Agenda

Pointer Arithmetic

void* revisited

A simple memory allocator

A simple implementation of malloc/free

A better implementation of malloc/free using a free list

Allocation strategies: First fit, worst fit, best fit

Fragmentation

Big fish, little fish revisited

Pointers Review – Fill in the blanks

```
int mystery(<pointer to int> x, <pointer to int> y) {
  int xx = <value at address x>;
  int yy = <value at address y>;
  xx++;
  yy++;
  <value at address x> = xx;
  <value at address y> = yy;
  return xx + yy;
}

int main() {
  int a = 2;
  int b = -4;
  <pointer to int> ptrA = <address of a>;
  <pointer to int> ptrB = <address of b>;

  int c = mystery(<address of a>, <address of b>);
  printf("%d\n", c); // what is the output?
}
```

Pointers – draw the stack diagram

```
int mystery(<pointer to int> x, <pointer to int> y) {
  int xx = <value at address x>;
  int yy = <value at address y>;
  xx++;
  yy++;
  <value at address x> = xx;
  <value at address y> = yy;
  return xx + yy;
}

int main() {
  int a = 2;
  int b = -4;
  <pointer to int> ptrA = <address of a>;
  <pointer to int> ptrB = <address of b>;

  int c = mystery(<address of a>, <address of b>);
  printf("%d\n", c); // what is the output?
  // stack diagram here
}
```

Advanced pointers: Pointer Arithmetic

Section 2.9.4

Useful in a few niche cases, specifically, working with generic blocks of memory

applications: databases, network messages, malloc

Recall: iterating over an array

```
int vals[4] = {1,2,3,4};
int* valptr = vals;

int v1 = vals[2];
int* v1ptr = valptr + 2;
int v2 = *v1ptr;
printf("%d %d\n", v1, v2);

for (int i = 0; i < 4; i++) {
  printf(“%d\n", vals[i]);
}

for (int* ptr = vals; ptr < vals+4; ptr++) {
  printf(“%p %d\n", ptr, *ptr);
}
```

Draw memory layout of vals

```
int vals[4] = {1,2,3,4};
int* valptr = vals;

int v1 = vals[2];
int* v1ptr = valptr + 2;
int v2 = *v1ptr;
printf("%d %d\n", v1, v2);
// draw stack diagram here

for (int i = 0; i < 4; i++) {
  printf(“%d\n", vals[i]);
}

for (int* ptr = vals; ptr < vals+4; ptr++) {
  printf(“%p %d\n", ptr, *ptr);
}
```

Suppose the beginning of the array is at location 0xffffeea0. What is the location of vals[0], vals[1], vals[2], etc. What is the location of valptr+1, valptr+2, etc?

Draw a stack diagram from this program

```
char vals[3] = {'h','I','\0'};
char* valptr = vals;

char v1 = vals[2];
char* v1ptr = valptr + 2;
char v2 = *v1ptr;
printf("%c %c\n", v1, v2);
// Draw stack here

for (int i = 0; i < 3; i++) {
  printf("%c\n", vals[i]);
}

for (char* ptr = vals; ptr < vals+3; ptr++) {
  printf("%p %c\n", ptr, *ptr);
}
```

IMPORTANT

When adding/subtracting offsets to pointers, we increment/decrement based on the pointer's type

Example: Suppose the address of x is 0x4c568000.

```
int* x = &a; x++; // x has address _____
char* x = &c; x++; // x has address _____
struct m
{
    int q;
    char buff[4];
};
struct m* x = &data; x++; // x has address _____
```

void* pointer

Pointer without an associated type

Can’t be de-referenced without first being cast to a type

Uses:

- thread API
- function arguments that accept more than one type
- working with structure-less blocks of memory

Which dereferences are safe?

Example: void*

```
void *gen_ptr;
int x = 3;
char ch = 'a';

gen_ptr = &x;  // gen_ptr can be assigned the address of an int
gen_ptr = &ch; // or the address of a char (or the address of any type)

int* int_ptr;
int_ptr = &x;

printf("The int value is %d\n", *int_ptr);
printf("The void* value is %c\n", * ((int*) gen_ptr));
printf("The void* value is %c\n", * ((char*) gen_ptr));
printf("The void* value is %f\n", * ((float*) gen_ptr));
return 0;
```

Example: void*

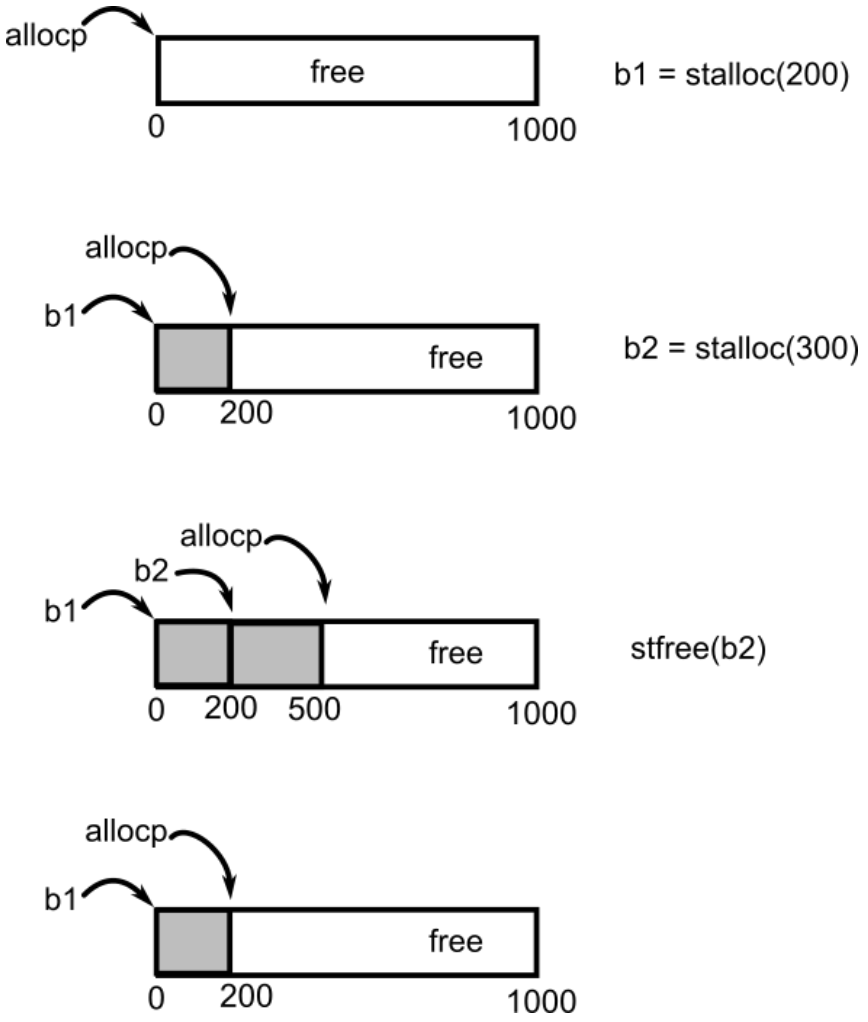
```
struct chunk {
  int size;
  struct chunk *next;
};

int main() {
  int size = sizeof(int) * 5;
  void *memory = malloc(size + sizeof(struct chunk));
  if (memory == NULL) {
    return 1;
  }

  struct chunk *cnk = (struct chunk*) memory;
  cnk->size = size;
  void* data = (void*) (cnk + 1);
  int* array = (int*) data;

  for (int i = 0; i < 5; i++) {
    array[i] = i;
  }
```

A simple allocator

Suppose all our program's dynamic memory comes from a large block of memory (e.g. a character array) allocated when the program starts.

Suppose we request/free memory from one end of the array (e.g. treat memory like a stack)

- When the user requests memory, they receive a block from the top of the stack.
- When they free memory, it must be the most recently acquired block.



Draw the stack diagram

```
static char allocbuf[ALLOCSIZE]; // our stack for allocating memory
static char* allocp = allocbuf;  // a pointer to the next free block in memory

char* stalloc(int n) {
    char* start = allocp;
    allocp = allocp + n;
    return start;
}

void stfree(char* p) {
   allocp = p;
}

int main() {
  char* b1 = stalloc(sizeof(int) * 5);
  char* b2 = stalloc(sizeof(char) * 10);
  stfree(b2);
  stfree(b1);
}
```

A simple allocator: properties

Used memory is always at the beginning of the array block

Free memory is always at the end of the array block

Only a single pointer is needed to keep track of the division between used and free memory

A simple allocator

What limitations does this approach to dynamic memory have?

- The user must deallocate memory in the opposite order that it is requested
- The user can ask for at most ALLOCSIZE blocks of memort

What edge cases should be handled by the allocator?

- We should check whether there is enough memory to satisfy a request (and return 0 if there isn't enough)
- We should ensure that our pointer always stays in the range of valid memory

Exercise: Edit the code on slide 13 so it's safe

Draw the stack diagram

```
static char allocbuf[ALLOCSIZE]; // our stack for allocating memory
static char* allocp = allocbuf;  // a pointer to the next free block in memory

char* stalloc(int n) {
    char* start = allocp;
    allocp = allocp + n;
    return start;
}

void stfree(char* p) {
   allocp = p;
}

int main() {
  char* b1 = stalloc(sizeof(int) * 5);
  char* b2 = stalloc(sizeof(char) * 10);
  stfree(b2);
  stfree(b1);
}
```

Another approach to allocation

Idea: Add meta data to allocations so that we can free memory in any order

Reading: mylloc.pdf (See slack)

`malloc` and `free` can be replaced with our own implementation!

How? By implementing new functions that follow the specification

- Analogies: overriding functions in a class, writing a game mod, or swapping a component in a car

sbrk

```
void *sbrk(intptr_t increment);
```

- **Program break**: the location of the end of the uninitialized data segment (a pointer to program memory)
- **sbrk** changes the location of the **program break** by the given number of bytes
- Increasing the program break allocates more memory to the process
- If successful, returns the previous program break.
  - e.g. the start of the newly allocated memory
- On error, the value `(void*) -1` is returned

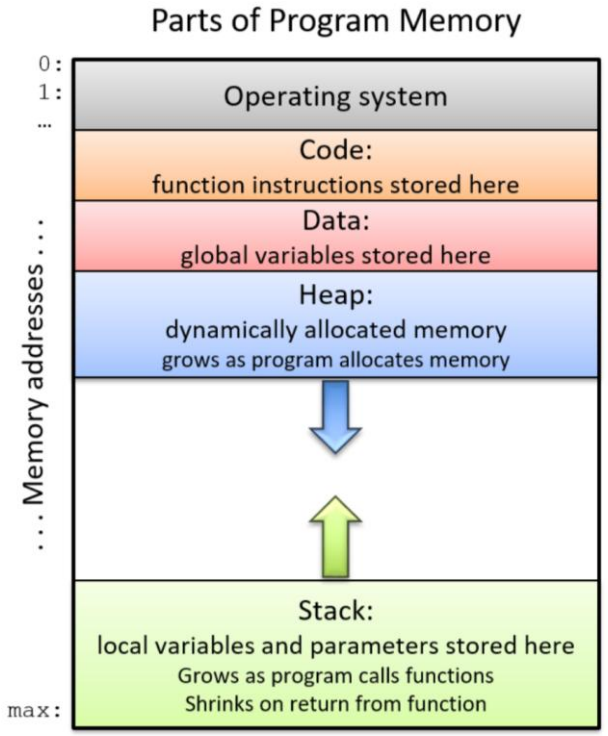


Figure 1. The parts of a program's address space.

Malloc specification

```
$ man malloc
```

```
The malloc() function allocates size
bytes and returns a pointer to the
allocated memory. The memory is not
initialized. If size is 0, then malloc()
returns either NULL, or a unique pointer
value that can later be successfully
passed to free().
```

Free specification

$ man free

```
The free() function frees the memory
space pointed to by ptr, which must have
been returned by a previous call to
malloc(), calloc(), or realloc().
Otherwise, or if free(ptr) has already
been called before, undefined behavior
occurs. If ptr is NULL, no operation is
performed.
```

Exercise: What if malloc/free were implemented as follows? Describe what malloc and free are doing.

```
void *malloc (size_t size) {
  if (size == 0){
    return NULL;
  }
  void *memory = sbrk(size);
  if (memory == (void *) -1) {
    return NULL;
  } else {
    return memory;
  }
}

void free(void *memory) {
  return;
}
```

Let’s test our new malloc/free

What does this code do?

```
// create array to hold memory requests
// initialize all elements to NULL

for (int j = 0 ; j < ROUNDS; j++) {
   for (int i= 0 ; i < LOOP ; i++) {
     int index = rand() % BUFFER;
     if (buffer[index] != NULL) {
       // free memory
     }
     else {
       // allocate memory
   }
  // print memory statistics
}

// free all memory
```

Demo: Compare the new malloc to libc's malloc

To use your new malloc,

```
$ make mybench2
```

To use libc malloc

```
$ make origbench2
```

What is different?

What are some problems with our implementation?

- We don’t re-use memory so we are calling sbrk more often than is necessary
- Calling sbrk can be slow

A better solution could try re-using memory. How might this work?

Implementing a free list

We need additional book keeping to keep track of the chunks of free memory

```
struct chunk {
  int size;
  struct chunk *next;
};
struct chunk *flist = NULL;
```

In malloc → Check the free list for a chunk we can re-use to satisfy the request. If we can't find one, allocate a new chunk and a corresponding header (struct chunk)

In free → add memory to the free list

New implementation of malloc

Step 1: Try to re-use memory from the free list

> “First fit” strategy – Find the first chunk in the free list that satisfies the user’s request

Step 2: If none is found, ask for more memory with sbrk

New implementation of malloc

```
void *malloc (size_t size) {
  if (size <= 0) return NULL;

  struct chunk *ptr = flist; //flist is global
  struct chunk *prev = NULL;
  while (ptr != NULL) {
    if (ptr->size >= size) {
      if (prev != NULL) {
        prev->next = ptr->next;
      } else {
        flist = ptr->next;
      }
      return (void*) (ptr + 1);
    } else {
      prev = ptr;
      ptr = ptr->next;
    }
  }
  void *memory = sbrk(size + sizeof(struct chunk));
  if (memory == (void *) -1) {
    return NULL;
  } else {
    struct chunk *cnk = (struct chunk*) memory;
    cnk->size = size;
    cnk->next = NULL;
    return (void*) (cnk + 1);
  }
}
```

New implementation of free

```
// flist is a global

void free(void *memory) {
  if (memory != NULL) {
    struct chunk *cnk = (struct chunk*) ((struct chunk*) memory -1);
    cnk->next = flist;
    flist = cnk;
  }
  return;
}
```

Exercise: Trace the behavior of our malloc/free in this program

```
struct test {
    float f;
    char msg[16];
};

void main() {
    struct test*  ptr = (struct test*) malloc(sizeof(test));
    free(ptr);
}
```

How should we choose a block from the freelist?

First Fit: Take the first block that is big enough to satisfy the user's request

Worst Fit: Take the largest block

Best Fit: Take the smallest block that satisfies the request

Exercise:

Assume our allocator uses best-fit to choose a block to re-use from the free list. Visualize the following allocations:

```
char* a = malloc(sizeof(char)*10);
char* b = malloc(sizeof(char)*30);
char* c = malloc(sizeof(char)*20);
free(c);
free(a);
free(b);

char* d = malloc(sizeof(char)*8);
```

Exercise:

Assume our allocator uses worst-fit to choose a block to re-use from the free list. Visualize the following allocations:

```
char* a = malloc(sizeof(char)*10);
char* b = malloc(sizeof(char)*30);
char* c = malloc(sizeof(char)*20);
free(c);
free(a);
free(b);

char* d = malloc(sizeof(char)*8);
```

Exercise:

Assume our allocator uses first-fit to choose a block to re-use from the free list. Visualize the following allocations:

```
char* a = malloc(sizeof(char)*10);
char* b = malloc(sizeof(char)*30);
char* c = malloc(sizeof(char)*20);
free(c);
free(a);
free(b);

char* d = malloc(sizeof(char)*8);
```

Challenges of memory allocation

We use the term fragmentation to refer to wasted memory that cannot be used

internal fragmentation: unusable memory within a chunk.

external fragmentation: Scattered and too-small-to-use chunks of non-contiguous memory between chunks.

Redux: Recall Big fish, Little fish

Can you explain why

- We overwrote the second array with a value of 78, not 76?

- The program crashed in free() and not before?

```
bigfish:
 10  11  12  13  14  15  16  17  18  19
littlefish:
  0   1   2   3   4   5   6   7   8   9

after loop:
bigfish:
 66  67  68  69  70  71  72  73  74  75
littlefish:
 78   1   2   3   4   5   6   7   8   9
Segmentation fault (core dumped)

~
```

```
int main(int argc, char *argv[]) {
   int *bigfish, *littlefish, i;

   // allocate space for two int arrays
   bigfish = (int *)malloc(sizeof(int)*10);
   littlefish = (int *)malloc(sizeof(int)*10);
   if (!bigfish || !littlefish) {
      printf("Error: malloc failed\n");
      exit(1);
   }
   for (i=0; i < 10; i++) {
      bigfish[i] = 10+i;
      littlefish[i] = i;
   }
   print_array(bigfish,10, "bigfish");
   print_array(littlefish,10, "littlefish");

   // here is a bad Heap memory access
   // (write beyond bounds of allocated memory):
   for (i=0; i < 13; i++) {
      bigfish[i] = 66+i;
   }
   printf("\nafter loop:\n");
   print_array(bigfish,10, "bigfish");
   print_array(littlefish,10, "littlefish");

   free(bigfish);
   free(littlefish);  // program will crash here
   return 0;
}
```

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
bigfish array:
 10  11  12  13  14  15  16  17  18  19
littlefish array:
  0   1   2   3   4   5   6   7   8   9

after loop:
bigfish array:
 66  67  68  69  70  71  72  73  74  75
littlefish array:
 78   1   2   3   4   5   6   7   8   9

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e28449 in arena_for_chunk (ptr=0x5555555592c0) at ./malloc/arena.c:156
156     ./malloc/arena.c: No such file or directory.
(gdb) where
#0  0x00007ffff7e28449 in arena_for_chunk (ptr=0x5555555592c0) at ./malloc/arena.c:156
#1  arena_for_chunk (ptr=0x5555555592c0) at ./malloc/arena.c:160
#2  __GI___libc_free (mem=<optimized out>) at ./malloc/malloc.c:3390
#3  0x000055555555533d in main ()
(gdb)
```