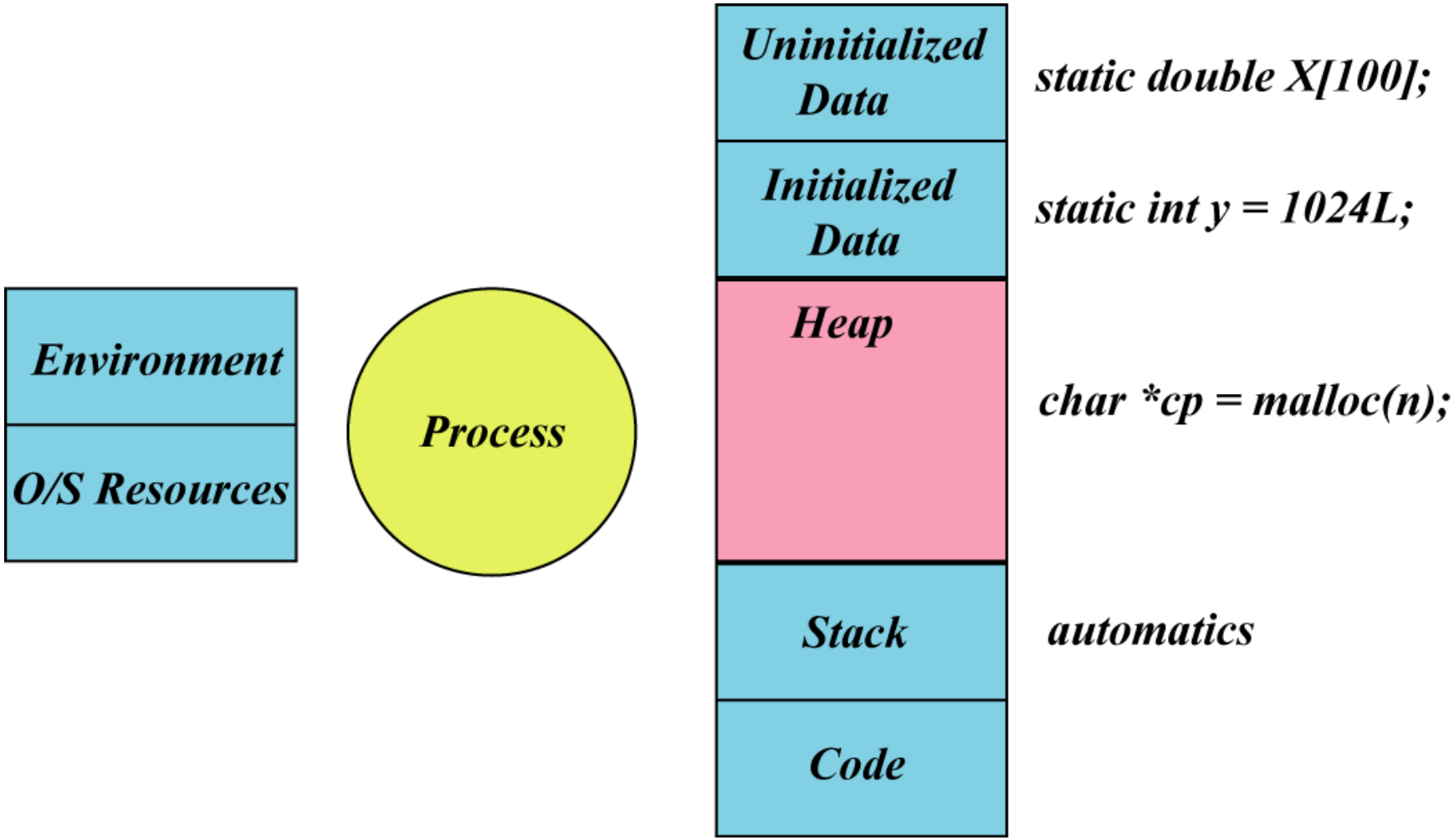
Dynamic memory in C

- The C memory model
- Managing the heap: `malloc()` and `free()`
- The void pointer and casting
- Memory leaks
- Applying the concepts:
  - **arrays that grow**
  - **linked lists**
  - **binary trees**
  - **hash maps**

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 "bottom" of writable memory (low 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 upward**
- Similarly, as memory is explicitly requested for allocation, the heap **grows downward**
- 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
- 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 large unused area of memory if program input sizes for program 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 a macro 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**.
  - All memory used in heap is 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 both allocates and initializes the block of heap memory
  - One adjusts heap structures to change size of a previously allocated block/chunk
  - Etc.
- 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.

Arrays that grow

- All of our C programs using arrays to date have been static in size
  - Assignment specifications state the largest input size.
  - Memory is allocated for these arrays from the C compiler and run time
  - We never need to manage this memory.
- Arrays are very handy structure
  - Easy to index and access (O(1) operations)
  - Contiguous block of memory can be exploited by other functions (qsort, memcopy).
- Therefore we would like to keep the convenience of arrays but also obtain the benefits of dynamic memory
  - … and do so without having to write more complex structures like lists, heaps, etc.

Nameval array

- Suppose we wish to maintain an array of <name, value> pairs
  - Name is a string
  - Value is an integer
- We want to add new items to our array as the arrive
- If there is not enough room in the array, we want to grow it.
- To support this we'll keep the array's size and items-to-date associated with the array via a struct.
  - Note use of "typedef"

```
typedef struct Nameval Nameval;
struct Nameval {
    char   *name;
    int     value;
};
```

```
struct Nvtab {
    int    nval;
    int    max;
    Nameval *nameval;
} nvtab;

enum { NVINIT = 1, NVGROW = 2 };
```

addname

```
int addname(Nameval newname)
{
    Nameval *nvp;

    if (nvtab.nameval == NULL) { /* first use of array */
        nvtab.nameval =
            (Nameval *) malloc(NVINIT * sizeof(Nameval));
        if (nvtab.nameval == NULL) { return -1; }
        nvtab.max = NVINIT;
        nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) {
        nvp = (Nameval *) realloc(nvtab.nameval,
            (NVGROW * nvtab.max) * sizeof(Nameval));
        if (nvp == NULL) { return = -1; }
        nvtab.max = NVGROW * nvtab.max;
        nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```