APSC 160 Review
Part 2

Functions and parameters

Arrays

The thing about the labs...

- Lab 1 starts today! (in-lab)
  - Zoom meeting details for each section (should be) on Piazza
  - Submission via GradeScope is configured
    - Enroll in the course by clicking on the GradeScope link in Canvas
  - Projects are for Visual Studio 2019
    - See main lab page on course website for how to use source files to create your own VS projects in case of incompatibility

- Attend your registered lab section
  - Geoff will discuss with department regarding DTS students

The thing about lectures…

- Geoff posted a broken poll on Piazza regarding lectures on Collaborate Ultra vs Zoom
  - Going to re-do the poll again, in a way which is more fair

The thing about midterms…

- Geoff has been notified of a conflict with the proposed 18:00 start time potentially affecting a significant number of students
    - Will run a Piazza poll to find a time that will minimize such conflicts

Function parameters

- **Actual** parameter
  - Value(s) or variable(s) specified by the function *caller*
- **Formal** parameter
  - Variables found in the signature/header of the *function* itself

- Formal parameters must match with actual parameters in *order*, *number*, and *data type*

Calling functions

What happens when a function is called

1. Copy parameter values/addresses (if any) from caller to function, regardless of variable names
2. Execute the function. Function ends when we reach *any* return statement
3. Pass back the answer (if any) via the return statement
4. Destroy all local variables in the *function*
5. Return control to the caller
6. Finish the rest of the calling statement (after replacing the function call with the return value, if any)

Function parameters

- Parameters may be **passed by value** ("call-by-value")
  - the *value* of the actual parameter is copied to the formal parameter when the function is called
- The actual parameters and formal parameters are *different variables in memory*, even if they are named the same
- If you change the value of the formal parameter, this does **not** affect the value of the actual parameter back in the caller's memory

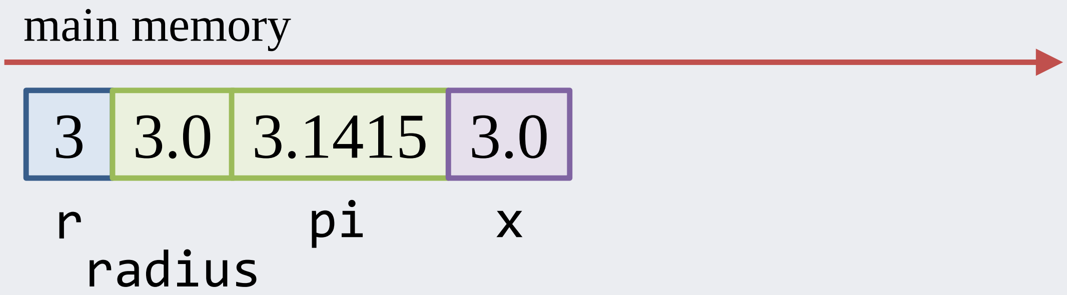
Function parameters and the call stack

Example

```
// ...
int r = 3;
double area = circleArea((double)r);
// ...
```

```
double square(double x){
  return x * x;
}
```

```
double circleArea(double radius){
     double pi = 3.1415;
     double sq_r = square(radius);
     return sq_r * pi;
}
```



Function parameters and the call stack

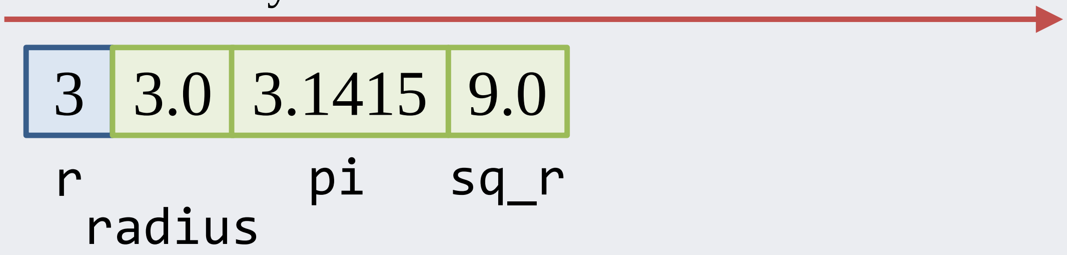
Example

```
// ...
int r = 3;
double area = circleArea((double)r);
// ...
```

```
double square(double x){
  return x * x;
}
```

```
double circleArea(double radius){
     double pi = 3.1415;
     double sq_r = square(radius);
     return sq_r * pi;
}
```

main memory



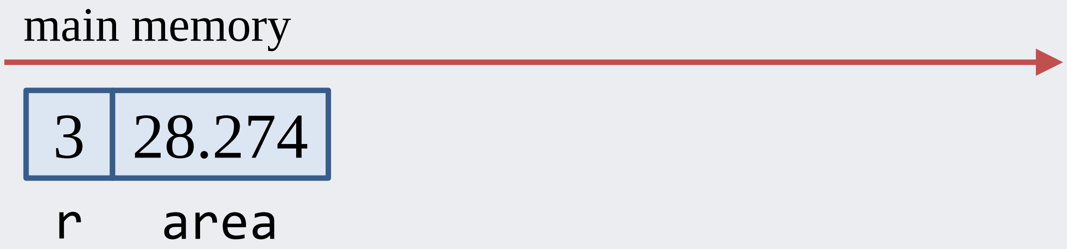
Function parameters and the call stack

Example

```
// ...
int r = 3;
double area = circleArea((double)r);
// ...
```

```
double square(double x){
  return x * x;
}
```

```
double circleArea(double radius){
     double pi = 3.1415;
     double sq_r = square(radius);
     return sq_r * pi;
}
```



main memory

3	28.274
r	area

Functions and parameters

- Consider the following code segment
- Fill in the blanks to show what is output to the screen when the program runs

```
void myFunc(int a, int b) {
  a = a + 4;
  b = b - 4;
  printf("In myFunc a = %d b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;
  myFunc(a, b);
  printf("In main a = %d b = %d\n", a, b);
  return 0;
}
```

```
In myFunc a = ___ b = ___
In main a = ___ b = ___
```

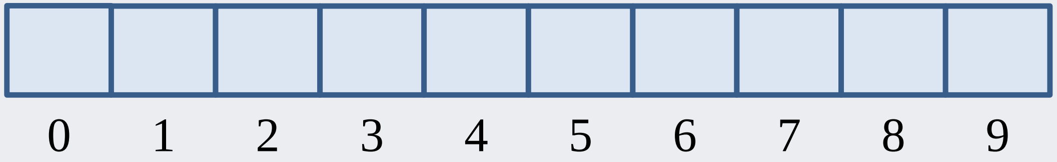
C Arrays

Arrays

- A collection of data elements of the same type
- Stored in consecutive memory locations and each element referenced by an index
- Declared like ordinary variables, followed by [ ] containing the size of the array
- Size must be a constant or a literal integer

```
int age[100];

const int DAYS = 365;
double temperatures[DAYS];
```



Array elements

- Arrays can be initialised directly when declared or by using a loop

```
int fib[] = {0,1,1,2,3,5,8,13}
```

```
int marks[10];
for (int i = 0; i < 10; i++)
  marks[i] = -1;
```

- Values can be assigned from input, or other operation

```
int marks[10];
for (int i = 0; i < 10; i++)
  scanf("%d", &marks[i]);
```

```
for (int i = 0; i < 10; i++)
  arr1[i] = arr1[i] * 3;
```

- Arrays *cannot* be assigned to an existing array

```
int arr1[4];
int arr2[4];
...
arr1 = arr2; // can't do this
arr1 = {1,3,5,7}; // or this
```

Arrays in loops

- Consider the following code segment
  - What is the value of `sum` after the code segment has completed execution?

```
int data[] = {2, 4, 8, 16, 32, 64};
int sum = 0;
int index = 1;

while (index < 4) {
  sum += data[index];
  index++;
}
```

a) `sum = 30`

b) `sum = 60`

c) `sum = 32`

d) `sum = 14`

e) None of the above

Arrays parameters

- An array parameter can be passed like an array variable
  - size is not specified between [ ]
- The array itself does not know its size
  - thus the size is usually passed as an additional variable to prevent out-of-bounds errors
  - e.g. a function prototype and a call to the function:

```
int sum(int arr[], int size) { // prototype
  ...
}
...
int arr1[4];
...
sum(arr1, 4); // function call
```

Array parameters

- Consider the following code function

```
int doSomething(int data[], int size, int someval) {
  int found = -1;
  for (int index = 0; index < size; index++) {
    if (data[index] == someval)
      found = index;
  }
  return found;
}
```

a) It returns true if `someval` is found in the first `size` entries of `data`, and false otherwise

b) If `someval` is contained in the first `size` entries of `data`, it returns the value `someval`, otherwise it returns -1

c) If `someval` is contained in the first `size` entries of `data`, it returns the index of the last slot where `someval` is found, otherwise it returns -1

d) If `someval` is contained in the first `size` entries of `data`, it returns the index of the first slot where `someval` is found, otherwise it returns -1

Array variable details

- An array variable records the address of the first element of the array
  - This address cannot be changed after the array has been declared
  - It is therefore a *constant pointer* (more about pointers later)

- As effects of this:
  - Existing array variables cannot be reassigned
  - Arrays passed to functions can be modified by those functions
    - (unlike the last example from the previous lesson)

Array parameters

- Consider the following code segment
  - Fill the blanks to show what is output to the screen when the program runs

```
#define SIZE 3
void process(int data[]);
int main() {
  int data[SIZE] = {5, -1, 2};
  process(data);
  for (int index = 0; index < SIZE; index++) {
    printf("%d", data[index]);
  }
}

void process(int data[]) {
  for (int index = 0; index < SIZE; index++)
    data[index] = 0;
}
```

Answer: ___ ___ ___

Function parameters

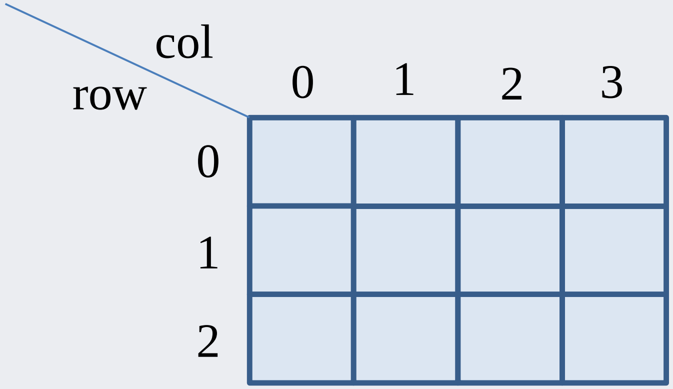
Pass by reference

- In some cases, parameters may be passed by reference ("call-by-reference")
  - The address (rather than the value) of the actual parameter is copied to the formal parameter when the function is called
  - Making a change to the value of the formal parameter effectively changes the value of the actual parameter
  - This is what occurs with array parameters, which are passed by reference by default
- More about this when we get to pointers

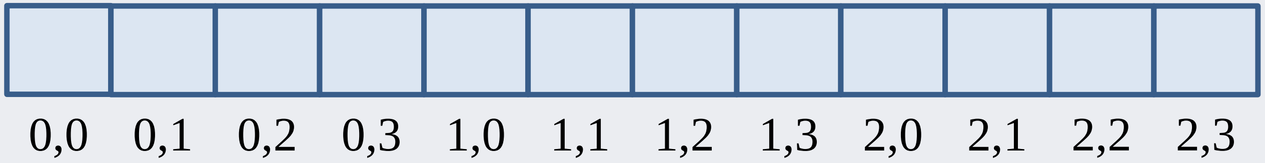
Multi-dimensional arrays

- A two-dimensional array is specified using two indices
  - First index denotes the row, second index denotes column

```
int marks[3][4];
```



- C stores a two-dimensional array contiguously like a 1D array



- Multi-dimensional arrays passed as parameters in same way

```
void myfunction(int data[][NUMCOLS], int numrows);
```

Searching in arrays

- Consider a situation where we have one array containing n integers and another array containing $2n$ integers. The arrays are unordered
  - A number x is randomly located at some (different) position in both arrays
- On average the ratio of the number of operations it takes to locate x in the array of size $2n$ when compared to the array of size n is:

a) The same amount of time is needed

b) Twice as much time is needed to find x in the array of size $2n$

c) Three times as much time is needed to find x in the $2n$ array

d) Four times as much time is needed to find x in the $2n$ array

e) Eight times as much time is needed to find x in the $2n$ array

Pointers

Pointers

Arrays

Dynamic Memory Allocation

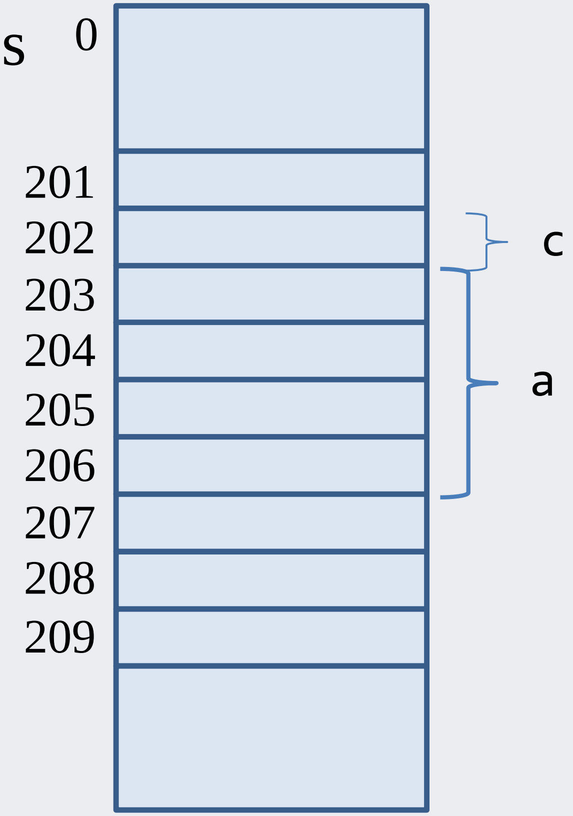
Addresses and pointers

- Every storage location in memory (RAM) has an *address* associated with it
  - The address is the location in memory where a given variable or identifier stores its data
- Can think of address in memory like a mailbox number
  - Use the address to find where the mailbox is
  - Look inside the mailbox to access the contents/value

Variable declaration

- Each byte of memory has a unique address

```
int a;
char c;
a = 5;
a++;
```



- At compile time, the compiler knows how much memory to allocate to each variable (e.g. 4 bytes for `int`, 1 byte for `char`, etc)

Addresses, &, and pointers

- You have already encountered addresses with the `scanf` function
  - `scanf` requires us to provide the address of a location using the "address of" operator, `&`
  - e.g. `scanf("%d", &a)`
  - This allows the `scanf` function to modify the value of the variable `a`, which is defined outside of `scanf`'s call stack

- A pointer is a data type that contains the address of the object in memory, but it is not the object itself

```
int a = 5;
int* p = &a;
```

  - `a` is an integer variable with the value 5
  - `p` is a pointer variable storing the address of `a`

Declaring pointers

- Pointer variables are declared as follows:

  ```
  datatype* identifier
  ```

  - e.g. `int* ptr;` or `int * ptr;` or `int *ptr;`
- Note that the type of a pointer is not the same as the type it points to
  - e.g. `ptr` is a pointer to an `int`, but is itself not an `int`

- Warning! The declaration

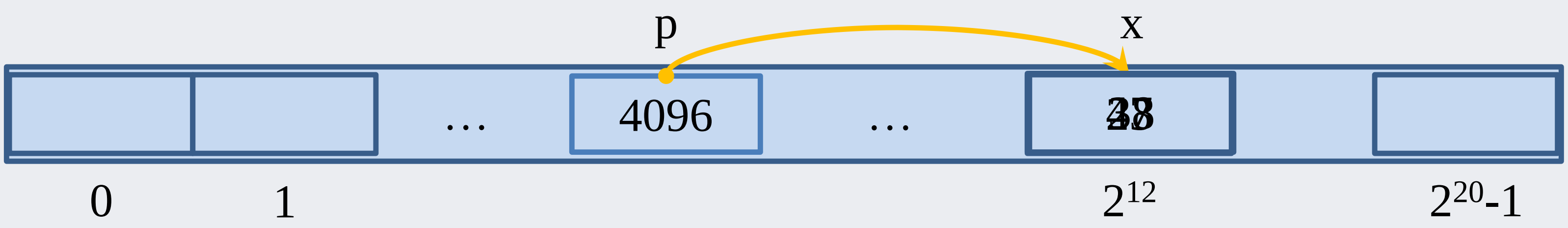
  ```
  int* var1, var2;
  ```

  - declares `var1` as a pointer, but `var2` as an integer!
- To declare both as pointers, either declare individually, or:

  ```
  int *var1, *var2;
  ```

Address operator and dereferencing

- Pointers can be assigned the address of an existing variable
  - Using the address operator, &
- The value which a pointer points to can be accessed by *dereferencing* the pointer
  - Using the * operator



```
int x = 23;
```

```
int* p = &x;
```

```
x = 47;
```

```
*p = 38;
```

Pointers as parameters

- Function parameters can be passed by reference using pointers

```c
int getArraySum(int arr[], int size, int* pcount) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (arr[i] > 0) (*pcount)++;
    sum += arr[i];
  }
  return sum;
}
```

```c
int numpositive = 0;
int numbers[] = {3, 7, -9, 5, -4};
int result = getArraySum(numbers, 5, &numpositive);
printf("Array sum: %d\n", result);
printf("Number of positive elements: %d\n", numpositive);
```

```
Array sum: 2
Number of positive elements: 3
```

Pointers as parameters

- What is out after the code on the right is executed? What is on the call stack for each function call?

```
void f1(int arg)
{
  arg = 22;
  printf("f1 arg: %d\n", arg);
}
```

```
void f2(int* arg)
{
  *arg = 410;
  printf("f2 arg: %d\n", arg);
}
```

```
int x = 45;

f1(x);
printf("x after f1: %d\n", x);

f2(&x);
printf("x after f2: %d\n", x);
```

Readings for this lesson

- Thareja
  - Chapter 3 – Arrays
  - Chapter 1.11 – Pointers

- Lab #1 available on course website