

Memory Allocation

Alan L. Cox
alc@rice.edu

Objectives

Be able to recognize the differences between static and dynamic memory allocation

Be able to use malloc() and free() to manage dynamic memory in your programs

Be able to analyze programs for memory management related bugs

Big Picture

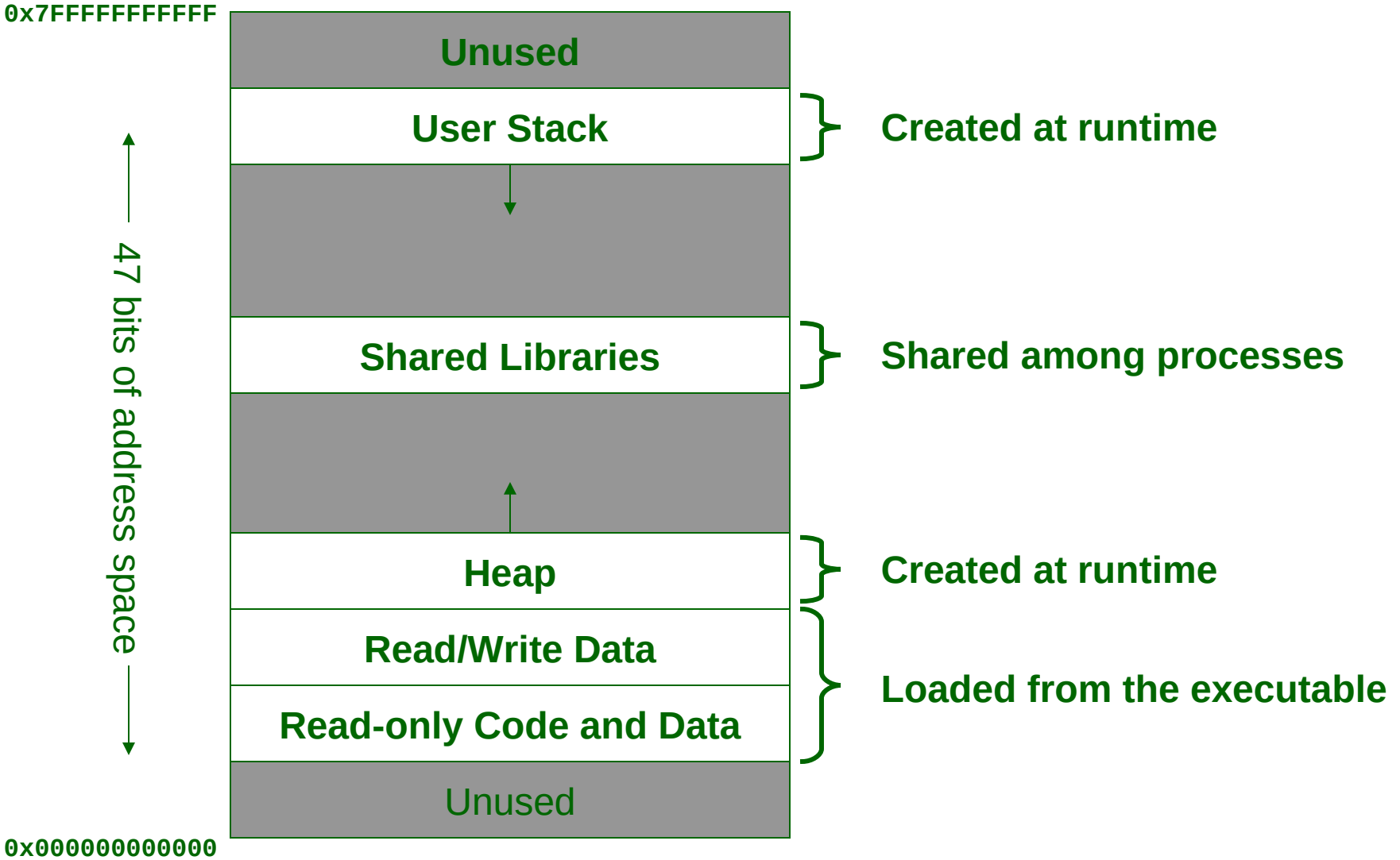
C gives you access to underlying data representations & layout

- ♦ **Needed for systems programming**
- ♦ **Dangerous for application programming**
- ♦ **Necessary to understand**

Memory is a finite sequence of fixed-size storage cells

- ♦ **Most machines view storage cells as bytes**
 - “byte-addresses”
 - Individual bits are not addressable
- ♦ **May also view storage cells as words**

A Running Program's Memory



Allocation

For all data, memory must be *allocated*

- ◆ **Allocated = memory space reserved**

Two questions:

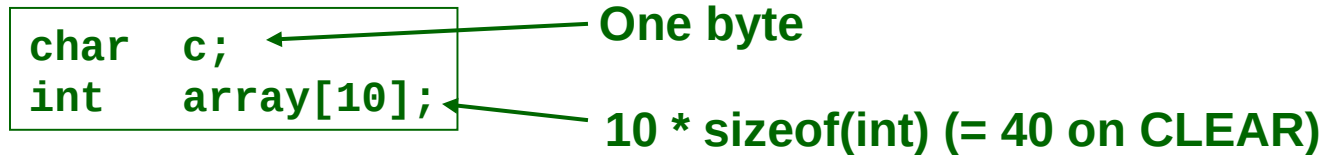
- ◆ **When do we know the size to allocate?**
- ◆ **When do we allocate?**

Two possible answers for each:

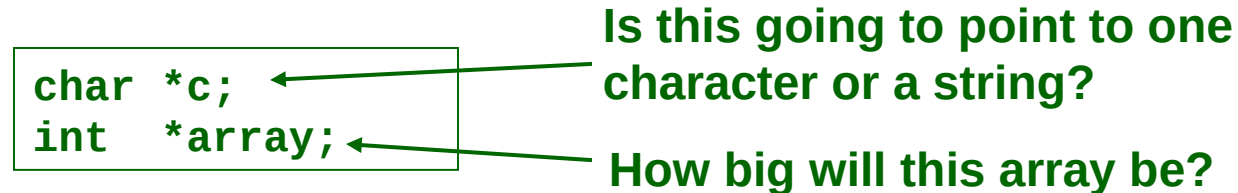
- ◆ **Compile-time (*static*)**
- ◆ **Run-time (*dynamic*)**

How much memory to allocate?

Sometimes obvious:



Sometimes not:



- ◆ **How will these be used???**
 - Will they point to already allocated memory (what we've seen so far)?
 - Will new memory need to be allocated (we haven't seen this yet)?

malloc()

Won't continually remind you of this

```
#include <stdlib.h>  
  
int *array = malloc(num_items * sizeof(int));
```

Allocate memory dynamically

- ◆ **Pass a size (number of bytes to allocate)**
 - Finds unused memory that is large enough to hold the specified number of bytes and reserves it
- ◆ **Returns a void * that points to the allocated memory**
 - No typecast is needed for pointer assignment
- ◆ **Essentially equivalent to new in Java and C++**

Using malloc()

```
int *i;
int *array;

i = malloc(sizeof(int));
array = malloc(num_items * sizeof(int));

*i = 3;
array[3] = 5;
```

Statically allocates space for 2 pointers

Dynamically allocates space for data

i and array are interchangeable

- ◆ Arrays \approx pointers to the initial (0th) array element
- ◆ **i** could point to an array, as well
- ◆ May change over the course of the program

Allocated memory is not initialized!

- ◆ **calloc zeroes allocated memory (otherwise, same as malloc; details to come in lab)**

Using malloc()

Always check the return value of system calls like malloc() for errors

```
int *a = malloc(num_items * sizeof(int));
if (a == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(1);
}
```

Terminate now!
And, indicate error.

- ♦ **For brevity, won't in class**
 - Lab examples will
- ♦ **Textbook uses capitalization convention**
 - Capitalized version of functions are wrappers that check for errors and exit if they occur (i.e. Malloc)
 - May not be appropriate to always exit on a malloc error, though, as you may be able to recover memory

When to Allocate?

Static time

- ♦ Typically global variables:

```
int value;
int main(void)
{
    ...
}
```

- ♦ Only one copy ever exists, so can allocate at compile-time

Dynamic time

- ♦ Typically local variables:

```
int f(...)
{
    int value;
    ...
}
int main(void)
{
    ...
}
```

- ♦ One copy exists for each call - may be unbounded # of calls, so can't allocate at compile-time


When to Allocate?

Static time

- ◆ Some local variables:

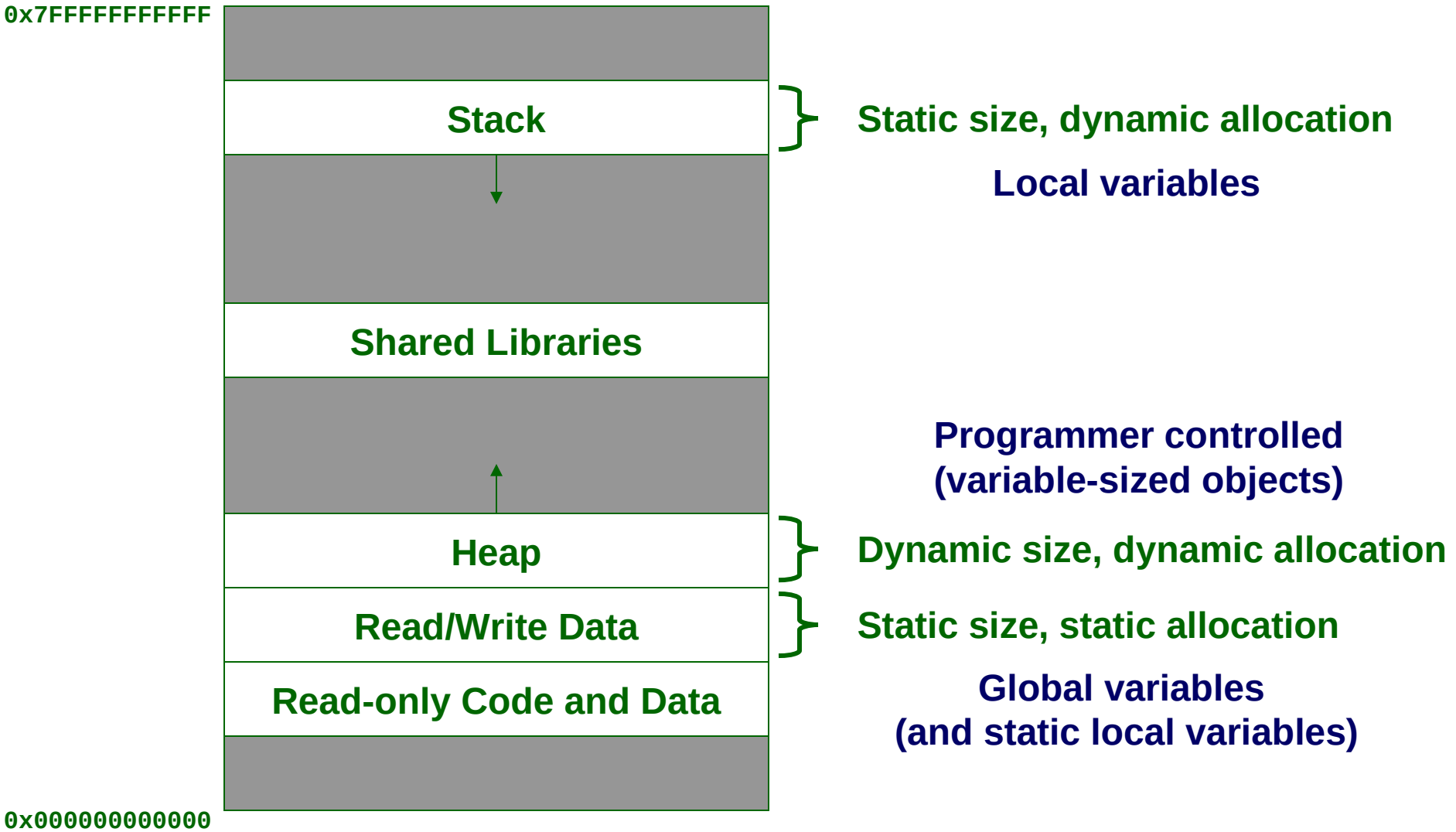
```
int f(...)
{
    static int value;
    ...
}
int main(void)
{
    ...
}
```

One copy exists
for all calls -
allocated at
compile-time



Confusingly, local
static has nothing to
do with global **static**!

Allocation in Process Memory



Deallocation

Space allocated via declaration (entering scope) is deallocated when exiting scope

```
... f(void)
{
    int y;
    int array[10];
    ...
}
```

- ♦ **Can't refer to `y` or `array` outside of `f()`, so their space is deallocated upon return**

Deallocation

malloc() allocates memory explicitly

- ◆ **Must also deallocate it explicitly (using free())!**
- ◆ **Not automatically deallocated (garbage collected) as in Python and Java**
- ◆ **Forgetting to deallocate leads to memory leaks & running out of memory**

```
int *a = malloc(num_items * sizeof(int));  
...  
free(a);  
...  
a = malloc(2 * num_items * sizeof(int));
```

Must not use a freed pointer unless reassigned or reallocated

Deallocation

Space allocated by malloc() is freed when the program terminates

- ♦ **If data structure is used until program termination, don't need to free**
- ♦ **Entire process' memory is deallocated**

Back to create_date

```
Date *
create_date3(int month,
             int day,
             int year)
{
    Date *d;

    d->month = month;
    d->day   = day;
    d->year  = year;

    return (d);
}
```



```
Date *
create_date3(int month,
             int day,
             int year)
{
    Date *d;

    d = malloc(sizeof(Date));

    d->month = month;
    d->day   = day;
    d->year  = year;

    return (d);
}
```


Pitfall

```
void
foo(void)
{
    Date *today;

    today = create_date3(1, 24, 2017);

    /* use "today" */
    ...

    return;
}
```

Potential problem:
memory allocation is
performed in this
function (may not know
its implementation)

Memory is still allocated for
"today"!

Will never be deallocated (calling
function doesn't even know
about it)

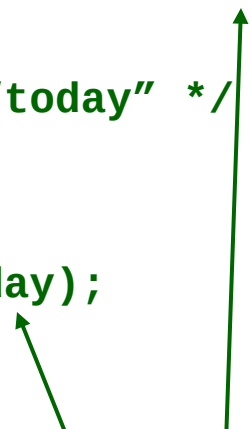
Possible Solutions

```
void
foo(void)
{
    Date *today;

    today = create_date3(...);

    /* use "today" */
    ...

    free(today);
    return;
}
```




Explicitly deallocate memory - specification of create_date3 must tell you to do this

```
void
foo(void)
{
    Date *today;

    today = create_date3(...);

    /* use "today" */
    ...

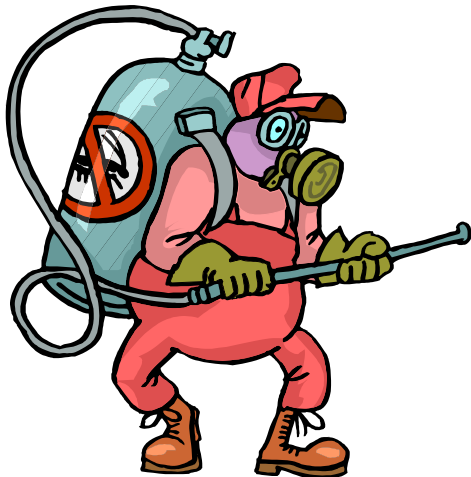
    destroy_date(today);
    return;
}
```



Complete the abstraction - "create" has a corresponding "destroy"



Common Memory Management Mistakes



What's Wrong With This Code?

```
int *f(...)
{
    int i;
    ...
    return (&i);
}
```

```
int *make_array(...)
{
    int array[10];
    ...
    return (array);
}
```

Consider `j = *f()`

Leads to referencing deallocated memory

- ◆ **Never return a pointer to a local variable!**

Behavior depends on allocation pattern

- ◆ **Space not reallocated (unlikely) → works**
- ◆ **Space reallocated → very unpredictable**

One Solution

```
int *f(...)
{
    int *i_ptr =
        malloc(sizeof(int));

    ...
    return (i_ptr);
}
```

```
int *make_array(...)
{
    int *array =
        malloc(10 * sizeof(int));

    ...
    return (array);
}
```

Allocate with malloc(), and return the pointer

- ◆ Upon return, space for local pointer variable is deallocated
- ◆ But the malloc-ed space isn't deallocated until it is free-d
- ◆ Potential memory leak if caller is not careful, as with create_date3...

What's Wrong With This Code?

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc(N * sizeof(int));
    int i, j;
    for (; i < N; i += 1)
        for (; j < N; j += 1)
            y[i] += A[i][j] * x[j];
    return (y);
}
```

Initialization
loop for

~~i=0~~
j=0

malloc-ed & declared space is not initialized!

- ♦ **i, j, y[i] initially contain unknown data - garbage**
- ♦ **Often has zero value, leading to seemingly correct results**

What's Wrong With This Code?

```
char **p;  
int    i;  
  
/* Allocate space for M*N matrix */  
p = malloc(M * sizeof(char));  
  
for (i = 0; i < M; i++)  
    p[i] = malloc(N * sizeof(char));
```

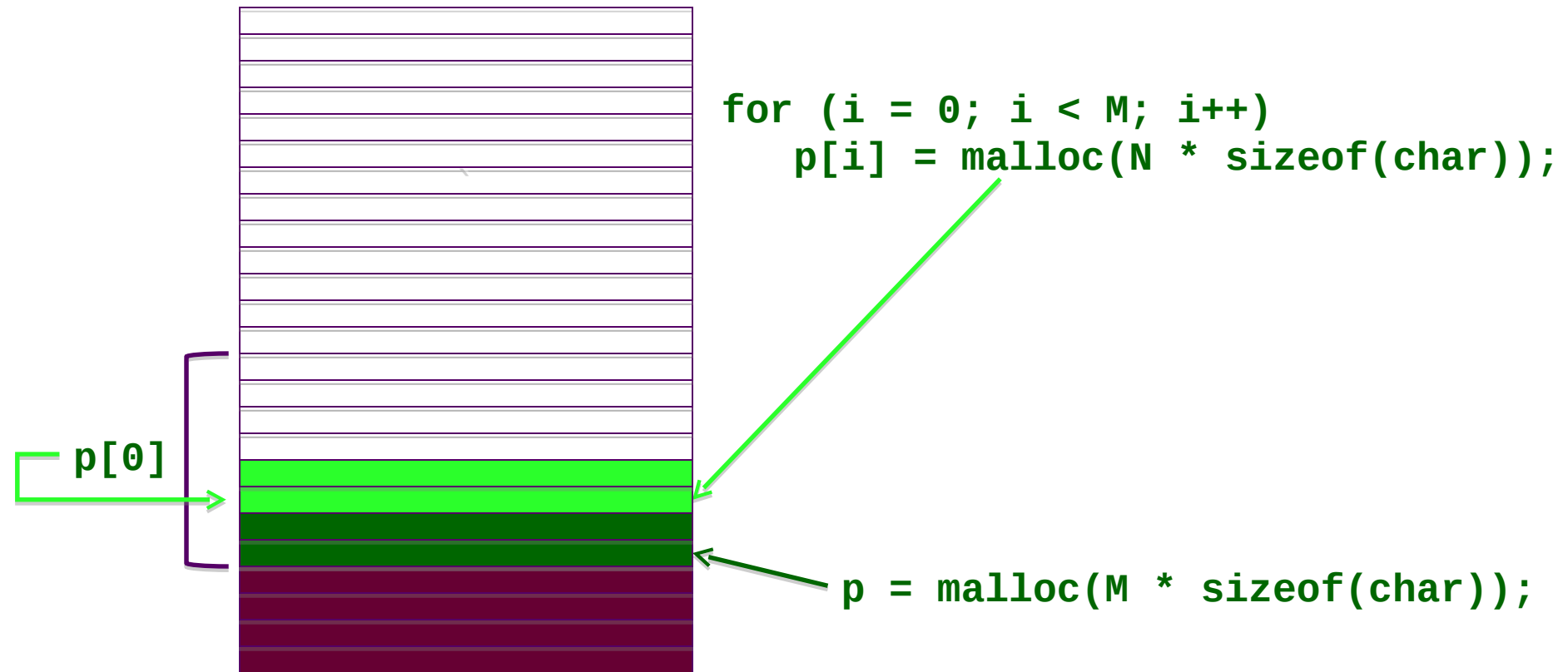
char *

Allocates wrong amount of memory

- ◆ **Leads to writing unallocated memory**

Explanation

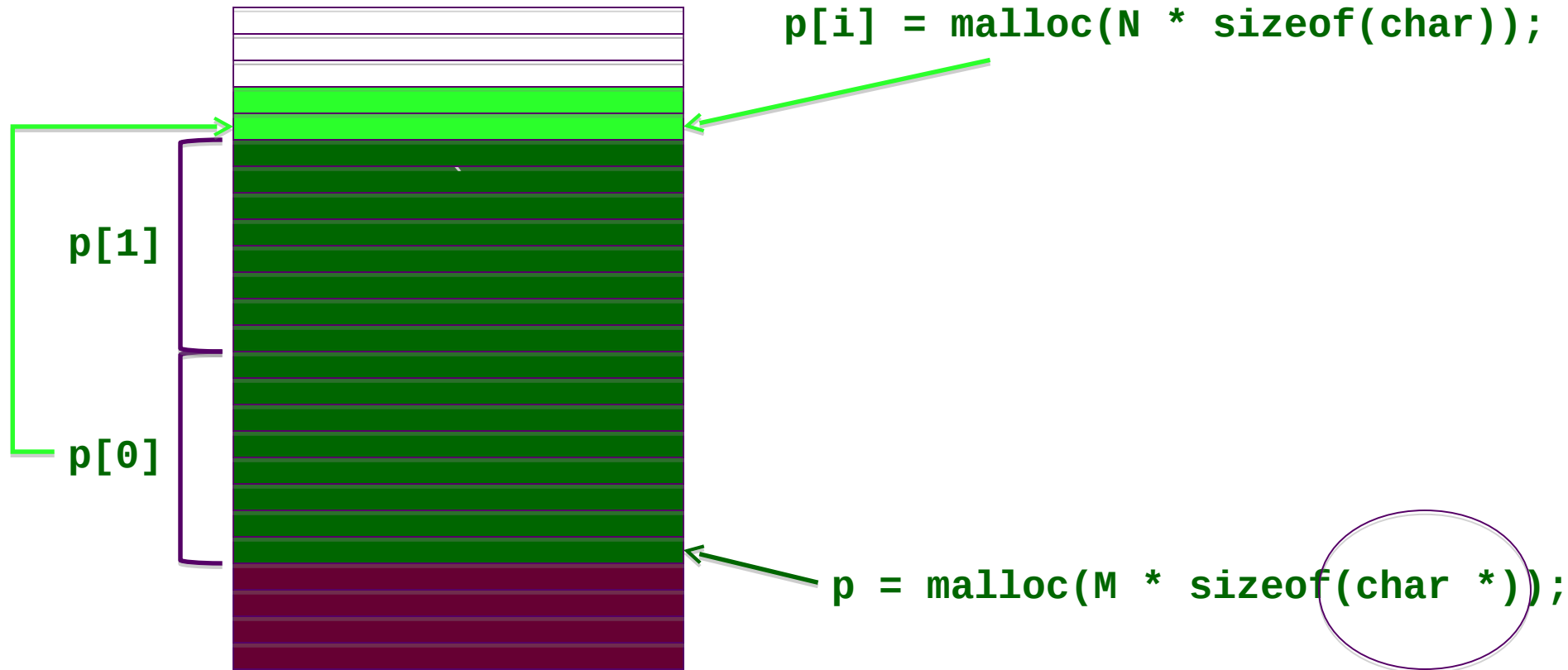
Heap region in memory (each rectangle represents one byte)
Assume $M = N = 2$, a memory address is 64 bit or 8 byte



Corrected code

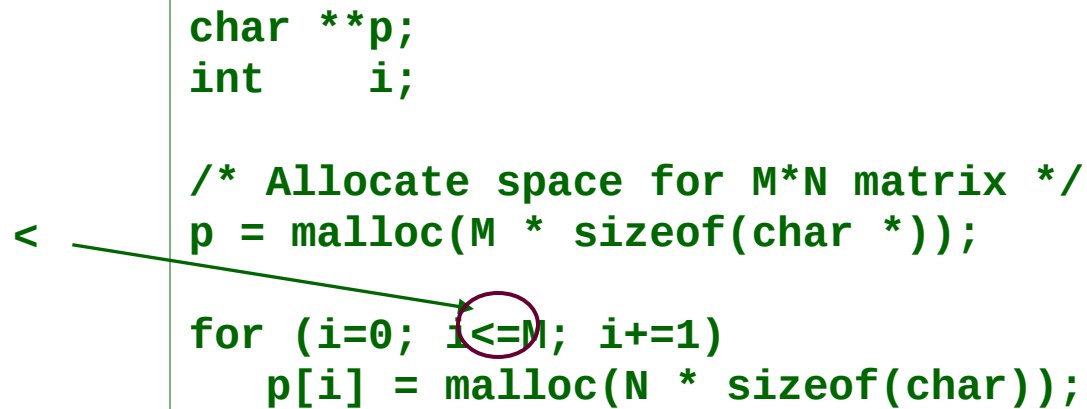
Heap region in memory (each rectangle represents one byte)
Assume $M = N = 2$, a memory address is 64 bit or 8 byte

```
for (i = 0; i < M; i++)  
    p[i] = malloc(N * sizeof(char));
```



What's Wrong With This Code?

```
char **p;  
int    i;  
  
/* Allocate space for M*N matrix */  
p = malloc(M * sizeof(char *));  
  
for (i=0; i<=M; i+=1)  
    p[i] = malloc(N * sizeof(char));
```



Off-by-1 error

- ◆ **Uses interval 0...M instead of 0...M-1**
- ◆ **Leads to writing unallocated memory**

Be careful with loop bounds!

Using const

const int * size

- ◆ Pointer to a constant integer
- ◆ Cannot write to *size

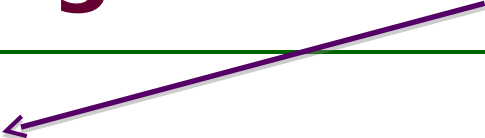
int * const size

- ◆ Constant pointer to an integer
- ◆ Cannot modify the pointer (size)
- ◆ Can write to *size

```
char *  
xyz(char * to, const char * from)  
{  
    char *save = to;  
    for (; (*to = *from); ++from, ++to);  
    return(save);  
}
```

What's Wrong With This Code?

```
char *s = "1234567";  
...  
char t[7];  
strcpy(t, s);
```




```
char *  
strcpy(char * to, const char * from)  
{  
    char *save = to;  
    for (; (*to = *from); ++from, ++to);  
    return(save);  
}
```

t[] doesn't have space for string terminator

- ◆ Leads to writing unallocated memory

One way to avoid:



```
char *s = "1234567";  
...  
char *t = malloc((strlen(s) + 1) * sizeof(char));  
strcpy(t, s);
```

What's Wrong With This Code?

```
/* Search memory for a value. */
/* Assume value is present.   */
int *search(int *p, int value) {
    while (*p > 0 &&
           *p != value)
        p += sizeof(int);
    return (p);
}
```

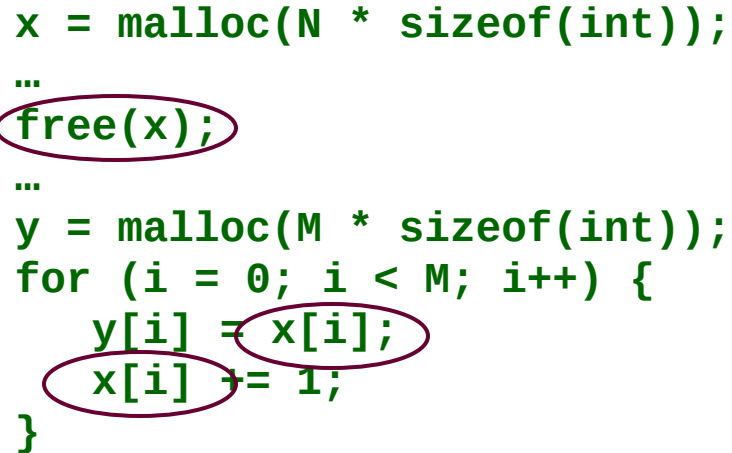
← p += 1;

Misused pointer arithmetic

- ♦ Search skips some data, can read unallocated memory, and might not ever see value
- ♦ Should never add `sizeof()` to a pointer
- ♦ Could consider rewriting this function & its uses to use array notation instead

What's Wrong With This Code?

```
x = malloc(N * sizeof(int));  
...  
free(x);  
...  
y = malloc(M * sizeof(int));  
for (i = 0; i < M; i++) {  
    y[i] = x[i];  
    x[i] += 1;  
}
```



Premature free()

- ◆ Reads and writes deallocated memory

Behavior depends on allocation pattern

- ◆ Space not reallocated → works
- ◆ Space reallocated → very unpredictable

What's Wrong With This Code?

```
void foo(void) {  
    int *x = malloc(N * sizeof(int));  
    free(x); →  
    return;  
}
```

Memory leak - doesn't free malloc-ed space

- ◆ **Data still allocated, but inaccessible, since can't refer to x**

Slows future memory performance

What's Wrong With This Code?

```
struct ACons {
    int      first;
    struct ACons *rest;
};
typedef struct ACons *List;

List cons(int first, List rest) {
    List item = malloc(sizeof(struct ACons));
    item->first = first;
    item->rest = rest;
    return (item);
}

void foo(void) {
    List list = cons(1, cons(2, cons(3, NULL)));
    ...
    free(list);
    return;
}
```

A peek at one way to define lists

Example continued

Memory leak - frees only beginning of data structure

- ◆ Remainder of data structure is still allocated, but inaccessible
- ◆ Need to write deallocation (destructor) routines for each data structure

Putting it together...

string

pointer

struct

malloc()

simple I/O

simple string operations

What does action1() do?

```
struct thing {  
    char *stuff;  
    struct thing *another thing;
```

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

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

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

```
struct thing *x = malloc(sizeof(struct thing)),
```

action1() inserts a new node storing the specified string into the linked list

```
x->stuff = malloc(strlen(stuff) + 1);
```

```
strcpy(x->stuff, stuff);
```

```
x->another thing = *y;
```

What does action2() do?

```
struct thing {
    char *stuff;
    struct thing *another_thing;
};

void
action2(struct thing **y)
{
    struct thing *x;
```

action2() prints the strings stored in the linked list nodes sequentially

```
printf("%s ", x->stuff);
```

```
    y = &x->another_thing;
Cox / Fagan Memory Allocation
```

```
}
```

What does action3() do?

```
struct thing {
    char *stuff;
    struct thing *another_thing;
};

int
action3(struct thing **y, const char *stuff)
{
    struct thing *x;
    while ((x = *y) != NULL) {
```

action3() finds out whether a string
is stored in the linked list

```
    else
```

What does action4() do?

```
struct thing {  
    char *stuff;  
    struct thing *another_thing;  
};  
  
void  
action4(struct thing **y, const char *stuff)  
{  
    struct thing *x;
```

action4() deletes the first list node that stores the specified string

Next Time

Lab: Debugging Assembly