Review

What is the function, or execution, stack?

What is a pointer?

What is the difference between static and dynamic allocation of memory?

What are two ways we can allocation 2D arrays dynamically?

Why use pointers?

- Change variables in functions so they persist when the function returns
- Dynamically allocate space for data
- Pass variables without copying
- Create linked data structures
- Interpret memory in different ways

Agenda

Pointer operators: *, &

Pass by pointer for non-array types

Struct pointers

Pointer safety

- NULL pointers
- Return values

Tools for C programming: C Tutor, GDB, Valgrind

Pointer operators: &, *

&, address operator, gives the address (its location in memory) of a variable

*, dereference operator, refers to the value of a pointer

```c
#include <stdio.h>

int main() {
  char c1 = 'a';
  char* ptr;
  ptr = &c1;
  char c2 = *ptr;
  printf("%c %c\n", c1, c2);
  return 0;
}
```

Pointer syntax

<data_type>*: declares a pointer type (LHS syntax)

*<variable>: Dereference a pointer to get the corresponding value at its address (RHS/LHS syntax)

&<variable>: address operator. Gets the address where this variable is stored (RHS syntax)

Pointers

Be careful: * has two meanings!

```
void pass_by_pointer(int* v) {
  *v = 10;
}
```

Exercise: Draw a stack diagram for this function

```
int main() {
   int *ptr1, *ptr2, x, y;
   x = 8;
   ptr2 = &x;
   ptr1 = NULL;
   *ptr2 = 10;
   y = *ptr2 + 3;
   ptr1 = ptr2;
   *ptr1 = 100;
   ptr1 = &y;
   *ptr1 = 80;
   // draw stack here
}
```

Pointers and Functions Revisited

Two choices for passing data

- Pass by value: copies values into parameters of a function (simple types)

- Pass by pointer: Copies the address of data. Changes to the data persist when the function returns (all arrays and pointers)

Example: pass by pointer for non-array types

```
void pass_by_value(int v) {
  v = 10;
}

void pass_by_pointer(int* v) {
  *v = 10;
}

int main() {
  int changeme = 0;
  pass_by_value(changeme);
  printf("The value is: %d\n", changeme);

  pass_by_pointer(&changeme);
  printf("The value is: %d\n", changeme);
  return 0;
}
```

What is the syntax for pass by pointer?

Pass by pointer: draw stack

```
void pass_by_value(int v) {
  v = 10;
}

void pass_by_pointer(int* v) {
  *v = 10;
}

int main() {
  int changeme = 0;
  pass_by_value(changeme);
  printf("The value is: %d\n", changeme);

  pass_by_pointer(&changeme);
  printf("The value is: %d\n", changeme);
  return 0;
}
```

Example: Pass by pointer

```c
#include <stdio.h>

void inc(int* i) {
  int local = *i;
  local = local + 1;
  *i = local;
}

int main() {
  int i = 5;
  printf("Before: %d\n", i);
  inc(&i);
  printf("After: %d\n", i);
  return 0;
}
```

Exercise: rand_point.c

Write a program that uses pass by pointer to set the values of x, y, and z to random positive integers in range [0, 10)

Struct pointers

Works like any other data type!

BUT use -> to access data inside the struct

Rule:

- Use -> for struct pointers
- Use . For struct values

Recall: struct pass by value

```
struct studentT {
   char name[8];
   float gpa;
};
```

```
void changeName(struct studentT s, char* newName) {
  if (newName == NULL) {
    return;
  }
  strcpy(s.name, newName);
}

void changeGpa(struct studentT s, float newGpa) {
  s.gpa = newGpa;
}
```

```
int main() {
   struct studentT student1;

   strcpy(student1.name, "Ruth");
   student1.gpa = 3.5;

   print("ORIGINAL", student1);
   changeName(student1, "Kwame");
   print("AFTER CHANGE NAME", student1);

   changeGpa(student1, 3.7);
   print("AFTER CHANGE GPA", student1);

   return 0;
}
```

What is the output of this program?
Draw the stack diagram.

Draw the function stack: struct pass by value

struct pass by pointer

```
void changeName(struct studentT* s, char* newName) {
  if (newName == NULL) {
    return;
  }
  strcpy(s->name, newName);
  // DRAW STACK HERE
}

void changeGpa(struct studentT* s, float newGpa) {
  s->gpa = newGpa;
}
```

What syntax is different from the previous program?
What is the output of this program?

```
int main() {
  struct studentT student1;

  strcpy(student1.name, "Ruth");
  student1.gpa = 3.5;

  print("ORIGINAL", student1);
  changeName(&student1, "Kwame");
  print("AFTER CHANGE NAME", student1);

  changeGpa(&student1, 3.7);
  print("AFTER CHANGE GPA", student1);

  return 0;
}
```

Draw the function stack: struct pass by pointer

Compiles or not?

```
void changeName(struct studentT s, char* newName) {
  if (newName == NULL) {
    return;
  }
  strcpy(s->name, newName);
}
```

Compiles or not?

```
void changeName(struct studentT* s, char* newName) {
  if (newName == NULL) {
    return;
  }
  strcpy(s->name, newName);
}
```

Compiles or not?

```
void changeName(struct student* s, float value) {
   s->gpa = value;
}
```

```
void changeName(struct student* s, float value) {
   s.gpa = value;
}
```

```
void changeName(struct student* s, float value) {
   s->gpa = &value;
}
```

Pointer safety

Using an invalid pointer will crash your program

Common pointer mistakes

- Using an uninitialized pointer
  - Solution: initialize pointers to NULL
- Dereferencing a NULL pointer
  - Solution: Check for NULL before using
- Freeing memory that has already been freed
  - Solution: Reset memory to NULL after free
- Dereferencing a pointer that refers to deleted or invalid memory
- Casting a pointer to a type that doesn't match the data (more later)

Use **GDB** and **Valgrind** to help you find memory problems in your code

Example: Checking for NULL

```
int main() {
  int* value = NULL;
  if (value != NULL) {
    printf("value is %d\n", *value);
  }

  int a = 4;
  value = &a;

  if (value != NULL) {
    printf("value is %d\n", *value);
  }
}
```

Return values

When we return a value, the *value* is copied

When we return a pointer, the *address* is copied

If the address refers to memory that is destroyed when the function stack pops, the address will be invalid

NEVER return an address to a local variable

Return value gotchas: Safe or not?

```
char* code(int v) {
  char msg[16];
  if (v == 0) strcpy(msg, "val0");
  else if (v == 1) strcpy(msg, "val1");
  else if (v == 2) strcpy(msg, "val2");
  return msg;
}

int main() {
  srand(time(0));
  int val = rand() % 3;
  char* printme = code(val);
  printf("%s\n", printme);
  return 0;
}
```

Draw the function stack/heap. Assume val = 1

Is this return value safe or not?

Return value gotchas: Safe or not?

```
char* code(int v) {
  if (v == 0) return "val0";
  else if (v == 1) return "val1";
  return "val2";
}

int main() {
  srand(time(0));
  int val = rand() % 3;
  printf("%s\n", code(val));
  return 0;
}
```

Draw the function stack/heap. Assume val = 1
Is this return value safe or not?

Return value gotchas: Safe or not?

```
char* code(int v) {
  char* msg = malloc(sizeof(char) * 16);
  if (v == 0) strcpy(msg, "val0");
  else if (v == 1) strcpy(msg, "val1");
  else if (v == 2) strcpy(msg, "val2");
  return msg;
}

int main() {
  srand(time(0));
  int val = rand() % 3;
  char* str = code(val);
  printf("%s\n", str);
  return 0;
}
```

Draw the function stack/heap. Assume val = 1
Is this return value safe or not?
What about memory leaks?

Return value gotchas: Safe or not?

```
void code(int v, char* msg) {
  if (v == 0) strcpy(msg, "val0");
  else if (v == 1) strcpy(msg, "val1");
  else if (v == 2) strcpy(msg, "val2");
}

int main() {
  srand(time(0));
  int val = rand() % 3;

  char msg[16];
  code(val, msg);
  printf("%s\n", msg);
  return 0;
}
```

Tools for C programming

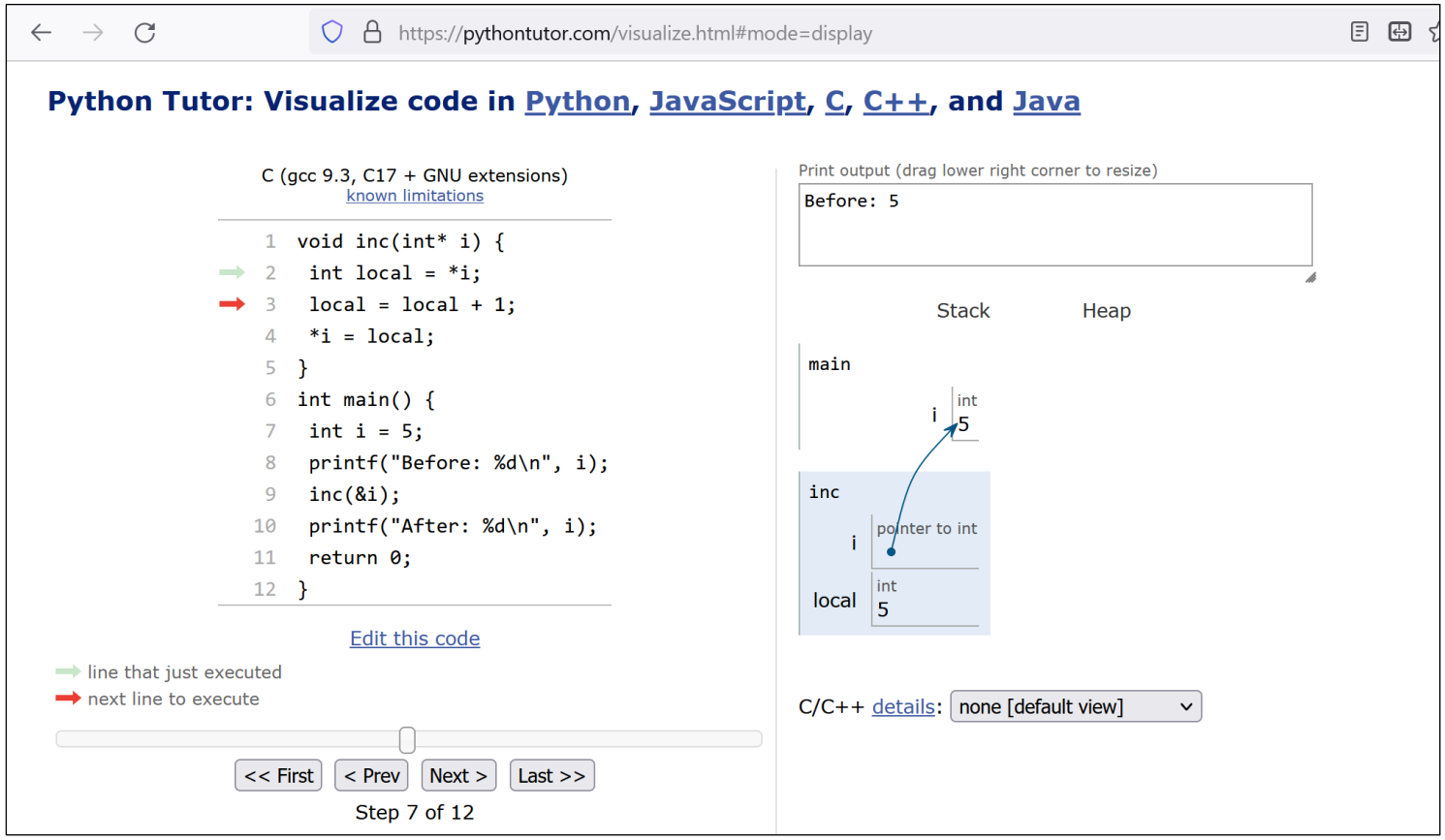
C Tutor

GDB

Valgrind

C Tutor

- https://pythontutor.com/c.html#mode=edit



GDB: GNU Debugger

run (r): Start a program and step through it line by line

break (b): Pause the execution of a program when it reaches certain points in its code

break if: Pause the execution of a program on user-specified conditions

print (p): Show the values of variables at the point in execution that a program is paused

continue (c): Continue a program's execution after a pause

where: Examine the program's execution state at the point when it crashes

up/down: Examine the contents of any stack frame on the call stack

Generating debugger symbols

The debugger needs information about the variable and function names in order to present the execution of the program in a human readable way

The **–g** option does this

gcc –g <filename>.c

Breakpoints

A **breakpoint** is a point in the program where the debugger should pause execution

b <function_name>

b <line_num>

b <filename>:<line_num>

Valgrind

Can help us find common memory errors!

- Reading/writing at unallocated memory (Index out of bounds)
- Freeing already freed memory
- Memory leaks
- Writing to a null pointer

Danger zone: Reading from uninitialized memory

```
int main() {
  int *ptr, x;
  ptr = malloc(sizeof(int) * 10);
  x = ptr[3]; // bad!
  return 0;
}
```

```
int main() {
  int *ptr, x;
  ptr = malloc(sizeof(int) * 10);
  x = ptr[3]; // bad!
  printf("%d\n", x);
  return 0;
}
```

NOTE: Your program might run without crashing even if there are errors in it!

Danger zone: Reading/writing at unallocated memory

```
int main() {
  int values[10];
  printf("%d\n", values[11]);
  values[11] = 5; // memory not allocated!
}
```

Danger zone: Freeing already freed memory

```
int main() {
  int* values = malloc(sizeof(int) * 10);
  free(values);
  free(values);
}
```

Danger zone?: Freeing already freed memory

```
int main() {
  int* values = malloc(sizeof(int) * 10);
  free(values);
  values = NULL;
  free(values);
}
```

Memory leaks: Use valgrind to find

```
int main() {
  int* ptr;
  ptr = malloc(sizeof(int) * 10);
  ptr = malloc(sizeof(int) * 5);
  return 0;
}
```

Invalid memory read example: bigfish.c

```
// allocate space for two int arrays
bigfish = (int *)malloc(sizeof(int)*10);
littlefish = (int *)malloc(sizeof(int)*10);
for (i=0; i < 10; i++) {
   bigfish[i] = 10+i;
   littlefish[i] = i;
}
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
for (i=0; i < 13; i++) {
   bigfish[i] = 66+i;
}
printf("\nafter loop:\n");
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
```

```
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
Segmentation fault (core dumped)
```

Use gdb/valgrind to find!

Invalid memory read example: bigfish.c

```
// allocate space for two int arrays
bigfish = (int *)malloc(sizeof(int)*10);
littlefish = (int *)malloc(sizeof(int)*10);
for (i=0; i < 10; i++) {
   bigfish[i] = 10+i;
   littlefish[i] = i;
}
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
for (i=0; i < 13; i++) {
   bigfish[i] = 66+i;
}
printf("\nafter loop:\n");
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
```

Example: badprog.c

```
int findAndReturnMax(int *array1, int len, int max) {
   int i;
   if (!array1 || (len <=0) ) {
      return -1;
   }
   max = array1[0];
   for (i=1; i <= len; i++) {
      if (max < array1[i]) {
         max = array1[i];
      }
   }
   return 0;
}
```

```
int main(int argc, char *argv[]) {
   int arr[5] = { 17, 21, 44, 2, 60 };
   int max = arr[0];

   if ( findAndReturnMax(arr, 5, max) != 0 ) {
      printf("strange error\n");
      exit(1);
   }
   printf("max value in the array is %d\n", max);
   return 0;
}
```

What is the output of this program supposed to be?