CS 333

Intro to Operating Systems
malloc() and friends

Dynamic Memory Allocation and Management with `malloc()` and Related Functions

- `calloc()`
- `realloc()`
- `strdup()`
- `free()`
  - and cousin `alloca()`

You've may have been using `new` and `delete` in your C++ programs to dynamically allocate and release memory while your program is running.

If you've only been using Java, you may have forgotten how to reclaim memory.

If you used Python, … you have no idea.

In this class you'll be using **`malloc`**`()`, **`calloc`**`()`, **`realloc`**`()`, and **`free`**`()`.

You'll also probably like **`strdup`**`()`, a lot.

Kinds of Memory Allocation in C

Kind of memory	When is it allocated?	When is it deallocated?
automatic	created on stack on call to a function	deallocated on return from function
static	before main starts	when program terminates
dynamic	**calls to `malloc`, `realloc`, or `calloc`**	**when the program calls `free` or terminates.**

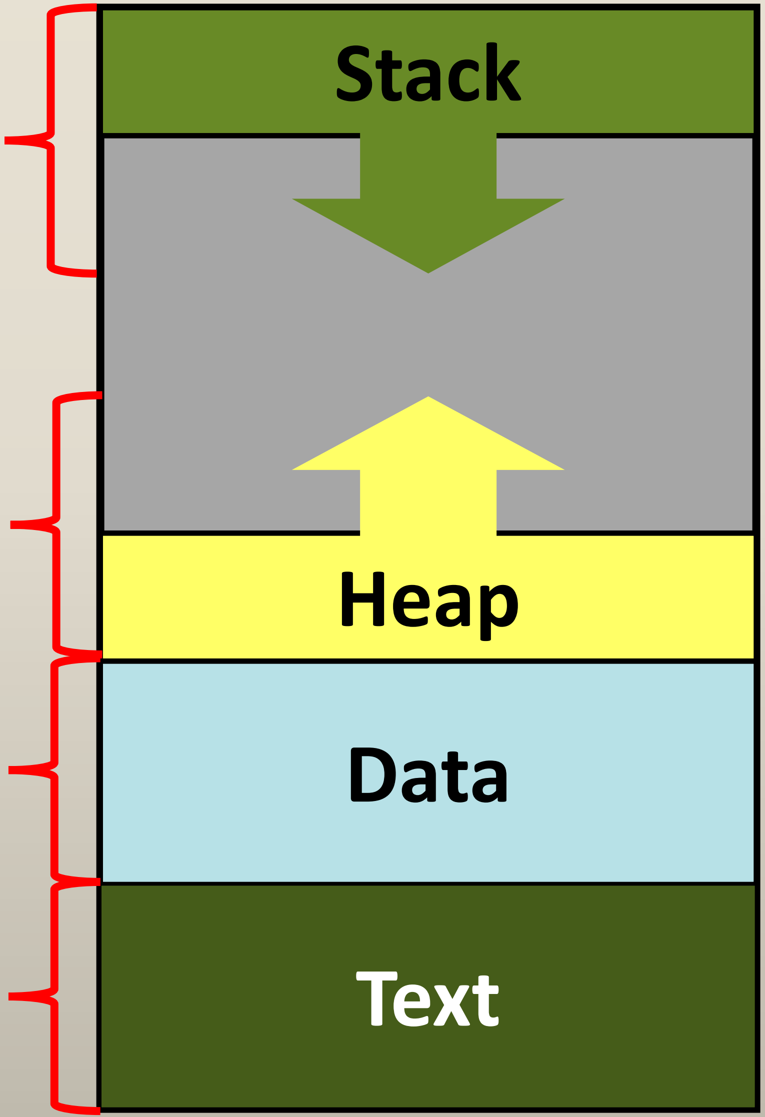
Memory Layout of a Process

Function call stack: activation records and automatic variables

Data returned from calls to **`malloc`**, **`realloc`**, or **`calloc`**

Initialized and uninitialized statically allocated data

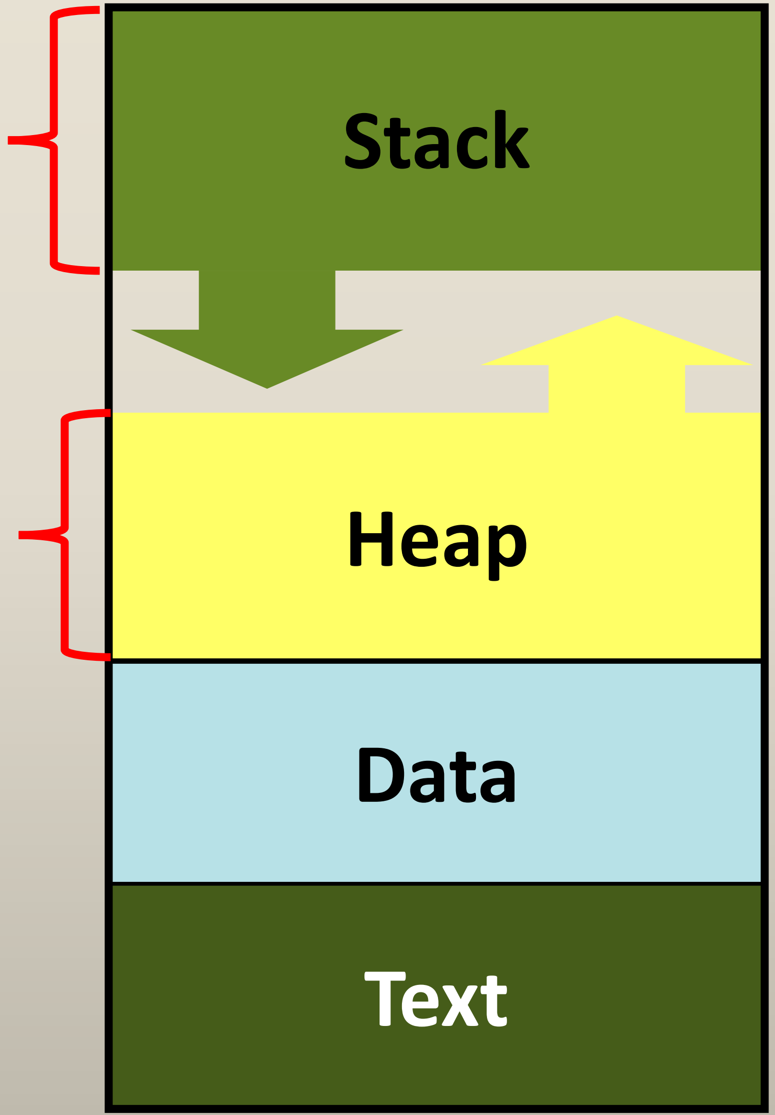
Your program instructions



Memory Layout of a Process

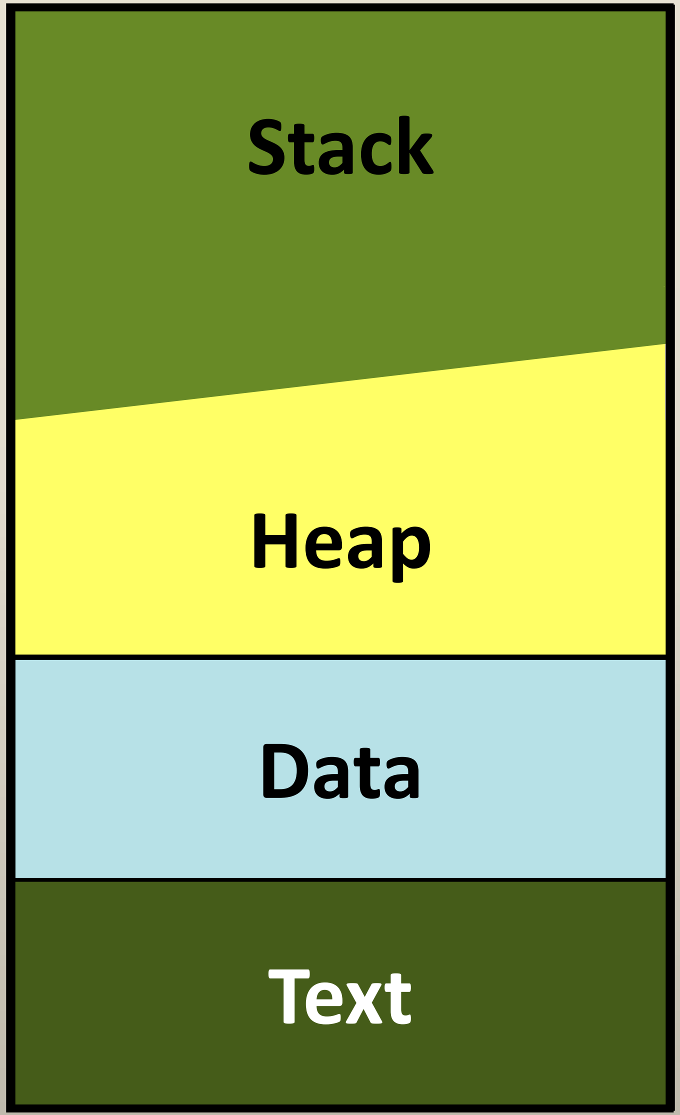
As your program calls functions and returns, the stack grows and shrinks.

As your program allocates dynamic memory, the heap grows. Your program consumes the heap.



Memory Layout of a Process

If the stack and heap meet or worse overlap, bad things happen to your program.



In what Segment will the Address be Located

```
int x;
int y = 0;

int main () {
      int a = 0;
      foo (a);
      printf("a: %d\n", a);
}
void foo (int d) {
      int b = 2;
      int *z = malloc(sizeof(int));
      static int c = 0;
      *z = 5;
      printf("d: %d z: %d\n", d, *z);
      c++;
}
```

Address	Process Segment
x	data
y	data
main	text
a	stack
foo	text
b	stack
d	stack
z	stack
*z	heap
c	data

The `sizeof()` Operator

Returns the **size** of a variable **or** datatype, measured in the **number of bytes** required for the type or structure.

I really like to ask about the `sizeof()` operator on exams.

I have an example in

```
~rchaney/Classes/cs333/src/sizeof/sizeof.c
```

The `sizeof()` Operator

```
char c;
char *cp;
int i;
int *ip;

sizeof(char);       // Guaranteed to be 1.
sizeof(c);
sizeof(int)         // System dependent, but 4 for us.
sizeof(i);

sizeof(char *);     // All pointers are the same size.
sizeof(cp);         // On our system, pointers are
sizeof(int *);      // 8 bytes (64 bits).
sizeof(ip);
```

The `sizeof()` Operator

```
typedef struct account_s {
   int account_number;
   char first_name[50];
   char last_name[50];
   float balance;
} account_t;

account_t act;
account_t *act_p;
```

The sum of the sizes of the members of a structure may differ from the `sizeof()` for the structure.

```
sizeof(account_t);
sizeof(struct account_s);
sizeof(act);

sizeof(account_t *);
sizeof(struct account_s *);
sizeof(act_p);
```

Remember, all pointers are the same size.

The `malloc()` Call

The allocated memory is not initialized

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

Straight from the heap.

What is a `void *`?

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

The allocated memory is not initialized

The number of **bytes** you wish to allocate.

The `malloc()` function allocates **`size`** bytes and **returns a pointer to the allocated memory.**

If the call to `malloc()` fails, it returns a NULL pointer.

The allocated memory is not initialized.

The allocated memory is not initialized

The void * Pointer

The void * pointer is a **generic** pointer in C.

A void pointer is a pointer that has no associated data type with it. A void pointer can hold an **address** of any type and can be **type cast** to any type.

The type of the data on the other side of a void * pointer is **opaque**, until the void * pointer is **type cast** to another type.

Wanna bet this is an exam question?

Type Casting in C

Type casting is a way to convert a variable from one data type to another data type.

`float a =` ~~`static_cast<float>(5)`~~
~~`/ static_cast<float>(2);`~~

This is the C++ way to do a type cast.

```
float a = ((float) 5) / ((float) 2);
```

The type **to** which you want to cast the value.

This is the C way to do a type cast.

Type Casting in C

Type coercion is the **automatic** conversion of a datum from one data type to another within an expression.

```
double x = 1;
```

Type casting is an explicit type conversion **defined within a program**. It is defined by the user in the program.

```
double da = 3.3;
double db = 3.3;
double dc = 3.4;
int result = (int)da + (int)db + (int)dc; //result == 9
```

Type Casting in C

```
int i;
int *ip = &i;
void *vp;

vp = (void *) ip;

ip = (int *) vp;
```

Type cast the `int` pointer to a `void` pointer.

Type cast the `void` pointer to an `int` pointer.

Back to malloc()

I want to allocate a block of memory large enough for 1,000 characters and I want a block of memory for 1,000 integers.

```
char *cp;
int *ip;
```

```
cp = malloc( 1000 * sizeof( char ) );
ip = malloc( 1000 * sizeof( int ) );
```

Sadly, many compilers now let you get away without explicit the cast.

The `malloc()` Call

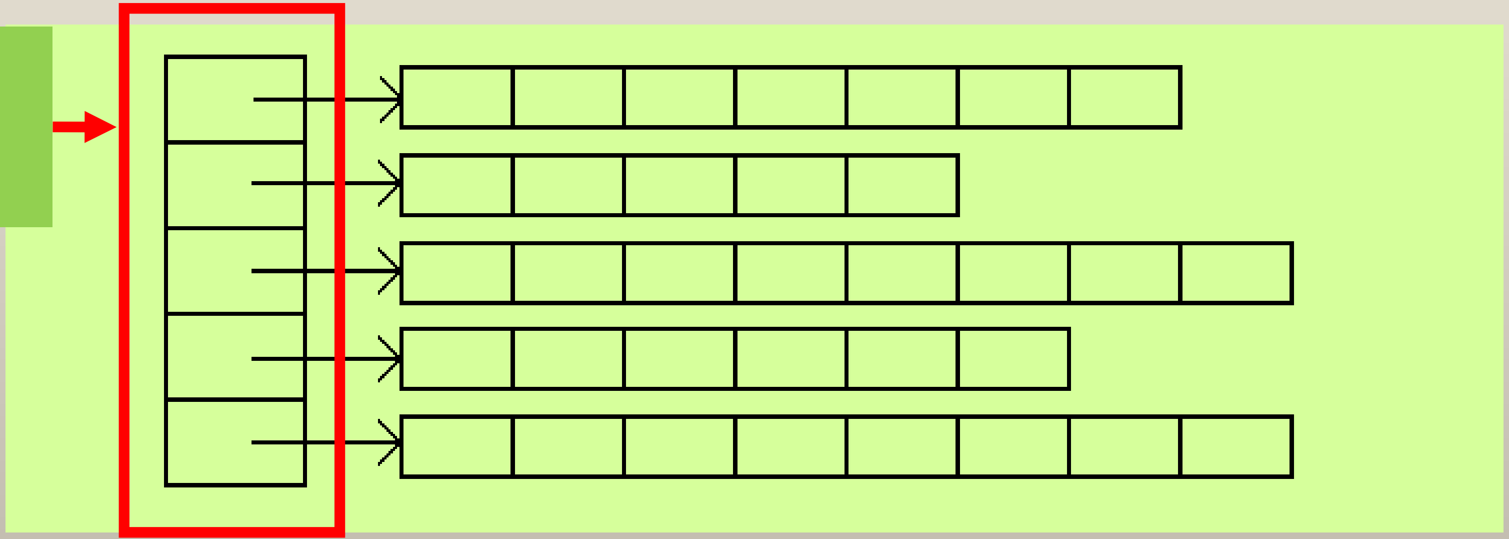
I want to allocate a block of memory large enough for 100 **pointers** to character arrays.

```
char **cp;
```

Notice this is a `char *`

```
cp = (char **) malloc( 100 * sizeof(char *) );
```

The call above only allocates this portion.



Does this look like a ragged array?
It should!

The `malloc()` Call

I want to allocate a block of memory large enough for 1,000 **account_t** structures.

```
typedef struct account_s {
    int account_number;
    char first_name[50];
    char last_name[50];
    float balance;
} account_t;
account_t *act_p;

act_p = (account_t *) malloc( 1000 * sizeof(account_t) );
```

The pointer type.

The structure type.

The malloc() Call

```
typedef struct account_s {
    int account_number;
    char first_name[50];
    char last_name[50];
    float balance;
} account_t;
int i;
```

How you access the elements in the allocated array of account_t?

```
account_t *act_p = NULL;
act_p = (account_t *) malloc( 1000 * sizeof(account_t) );
for (i = 0; i < 1000; i++) {
     act_p[i].account_number = i;
}
```

Notice the use of the dot, not use of the ->

The `memset()` Call

The memory returned from a call to `malloc()` **is NOT initialized**.

You must assume that it is garbage.

If you want to set it to a constant value, use

Guess what often immediately follows a call to `malloc()`?

NAME

`memset` - fill memory with a constant byte

SYNOPSIS

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

DESCRIPTION

The `memset()` function fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c`.

RETURN VALUE

The `memset()` function returns a pointer to the memory area `s`.

This is a valid call:

```
memset(arr, ' ', sizeof(arr));
```

Other mem*() Calls

In addition to the ever useful `memset()` call, there are other functions that work directly on memory.

NAME

memcpy - copy memory area

SYNOPSIS

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

Not all memory is nice a NULL terminated string.

NAME

memcmp - compare memory areas

SYNOPSIS

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

The calloc() Call

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size );
```

The size of **each** element of array.

Number of elements in array.

The allocated memory is initialized

The calloc() allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory.

The allocated memory is initialized to all zeroes.

The free() Call

```
#include <stdlib.h>

void free(void *ptr);
```

De-allocates memory. Returns it so that is can be reused.

The free() call de-allocates the memory space pointed to by ptr, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**.

If free(ptr) has already been called before on that pointer, **undefined behavior occurs**.

Don’t free memory that has already been deallocated.

The `free()` Call

The argument to `free()` **must** be the address of the **beginning** of a currently allocated block in the **heap**.

Blocks of memory deallocated by `free()` become available for future allocation by `malloc()`, `realloc()`, or `calloc()`, this is a kind of recycling.

Calling `free()` on a NULL pointer is okay.

Common Mistakes Related to `free()`

1. Attempting to `free()` stack or static memory.
   - Don't do it, free works only with heap memory. If you are lucky, `free()` will only silently fail.
2. Attempting to `free()` only part of a block.
   - Don’t try to free a pointer that points into the middle of a dynamically allocated block. It will be U-G-L-Y.
3. Attempting to `free()` a block that is already free.
   - This usually causes trouble.

Common Mistakes Related to `free()`

Another big mistake related to `free()` is

Wait for it....

Forgetting to call `free()`!

This kind of mistake is called a memory leak. The easy way to check for memory leeks it to run your code through `valgrind`. We'll cover `valgrind` in a lab.

The `realloc()` Call

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

Changes the size of the block of memory pointed to by `ptr` and it **copies** the contents of the old block into the new block.

```
void *realloc(void *ptr, size_t size);
```

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes.

The contents of the old pointer are **copied** into the newly allocated area.

The old block of memory will be automatically deallocated.

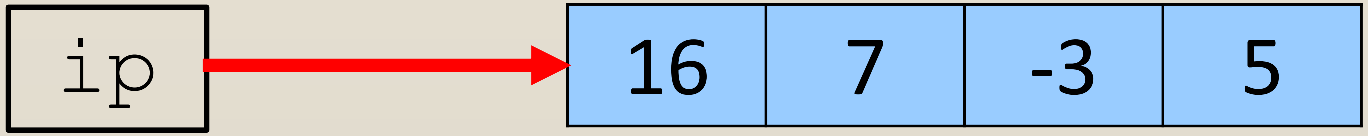
The `realloc()` Call

`realloc` **copies** a number of bytes from the beginning of the old block to the beginning of the new block. The number of bytes copied will be the old block size or the new block size, **whichever is smaller**.

When the new size is larger than the old size, the additional bytes are **not initialized** and should be assumed to contain garbage.

The `realloc()` Call

```
int *ip = NULL;
ip = (int *) realloc( ip, 4 * sizeof(int) );
ip[0] = 16; ip[1] = 7; ip[2] = -3; ip[3] = 5;
```

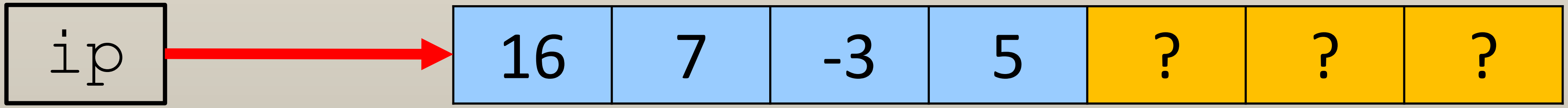


ip

16	7	-3	5

After the call to `realloc()` and some assignments.

```
ip = (int *) realloc( ip, 7 * sizeof(int) );
```



ip

16	7	-3	5	?	?	?

After the call to `realloc()`.

This works for pointers to pointers; such as **ragged** arrays.

The `strdup()` Call

```
#include <string.h>

char *strdup(const char *str);
```

The `strdup()` function returns a pointer to a new string which is a **duplicate** of the string `str`.

Memory for the new string is obtained with `malloc()`, and should be deallocated with `free()`.

The amount of memory allocated will be only as large as is necessary to hold the data.

The `strdup()` Call

```
char *cp1, *cp2;

cp1 = strdup("This is a string");
cp2 = strdup(cp1);
…
free(cp1);
free(cp2);
```

Being a string, the NULL is copied as well.

Since the memory was allocated with `malloc`, it should be deallocated with `free`.

The `alloca()` Call

```
#include <alloca.h>

void *alloca(size_t size);
```

Use of `alloca()` should be done only with great care.

The `alloca()` function allocates `size` bytes of space in the **stack frame** of the caller. This temporary space is **automatically freed** when the function that called `alloca()` returns to its caller.

Do not attempt to `free()` space allocated by `alloca()`!

The `alloca()` Call

Unlike the memory returned by `malloc`, `calloc`, and `realloc`, the memory returned by `alloca` is not from the heap, but **from the stack**. Do not try and use `alloca` for large chunks of memory.

Unlike memory returned by `malloc`, `calloc`, and `realloc`, you do not need to free the memory returned by `alloca`. It will be automatically returned to the stack when the function creating it returns.

The `alloca()` Call

- The `alloca()` function is super cool. However, you must be very careful about when you use it.
- Probably the best thing about `alloca()` is that its use does not fragment the heap.
- Since the memory allocated by `alloca()` comes from the stack, not heap, the heap will not become fragmented by its use.
- Do not use `alloca()` in a recursive function call.