CS 50:
Software Design and Implementation

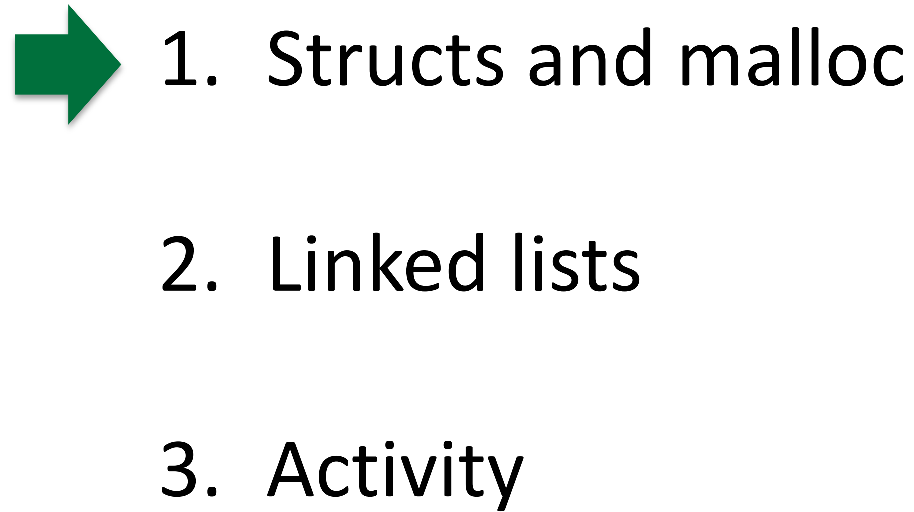
Dynamic memory and linked lists

Big take away message today

Each time you dynamically allocate memory (using malloc or calloc)…

… you must call free!

Agenda



1. Structs and malloc
2. Linked lists
3. Activity

Structs allow multiple variables to be grouped together (but not code!)

students1.c

```
14 #include <stdio.h>
15 #include <string.h>
16
17 #define MAX_NAME_LENGTH 20
18
19 struct student {
20     char name[MAX_NAME_LENGTH];
21     int year;
22 };
23
24 void print_student(struct student s) {
25     printf("%s \'%02d\n", s.name, s.year);
26 }
27
28 int main(int argc, char *argv[]) {
29     struct student alice;
30     struct student bob;
31
32     strncpy(alice.name, "Alice", MAX_NAME_LENGTH);
33     alice.year = 23;
34     print_student(alice);
35
36     strncpy(bob.name, "Bob", MAX_NAME_LENGTH);
37     bob.year = 24;
38     print_student(bob);
39
40     return 0;
41 }
```

Struct students hold name and year for a student

Access struct member with `var.member`

Declare alice and bob as `struct student`

Unlike Java, code cannot be encapsulated into a struct

C is not an OOP language!

Structs allow multiple variables to be grouped together (but not code!)

students1.c

```
14 #include <stdio.h>
15 #include <string.h>
16
17 #define MAX_NAME_LENGTH 20
18
19 struct student {
20     char name[MAX_NAME_LENGTH];
21     int year;
22 };
23
24 void print_student(struct student s) {
25     printf("%s \'%02d\n", s.name, s.year);
26 }
27
28 int main(int argc, char *argv[]) {
29     struct student alice;
30     struct student bob;
31
32     strncpy(alice.name, "Alice", MAX_NAME_LENGTH);
33     alice.year = 23;
34     print_student(alice);
35
36     strncpy(bob.name, "Bob", MAX_NAME_LENGTH);
37     bob.year = 24;
38     print_student(bob);
39
40     return 0;
41 }
```

```
$ mygcc students1.c
$ ./a.out
Alice '23
Bob '24
```

We can create pointers to structs and access members with "->"

students2.c

```
19 #define MAX_NAME_LENGTH 20
20
21 struct student {
22     char name[MAX_NAME_LENGTH];
23     int year;
24 };
25
26 void print_student(struct student s) {
27     printf("%s \'%d\n", s.name, s.year);
28 }
29
30 int main(int argc, char *argv[]) {
31     struct student alice;
32     struct student bob;
33     struct student *charlie;
34
35     strncpy(alice.name, "Alice", MAX_NAME_LENGTH);
36     alice.year = 23;
37     print_student(alice);
38
39     strncpy(bob.name, "Bob", MAX_NAME_LENGTH);
40     bob.year = 24;
41     print_student(bob);
42
43     charlie = malloc(sizeof(struct student));
44     if (charlie == NULL) { return 1; }
45     strncpy(charlie->name, "Charlie", MAX_NAME_LENGTH);
46     charlie->year = 25;
47     print_student(*charlie);
48     free(charlie);
49
50     return 0;
51 }
```

Charlie declared as pointer to a struct student

Charlie is 8 bytes (on a 64-bit system) on the stack ready to hold a memory address

Use `malloc` to allocate space for struct student

Memory is allocated on the heap

Always confirm `malloc` succeeded!

Access members with `var->member`

DO NOT forget to free malloc'd memory!

We can create pointers to structs and access members with “->”

students2.c

```c
19 #define MAX_NAME_LENGTH 20
20
21 struct student {
22     char name[MAX_NAME_LENGTH];
23     int year;
24 };
25
26 void print_student(struct student s) {
27     printf("%s \'%d\n", s.name, s.year);
28 }
29
30 int main(int argc, char *argv[]) {
31     struct student alice;
32     struct student bob;
33     struct student *charlie;
34
35     strncpy(alice.name, "Alice", MAX_NAME_LENGTH);
36     alice.year = 23;
37     print_student(alice);
38
39     strncpy(bob.name, "Bob", MAX_NAME_LENGTH);
40     bob.year = 24;
41     print_student(bob);
42
43     charlie = malloc(sizeof(struct student));
44     if (charlie == NULL) { return 1; }
45     strncpy(charlie->name, "Charlie", MAX_NAME_LENGTH);
46     charlie->year = 25;
47     print_student(*charlie);
48     free(charlie);
49
50     return 0;
51 }
```

```
$ mygcc students2.c
$ ./a.out
Alice '23
Bob ‘24
Charlie '25
```

typedef allows us to define our own variable types – a useful shortcut

students3.c

```
20 #define MAX_NAME_LENGTH 20
21
22 typedef struct student {
23     char name[MAX_NAME_LENGTH];
24     int year;
25 } student_t;
26
27 void print_student(student_t s) {
28     printf("%s \'%d\n", s.name, s.year);
29 }
30
31 int main(int argc, char *argv[]) {
32     student_t alice;
33     student_t bob;
34     student_t *charlie;
35
36     strncpy(alice.name, "Alice", MAX_NAME_LENGTH);
37     alice.year = 23;
38     print_student(alice);
39
40     strncpy(bob.name, "Bob", MAX_NAME_LENGTH);
41     bob.year = 24;
42     print_student(bob);
43
44     charlie = malloc(sizeof(struct student));
45     if (charlie == NULL) { return 1; }
46     strncpy(charlie->name, "Charlie", MAX_NAME_LENGTH);
47     charlie->year = 25;
48     print_student(*charlie);
49     free(charlie);
50
51     return 0;
52 }
```

typedef allows us to define our own variable types

Here I define `student_t`

_t is commonly use for "type"

Now we can use `student_t` instead of `struct student`

But we don't have to!

typedef allows us to define our own variable types – a useful shortcut

students3.c

```
20 #define MAX_NAME_LENGTH 20
21
22 typedef struct student {
23     char name[MAX_NAME_LENGTH];
24     int year;
25 } student_t;
26
27 void print_student(student_t s) {
28     printf("%s \'%d\n", s.name, s.year);
29 }
30
31 int main(int argc, char *argv[]) {
32     student_t alice;
33     student_t bob;
34     student_t *charlie;
35
36     strncpy(alice.name, "Alice", MAX_NAME_LENGTH);
37     alice.year = 23;
38     print_student(alice);
39
40     strncpy(bob.name, "Bob", MAX_NAME_LENGTH);
41     bob.year = 24;
42     print_student(bob);
43
44     charlie = malloc(sizeof(struct student));
45     if (charlie == NULL) { return 1; }
46     strncpy(charlie->name, "Charlie", MAX_NAME_LENGTH);
47     charlie->year = 25;
48     print_student(*charlie);
49     free(charlie);
50
51     return 0;
52 }
```

```
$ mygcc students3.c
$ ./a.out
Alice '23
Bob ‘24
Charlie '25
```

malloc allocates heap memory; does not go out of scope with function local vars

students4.c

```
21 #define MAX_NAME_LENGTH 20
22
23 typedef struct student {
24     char name[MAX_NAME_LENGTH];
25     int year;
26 } student_t;
27
28 void print_student(student_t s) {
29     printf("%s \'%d\n", s.name, s.year);
30 }
31
32 student_t *new_student(char name[], int year) {
33     student_t *student = malloc(sizeof(student_t));
34     if (student == NULL) { return NULL; }
35     strcpy(student->name, name);
36     student->year = year;
37     return student;
38 }
39
40 int main(int argc, char *argv[]) {
41     student_t *denise = new_student("Denise",26);
42     if (denise == NULL) { return 1; }
43     print_student(*denise);
44     free(denise);
45
46     return 0;
47 }
```

This function allocates heap memory for a student

Returns a pointer to that memory

Local variable `student` would get popped from stack (go out of scope) when this function ends

Because it is allocated on the heap, `denise` can reference that memory

Note: dereference `denise` to pass a `student_t` to `print_student`

Still need to free memory!

Now outside the function that allocated it!

malloc allocates heap memory; does not go out of scope with function local vars

students4.c

```
21 #define MAX_NAME_LENGTH 20
22
23 typedef struct student {
24     char name[MAX_NAME_LENGTH];
25     int year;
26 } student_t;
27
28 void print_student(student_t s) {
29     printf("%s \'%d\n", s.name, s.year);
30 }
31
32 student_t *new_student(char name[], int year) {
33     student_t *student = malloc(sizeof(student_t));
34     if (student == NULL) { return NULL; }
35     strcpy(student->name, name);
36     student->year = year;
37     return student;
38 }
39
40 int main(int argc, char *argv[]) {
41     student_t *denise = new_student("Denise",26);
42     if (denise == NULL) { return 1; }
43     print_student(*denise);
44     free(denise);
45
46     return 0;
47 }
```

```
$ mygcc students3.c
$ ./a.out
Denise '26
```

Good practices with memory allocation

1. Each call to `malloc` or `calloc` block must be matched with a call to `free`!
2. Check the return value of `malloc` or `calloc` to ensure the allocation succeeded (returns NULL if fails)
3. Initialize memory from `malloc`
4. Set pointer to NULL after calling `free` to avoid errors
5. Do not overuse dynamic memory
   - Use when you do not know the size of memory needed
   - Use when memory will be needed across functions

Agenda

1. Structs and malloc

2. Linked lists

3. Activity

Situation: you need to track the names of multiple students

There are multiple students
Track each student’s name
Names can be of variable length

Possible solutions:

- 2D array
- Array of pointers
- Linked List

freadlinep reads lines of arbitrary length from a file or stdin

readlinep.c

```
15 char *freadlinep(FILE *fp) {
16   // validate the parameter
17   if (fp == NULL || feof(fp)) { return NULL; }
18
19   // allocate buffer big enough for "typical" words/lines
20   int len = 81;
21   char *buf = calloc(len, sizeof(char));
22   if (buf == NULL) { return NULL;   } // out of memory
23
24   // Read characters from file until newline or EOF,
25   // expanding the buffer when needed to hold more.
26   int pos;
27   char c;
28   for (pos = 0; (c = fgetc(fp)) != EOF && (c != '\n'); pos++) {
29     // We need to save buf[pos+1] for the terminating null
30     // and buf[len-1] is the last usable slot,
31     // so if pos+1 is past that slot, we need to grow the buffer.
32     if (pos+1 > len-1) {
33       char *newbuf = realloc(buf, ++len);
34       if (newbuf == NULL) {
35          free(buf);
36          return NULL;         // out of memory
37       } else {
38          buf = newbuf;
39       }
40     }
41     buf[pos] = c;
42   }
43
44   if (pos == 0 && c == EOF) {
45     // no characters were read and we reached EOF
46     free(buf);
47     return NULL;
48   } else {
49     // pos characters were read into buf[0]..buf[pos-1].
50     buf[pos] = '\0'; // terminate string
51     return buf;
```

Initially allocate 81 characters on heap (calloc sets all bytes to zero)
Check memory was successfully allocated

Read one character at a line until:
- **End of line ('\n')**
- **End of file (EOF)**
- **80 characters read**

If more than 80 chars, reallocate for more memory and keep reading

Save character read

Return NULL for empty file

Otherwise return buffer on heap of characters read

freadlinep reads lines of arbitrary length from a file or stdin

readlinep.c

```
15 char *freadlinep(FILE *fp) {
16   // validate the parameter
17   if (fp == NULL || feof(fp)) { return NULL; }
18
19   // allocate buffer big enough for "typical" words/lines
20   int len = 81;
21   char *buf = calloc(len, sizeof(char));
22   if (buf == NULL) { return NULL;   } // out of memory
23
24   // Read characters from file until newline or EOF,
25   // expanding the buffer when needed to hold more.
26   int pos;
27   char
28   for
29     //
30     //
31     //
32     if
33
34
35
36
37
38
39
40     }
41     bu
42   }
43
44   if (pos == 0 && c == EOF) {
45     // no characters were read and we reached EOF
46     free(buf);
47     return NULL;
48   } else {
49     // pos characters were read into buf[0]..buf[pos-1].
50     buf[pos] = '\0'; // terminate string
51     return buf;
```

readlinep calls freadlinep with stdin

```
static inline char *readlinep(void) { return freadlinep(stdin); }
```

One solution is a 2D array, but there are two primary shortcomings

names2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "freadline.h"

int main(const int argc, char *argv[]) {
  const int maxNames = 100;  // maximum number of names
  const int maxLength = 50;  // maximum length of a name (minus 2)
  char names[maxNames][maxLength];
  FILE *fp;                   // input file
  int n = 0;                  // number of names read

  // validate arguments and open input file
  if (argc == 1) {
    fp = stdin;
  }
  else if (argc == 2) {
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
      fprintf(stderr, "%s cannot open file '%s'\n", argv[0], argv[1]);
      exit (2);
    }
  }
  else {
    fprintf(stderr, "usage: %s [filename]\n", argv[0]);
    exit (1);
  }

  // read the list of names
  for (n = 0; n < maxNames && !feof(fp); ) {
    if (freadline(fp, names[n], maxLength)) {
      n++; // only increment if no error
    }
  }

  fclose(fp);

  printf("%d names\n", n);
  // print the list of names; note each name ends in newline.
  for (int i = 0; i < n; i++) {
    printf("%d: %s", i, names[i]);
```

Names is a 2D array, one row for each student

Each row is an array of characters of maxLength

Read from file (argv[1] or stdin until EOF

Use Control-D for EOF in stdin

One solution is a 2D array, but there are two primary shortcomings

names2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "freadline.h"

int main(const int argc, char *argv[]) {
  const int maxNames = 100;  // maximum number of names
  const int maxLength = 50;  // maximum length of a name (minus 2)
  char names[maxNames][maxLength];
  FILE *fp;                   // input file
  int n = 0;                  // number of names read

  // validate arguments and open input file
  if (argc == 1) {
    fp = stdin;
  }
  else if (argc == 2) {
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
      fprintf(stderr, "%s cannot open file '%s'\n", argv[0], argv[1]);
      exit (2);
    }
  }
  else {
    fprintf(stderr, "usage: %s [filename]\n", argv[0]);
    exit (1);
  }

  // read the list of names
  for (n = 0; n < maxNames && !feof(fp); ) {
    if (freadline(fp, names[n], maxLength)) {
      n++; // only increment if no error
    }
  }

  fclose(fp);

  printf("%d names\n", n);
  // print the list of names; note each name ends in newline.
  for (int i = 0; i < n; i++) {
    printf("%d: %s", i, names[i]);
```

```
$ mygcc -o names names2.c freadline.c
$ ./names
```

Issues:

- **What if names are longer than maxLength?**
- **What if more students than maxNames?**

Use freadline to buffer names[i] names up to maxLength characters

Close file pointer (not needed for stdin but doesn't hurt)

Print array of student names

Another idea is to create an array of pointers, but it still has a shortcoming

names3.c

```
#include <stdio.h>
#include <stdlib.h>
#include "readlinep.h"

int main()
{
  const int maxNames = 100;  // maximum number of names
  char *names[maxNames];      // array of names, each a pointer to string
  int n = 0;                  // number of names read

  // read the list of names
  for (n = 0; n < maxNames && !feof(stdin); ) {
    char *name = readlinep();
    if (name != NULL) {
      names[n] = name;
      n++; // only increment if no error
    }
  }

  printf("%d names\n", n);
  // print the list of names, and free as we go
  for (int i = 0; i < n; i++) {
    printf("%d: %s\n", i, names[i]);
    free(names[i]);
  }
}
```

Read up to 100 names, but each name is a pointer to a string

Each name can now be variable length

Use readlinep to get variable length name

If call `calloc`, (which readlinep does) must remember to free memory on heap!

Issues:

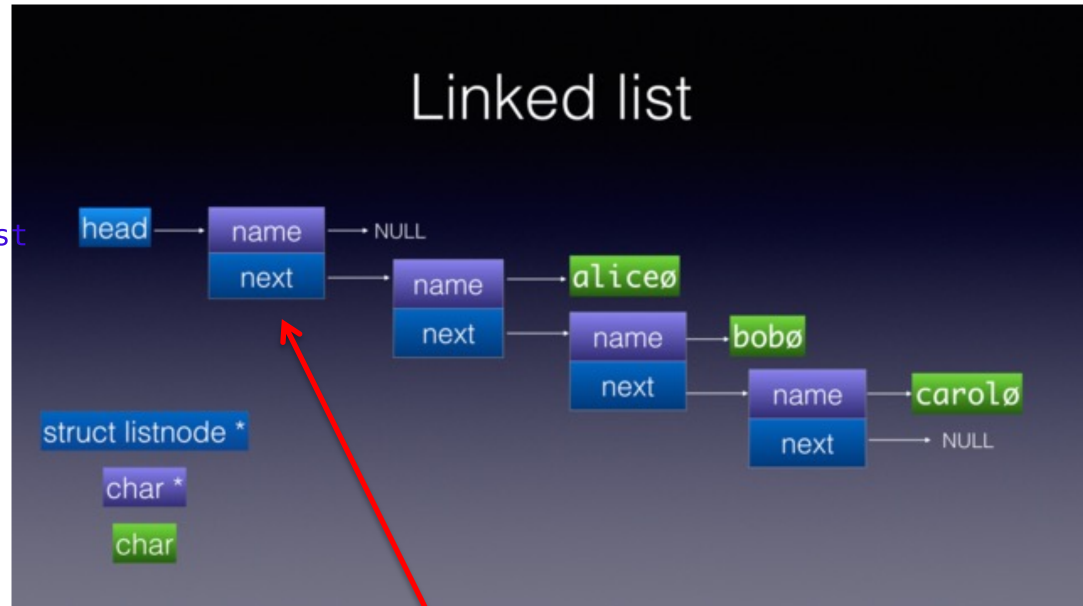
- **What if more students than maxNames?**

A better way is to create a linked list using a struct to hold name and next pointer

names5.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "readlinep.h"


// A structure for each node in linked list
struct listnode {
  char *name;
  struct listnode *next;
};
```



Struct holds pointer to string for name

String can be of variable length

Also holds pointer to next node

In this implementation head is a struct, not a pointer to a struct (does not need to be freed)

Then name member is set to NULL

Each node has name and next pointer

Read variable length names from stdin and insert a new node into the linked list

names5.c

```
int main() {
  struct listnode head = {NULL, NULL}; // dummy node at head of empty list
  int n = 0;                           // number of names read


  // read the list of names - any number of names!
  while (!feof(stdin)) {
    char *name = readlinep();
    if (name != NULL) {
      if (list_insert(&head, name)) {
        n++; // only increment if no error
      }
    }
  }


  printf("%d names:\n", n);
  // print the list of names
  for (struct listnode *node = head.next; node != NULL; node = node->next)
    printf("%s\n", node->name);

  // here we are lazy and do not free the list.
```

Head is a local variable on the stack, not a pointer

Call `readlinep` to get pointer to string (remember uses `calloc`, so we must free the heap space when we are done with it)

Call `list_insert` (next slide) to add to front of list

listnode_new creates a new node, list_insert adds to front of list

names5.c

```
/* *********************************************
 * list_insert: insert the given name into the list.
 * Return true if success, false if failure.
 */
bool list_insert(struct listnode *headp, char *name) {
  struct listnode *node = listnode_new(name);
  if (headp == NULL || node == NULL) {
    return false;
  } else {
    // insert the new node at head of the list
    node->next = headp->next;
    headp->next = node;
  }
  return true;
}
```

list_insert uses listnode_new to add a new node to front of list pointed by by headp

Get new node

Check for NULLs

Add to font of list

Returns true if successful false otherwise

```
/* *********************************************
 * listnode_new: create a new node to store the given name.
 * Returns pointer to new node, if successful, else returns NULL.
 * The pointer 'name' is assumed to be malloc storage, and is not copied.
 * Caller is responsibe for later deleting 'name'.
struct listnode *listnode_new(char *name) {
  // allocate memory for the new node
  struct listnode *node = malloc(sizeof(struct listnode));
  if (node == NULL) {
    return NULL;
  } else {
    // initialize node contents
    node->next = NULL;
    node->name = name;
  }
  return node;
}
```

listnode_new creates and returns a pointer to a new listnode

malloc reserves sizeof(struct listnode) bytes on heap, caller must free

Always check malloc/calloc succeed

Set new nodes name to string passed in and next to NULL

Return node

Output is in reverse order of items input

names5.c

```
int main() {
  struct listnode head = {NULL, NULL}; // dummy node at head of empty list
  int n = 0;                           // number of names read

  // read the list of names - any number of names!
  while (!feof(stdin)) {
    char *name = readlinep();
    if (name != NULL) {
      if (list_insert(&head, name)) {
        n++; // only increment if no error
      }
    }
  }

  printf("%d names:\n", n);
  // print the list of names
  for (struct listnode *node = head.next; node != NULL; node = node->next)
    printf("%s\n", node->name);

  // here we are lazy and do not free the list.
```

Notice: we did not free the memory allocated by malloc/calloc
That is our activity today

I prefer a while loop instead of a for loop when the number of items is not known

Use `while (node != NULL)`

Solution solves the previous issues

- **What if names are longer than maxLength?**
- **What if more students than maxNames?**

```
$ mygcc -o names names5.c readlinep.c
$ ./names
alice
bob
charlie
3 names:
charlie
bob
alice
```

Output in reverse

Why?

New items added to head

Agenda

1. Structs and malloc

2. Linked lists

3. Activity

Bonus material

Local arrays are allocated as a block of contiguous memory on the stack

```
15 #include<stdio.h>
16 #include<stdlib.h>
17
18 const int SIZE = 10; //allocated in global memory (not stack)
19
20 int main() {
21    int arr[SIZE];
22
23    //print array on stack
24    printf("arr at location %p\n",(void *)&arr);
25    for (int i = 0; i < SIZE; i++) {
26       arr[i] = i * 2;
27       printf("value %d at %p\n",arr[i],(void *)&arr[i]);
28    }
29    printf("\n");
30
```

Global variable allocated in global memory

Local array allocated on stack

ptr_array.c

```
$ mygcc ptr_array.c
$ ./a.out
arr at location 0x7fffca695020
value 0 at 0x7fffca695020
value 2 at 0x7fffca695024
value 4 at 0x7fffca695028
value 6 at 0x7fffca69502c
value 8 at 0x7fffca695030
value 10 at 0x7fffca695034
value 12 at 0x7fffca695038
value 14 at 0x7fffca69503c
value 16 at 0x7fffca695040
value 18 at 0x7fffca695044
```

Stack memory

arr is the start of a block of contiguous memory on the stack

arr is the memory address of first element in the array

Each array element address is 4 bytes larger than the previous

So, the array starts at the bottom of the stack. This can lead to buffer overflow attacks – see CS55!)

malloc allocates a block of contiguous memory on the stack

```
15 #include<stdio.h>
16 #include<stdlib.h>
17
18 const int SIZE = 10; //allocated in global memory (not stack)
19
20 int main() {
21    int arr[SIZE];
22
23    //print array on stack
24    printf("arr at location %p\n",(void *)&arr);
25    for (int i = 0; i < SIZE; i++) {
26       arr[i] = i * 2;
27       printf("value %d at %p\n",arr[i],(void *)&arr[i]);
28    }
29    printf("\n");
30
31    int *ptr_arr = (int *)malloc(SIZE*sizeof(int));
32    if (ptr_arr == NULL) {
33       perror("Memory was not allocated!");
34       return 1;
35    }
36
37    //print the pointer to allocated memory on heap
38    printf("ptr_arr at location %p\n",(void *)ptr_arr);
39    int *p = ptr_arr;
40    for (int i = 0; i < SIZE; i++, p++) {
41       *p = i;
42       printf("value %d at %p\n",*p, (void *)p);
43    }
44    printf("\n");
45    free(ptr_arr); //DON'T FORGET!!!!
46
47
```

malloc allocates on heap (not initialized)

Number of bytes is SIZE * 4 = 40

Check that memory allocation successful

Defense: malloc returns NULL if not

ptr_array.c

```
$ mygcc ptr_array.c
$ ./a.out
arr at location 0x7fffca695020
value 0 at 0x7fffca695020
value 2 at 0x7fffca695024
value 4 at 0x7fffca695028
value 6 at 0x7fffca69502c
value 8 at 0x7fffca695030
value 10 at 0x7fffca695034
value 12 at 0x7fffca695038
value 14 at 0x7fffca69503c
value 16 at 0x7fffca695040
value 18 at 0x7fffca695044

ptr_arr at location 0x5631afe2d260
value 0 at 0x5631afe2d260
value 1 at 0x5631afe2d264
value 2 at 0x5631afe2d268
value 3 at 0x5631afe2d26c
value 4 at 0x5631afe2d270
value 5 at 0x5631afe2d274
value 6 at 0x5631afe2d278
value 7 at 0x5631afe2d27c
value 8 at 0x5631afe2d280
value 9 at 0x5631afe2d284
```

Stack memory

Heap memory

ptr_arr is also the start of a contiguous block of memory, but here on the heap

p is incremented by 4 (ints are 4 bytes) each time through loop

If you call malloc, you MUST call free!