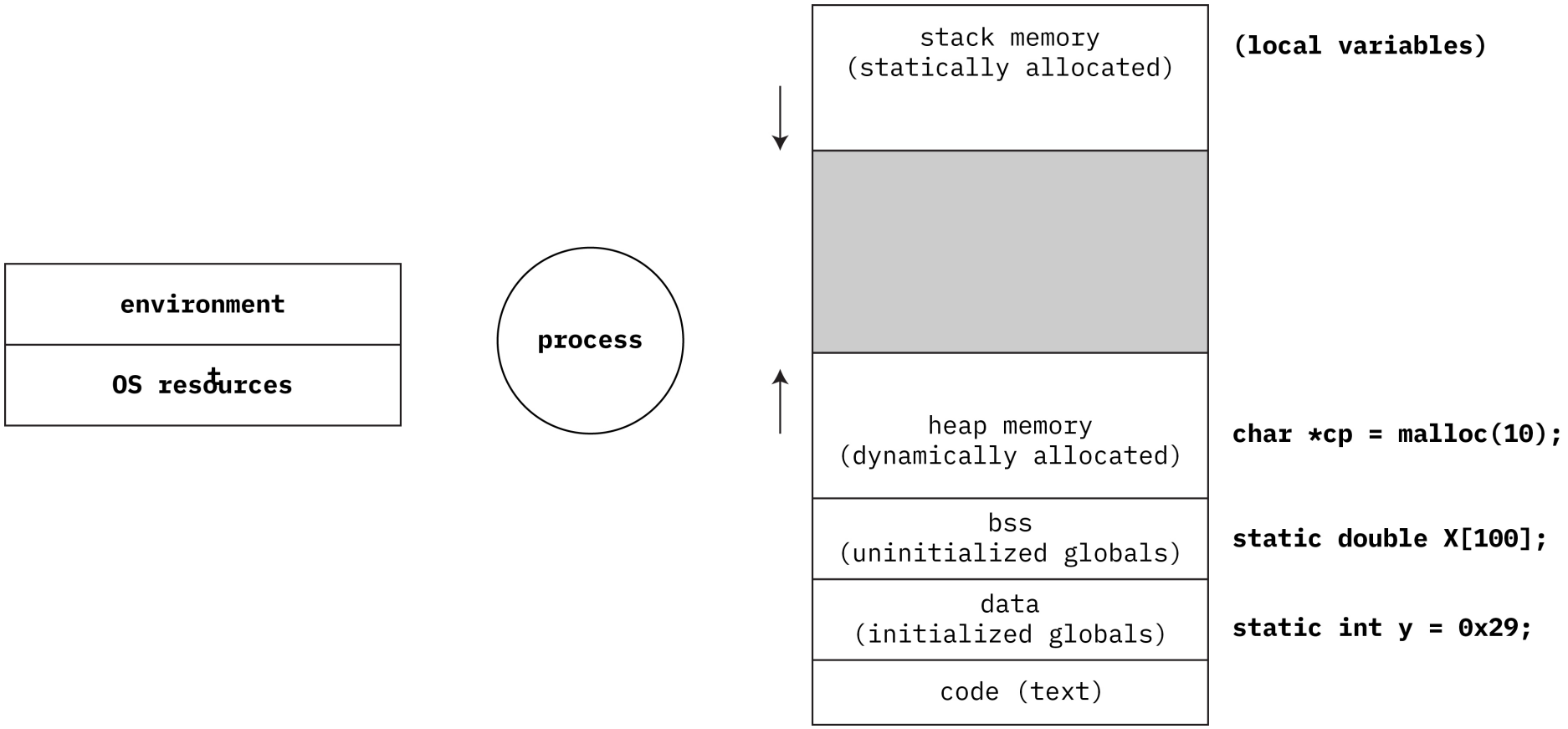
Dynamic memory in C

- The C memory model
- Managing the heap: `malloc()` and `free()`
- The void pointer and casting
- Memory leaks

C memory model

- Memory is divided into five segments: **uninitialized data**, **initialized data**, **heap**, **stack**, and **code**
- **Stack** is used to store **automatics** and activation records (stack frames) for functions
- Stack is located at the "top" of writable memory (high addresses)
- **Heap** stores explicitly requested memory which must be **dynamically allocated**
- Limits of heap and stack can bump into each other
- Initialized and uninitialized data located at at "top" of writable memory (high addresses)

C memory model



C memory model

- As the program executes, and function call-depth increases, the stack **grows downward**
- Similarly, as memory is explicitly requested for allocation, the heap **grows upward**
- Every time a function call is made, a new **stack frame** is created and memory allocated for variables to be used by the function
- Eventually, if stack and heap continue to grow, all memory available to the process will be exhausted and an **out of memory** condition will occur

Motivation for dynamic memory

- Memory must be allocated for storage before variables can be used, yet the amount might not be known at compile time
- Examples:
  - Reading records from a file in order to sort them, where file size is not known at time program is written
  - Constructing a list of "keywords" based on the content of some text file (whose size is unknown at compile time)
  - Representing a set as a linked list

Motivation for dynamic memory

- One solution: Write the program by hard-coding in the largest amount of memory that could possible be needed
- Problem with this approach: Possibly occupies a large, unused area of memory if program-input sizes for most executions are almost always small
  - Note: virtual-memory systems mitigate the effects of this somewhat

Where to store what…

- Use **heap memory** for dynamic allocation when size is not known until run-time
- Use **stack memory** for parts where size is known at compile time
- Working with the stack is easy -- all variables are defined at compile time (no extra work for you)
- Working with the heap is a bit harder
  - In Java and Python, heap memory is automatically allocated to objects through use of the "new" keyword
  - Also in Java & Python, heap memory no longer used by a variable may be reclaimed for the system (**garbage collection**)

Heap memory in C

- Memory addresses in the heap are sometimes called **anonymous variables**
  - These variables do not have names like automatics and static variables
- malloc(): allocate dynamic memory
  - Takes a single parameter representing the **number of bytes of heap memory to be allocated**
  - **Returns a memory address** to the beginning of newly allocated block of memory
  - **If allocation fails**, malloc() returns NULL
- #include <stdlib.h> : contains function prototypes for malloc and related functions.

sizeof

- sizeof() is an operator that computes the number of bytes allocated for a specified type or variable (basic types, aggregate types)
- Use sizeof() to determine block size required by malloc()
- You must **always** check the value returned by malloc()!

```
int *a = malloc (sizeof(int));
if (a == NULL) { /* error */ }

struct datetype *dt = malloc (sizeof(struct datetype));
if (dt == NULL) { /* error */ }

char *buffer = malloc (sizeof(char) * 100);
if (buffer == NULL) { /* error */ }
```

malloc() + casting

- Function prototype for malloc() :
  - extern void *malloc(size_t n);
  - typedef unsigned int size_t;
- The pointer returned is a generic pointer
- To use the allocated memory, we must **typecast** the returned pointer
  - Denoted by (<sometype> *)
  - Casting is a hint to the compiler (applies different typechecking to the block of memory after the typecast)

```
double *f = (double *) malloc (sizeof(double));

char *buf = (char *) malloc(100);
```

Casting

- What if we do not typecast?

```
double *f = malloc(sizeof(double));
...
<from some compilers>
 warning: assignment makes pointer from
                double without a cast
```

- Always a good idea to cast
- Once heap memory is allocated:
  - It stays allocated for the duration of the program's execution…
  - ... unless it is **explicitly deallocated**
  - ... and this is true regardless of what functions calls malloc()
  - All memory used in heap is automatically returned to system when process/program terminates.

A family of functions

- There is more than just malloc:
  - calloc
  - realloc
  - valloc
- These serve slightly different purposes
  - One function both allocates and initializes the block of heap memory
  - One function adjusts heap structures to change size of a previously allocated block/chunk
  - One is now pretty much obsolete (i.e., meant to force allocation on some page boundary – which will make more sense in CSC 360)
- As we need these extra functions we'll trot them out

free()

- We use free() to return heap memory no longer needed back to the heap pool

  extern void free(void *);

- free() takes a pointer to the allocated block of memory

```
void very_polite_function( int n )
{
   int *array = (int *) malloc (sizeof(int) * n);

   /* Code using the array */

   free (array);
}
```

Issues with dynamic memory

- A **memory leak** occurs when heap memory is **constantly allocated** but is **not freed** when no longer needed
- Memory leaks are almost always unintentional
  - Allocation and deallocation code locations are often widely separated.
  - Can be hard to find the memory-leak bug as it often depends upon the program running for a long time.
- Systems with automatic garbage collection (almost) never have memory leaks
  - Redundant memory is returned to heap for re-use
  - Downside: garbage collection is not always under control of the programmer
  - Also: some garbage collectors cannot reclaim some kinds of redundant instances of data types.

C string programming idioms

- "programming idiom"
  - "means of expressing a recurring construct in one or more programming languages"
  - use of idioms indicates language fluency
  - also assumes some comfort with the language
- idioms also imply terseness
  - expressions using idioms tend to be the "ideal" size
  - "terseness" can even have an impact as machine-code level
- non-string example: infinite loop

A C-language idiom: infinite loop

```
/*
 * Not the ideal technique
 */

while (1) {
    some_function();
    if (someflag == 0) {
        break;
    }
    some_other_function();
}
```

Loop must always perform a check at the start of the loop.

```
/*
 * Recommended approach ("idiomatic").
 */

for (;;) {
    some_function();
    if (someflag == 0) {
        break;
    }
    some_other_function();
}
```

There is no check at the start of the loop -- no extra instructions!

Example: Computing string length

- Note!
  - Normally we use built-in library functions wherever possible.
  - There is a built-in string-length function ("strlen").
  - These libraries functions are very efficient and very fast (and bug free)
- Algorithm:
  - Function accepts pointer to a character array as a parameter
  - Some loop examines characters in the array
  - Loop terminates when the null character is encountered
  - Number of character examined becomes the string length

First example

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

#define MAX_STRING_LEN 100

int stringlength_1(char a_string[])
{
        int len = 0;

        while (a_string[len] != '\0') {
                len = len + 1;
        }
        return len;
}

int main(int argc, char *argv[])
{
        char buffer[MAX_STRING_LEN];

        if (argc == 1) { exit(0); }
        strncpy(buffer, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_1(buffer));

         exit(0);
}
```

C knows nothing about the bounds of arrays!

"char a_string[]" is the same as "char *a_string"

Each character is explicity compared against the null character. Note the single quotes!

Name of character array is passed as the parameter to stringlength_1.

"buffer" is the same as "&buffer[0]"

First example: not idiomatic

- C strings are usually manipulated via indexed loops
  - "for" statements
- "For" statements are suitable to use with loops:
  - where termination depends the size of some array
  - where termination depends upon the size of some linear structure
  - where loop tests are at loop-top and loop-variable update operations occur at the loop-end
- "While" statements are most suitable with loops:
  - where termination depends on the change of some state (usually with the sequence over which the loop is traversing) during one or many loop iterations
  - where termination depends on some property of a complex data structure
  - where actual loop operations can possibly lengthen or shorten number of loop iterations (e.g., "worklist" algorithms)

Second example

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

#define MAX_STRING_LEN 100

int stringlength_2(char a_string[])
{
        int len;

        for (len = 0; a_string[len] != '\0'; len = len + 1) { }

        return len;
}

int main(int argc, char *argv[])
{
        char buffer[MAX_STRING_LEN];

        if (argc == 1) { exit(0); }
        strncpy(buffer, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_2(buffer));

         exit(0);
}
```

Each character is explicity compared against the null character, but this is done within the "for" header.

"For" loop itself is empty.

Second example: not idiomatic

- C strings are most often accessed via char pointers
- Accessing individual characters by array index is rare
  - Principle is that strings are usually processed in one direction or another
  - That direction proceeds char by char
- More idiomatic usage also depends upon pointer arithmetic

Third example

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

#define MAX_STRING_LEN 100

int stringlength_3(char a_string[])
{
        char *c;
        int len = 0;

        for (c = a_string; *c != '\0'; c = c + 1) {
                len = len + 1;
        }
        return len;
}

int main(int argc, char *argv[])
{
        char buffer[MAX_STRING_LEN];

        if (argc == 1) { exit(0); }
        strncpy(buffer, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_3(buffer));

         exit(0);
}
```

Note that a character pointer is used (i.e., dereferenced in the control expression, and incremented in the post-loop expression).

The body of the "for" loop is not empty here as variable "len" is increment in it.

(Note: We could add "len" to our "for"-loop header and keep the body empty. What would that look like?

Fourth example

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

#define MAX_STRING_LEN 100

int stringlength_4(char a_string[])
{
        char *c;
        int len;

        for (len = 0, c = a_string; *c; c++, len++) { }

        return len;
}

int main(int argc, char *argv[])
{
        char buffer[MAX_STRING_LEN];

        if (argc == 1) { exit(0); }
        strncpy(buffer, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_4(buffer));

         exit(0);
}
```

Note the "for"-loop termination condition!

We depend here on the meaning of "true" and "false" in C.

Note use of commas in the "for"-loop header.

Last examples: more idiomatic

- Char pointers were dereferenced
  - Value of dereference directly used to control loop.
- Char pointers were incremented
  - The most idiomatic code (not shown) combines dereferencing with incrementing
  - Example: *c++
  - Only works because "*" has a higher precedence than "++"
  - Meaning of example: "read the value stored in variable 'c', read the memory address corresponding to that value, return the value in that address as the expression value, and then increment the address stored in variable 'c'."

And tighter still…

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

#define MAX_STRING_LEN 100

int stringlength_5(char a_string[])
{
        char *c;

        for (c = a_string; *c; c++);

        return c - a_string;
}

int main(int argc, char *argv[])
{
        char buffer[MAX_STRING_LEN];

        if (argc == 1) { exit(0); }
        strncpy(buffer, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_5(buffer));

         exit(0);
}
```