CMPT 125

Arrays and Strings

Arrays and Strings

- Arrays
- Arrays and pointers
- Loops and performance
- Array comparison
- Strings

Arrays

Python List and C Array

- Python
  - a sequence of data
  - access elements with [index]
  - index from [0] to [len-1]
  - dynamic length
  - heterogeneous types
  - has methods

- C
  - a sequence of data
  - access elements with [index]
  - index from [0] to [len-1]
  - fixed length
  - homogeneous types
  - has no methods

Declaring an Array

- Declare an array with its type and []s after the name
  - `int scores[4];`
  - Then set values using an index
  - `scores[0] = 5;`
  - `scores[1] = 3;`
  - `scores[2] = 7;`
  - `scores [3] = 4;`
- Alternatively declare and initialize arrays in one
  - `int scores[4] = {5, 3, 7, 4};`
  - Can only be used on declaration
    - Can not be used to set values on existing arrays

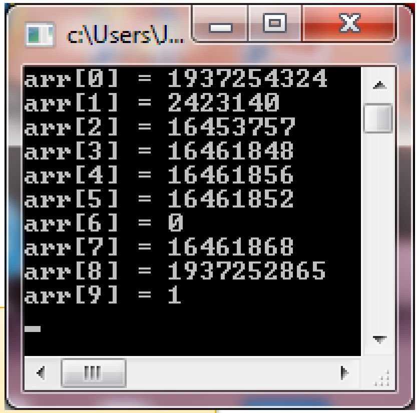
length of the array

Initializing Arrays

- If an array is not initialized it will contain garbage values
  - The bit pattern that happens to be stored in the array elements' memory locations
- The *sizeof* function can be used to find the length of an array
  - But only for static arrays

```
int arr[10];

for(int i = 0; i < sizeof(arr) / sizeof(int); ++i){
        printf("arr[%d] = %d\n", i, arr[i]);
}
```



```
arr[0] = 1937254324
arr[1] = 2423140
arr[2] = 16453757
arr[3] = 16461848
arr[4] = 16461856
arr[5] = 16461852
arr[6] = 0
arr[7] = 16461868
arr[8] = 1937252865
arr[9] = 1
_
```

Array Bounds

- Be careful not to access an array using an index that is out of bounds
  - Less than zero or greater than array size - 1
- Something undesirable will happen
  - It might print garbage
  - Or crash
  - Python would generate a run-time error

```c
int arr[10] = {0,1,2,3,4,5,6,7,8,9};

for(int i = 0; i <= 10; ++i){
       printf("arr[%d] = %d\n", i, arr[i]);
}
```

Arrays and Pointers

Arrays are Pointers

- If an array is passed to a function, changes made to it within the function will persist
  - Because an array variable is a *constant pointer* to the first element of the array
  - So an array parameter actually specifies the address of the array
  - This is still pass by value
    - Just that the value being passed is a pointer

Passing Arrays to Functions

```
int sumArray(int arr[], int size)
{
        int sum = 0;
        for(int i=0; i < size; ++i){
                sum += arr[i];
        }
        return sum;
}
```

the function could also have been written like this

```
int sumArray(int* arr, int size)
{
        int sum = 0;
        for(int i=0; i < size; ++i){
                sum += arr[i];
        }
        return sum;
}
```

Since an array variable is a pointer the two function headers are essentially identical

But the first makes it explicit that an array is passed to the function

More About Pointers

```c
void printArray(int arr[], int size)
{
        int* p;
        for(p = arr; p < arr + size; p++){
               printf("%d\n",*p);
        }
}
```

what's going on here?

this correctly prints an array – why?

dereference *p* to access the value it points to

this seems OK, assign the address of the first element of the array to *p*

Pointer Arithmetic

- The preceding example included this statement
  - `p++; // p is a pointer to an int`
- This is an example of *pointer arithmetic*
  - The statement looks like it should add one to the address that *p* stores
    - Making it point to an address that doesn't match a variable
  - However it does not do this, instead it adds the *size of an int* to the address stored in *p*
    - i.e. 4
- This is another example of operations that behave differently based on operand type

Pointer Arithmetic Example

Pointer arithmetic

```
int arr[] = {1,2,3,4};
int* p = arr;
```

variable	type	address
arr	constant int*	2048
p	int*	2064

symbol table

next free byte

data	1				2				3				4				2048					
address	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	...

call stack

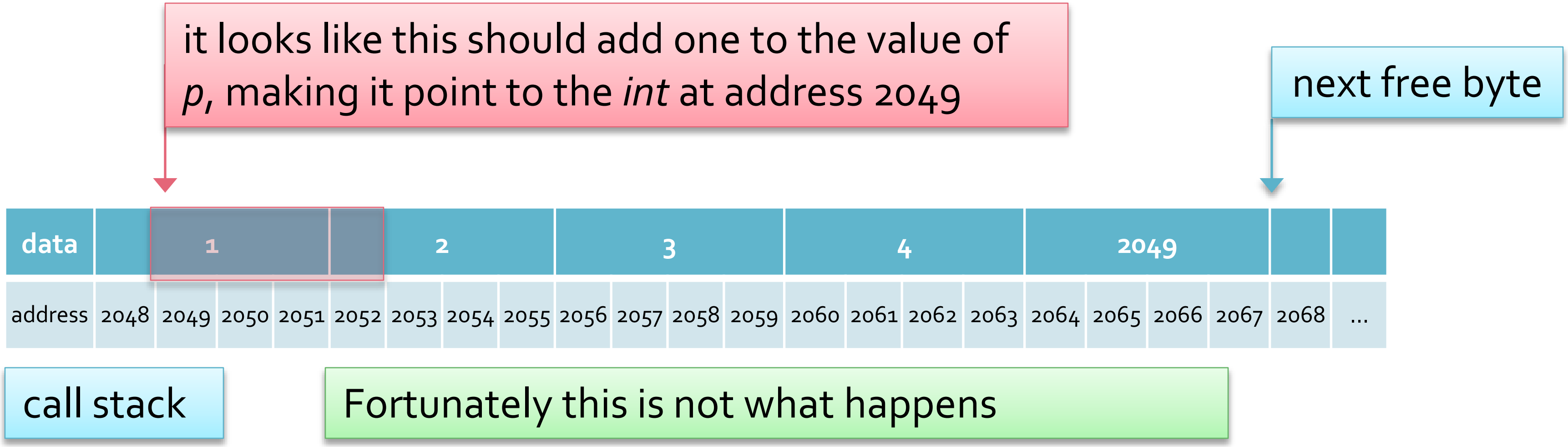
Pointer Arithmetic Example

Pointer arithmetic

```
int arr[] = {1,2,3,4};
int* p = arr;
p++;
```

variable	type	address
arr	constant int*	2048
p	int*	2064

symbol table



it looks like this should add one to the value of *p*, making it point to the *int* at address 2049

next free byte

data		1			2				3				4				2049					
address	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	...

call stack

Fortunately this is not what happens

Pointer Arithmetic Example

Pointer arithmetic

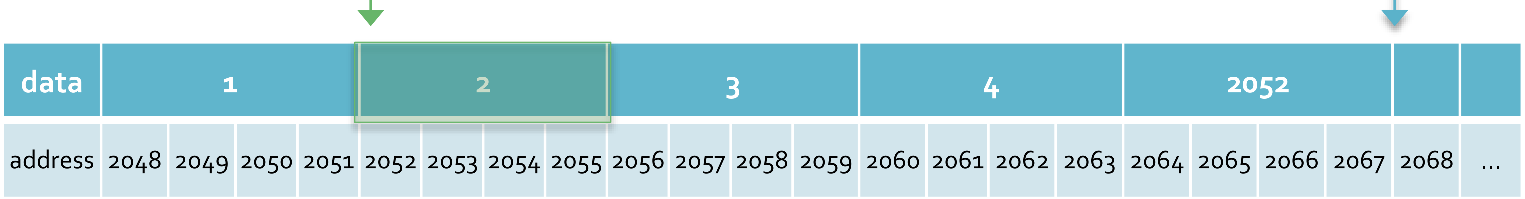
```
int arr[] = {1,2,3,4};
int* p = arr;
p++;
```

variable	type	address
arr	constant int*	2048
p	int*	2064

symbol table

instead, the size of the type of data that *p* points to is added to *p*, making it point to the next *int*

next free byte



data	1				2				3				4				2052					
address	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	…

call stack

Another Sum Function

Here is another example of using pointer arithmetic to iterate through an array

```
int sumArray2(int* start, int* end)
{
        int sum = 0;
        while(start < end){
                sum += *start++;
        }
        return sum;
}
```

the function would be called like this

```
sumArray2(arr2, arr2 + ARR_SIZE)
```

dereferences *start*, adds the value to *sum* then increments *start*

Note that I don't see any real value in writing the function like this, and it is fairly hard to understand unless you have a good knowledge of pointers

Note the interesting precedence rules, 1 (size of an int) is added to start, not to the variable pointed to by start

Pointer Operations

- Assignment, e.g. `p = &x`
  - Pointer type should be compatible with variable
- Dereferencing , e.g. `*p = 12`
  - The *operator accesses the variable that is pointed to
- Arithmetic , e.g. `++p, p += 4, p--`
  - The amount added to or subtracted from the address is multiplied by the size (in bytes) of the variable pointed to
- Differencing , e.g. `int x = p - q`
  - The difference in elements between the pointers
    - i.e. the difference in the addresses of *p* and *q*, divided by the size of the variable pointed to

Valid and Invalid Operations

```
int arr[4];
int* p;
int* q;
```

Valid	Invalid	Notes
`p++;`	`arr++`	*arr* is a *constant* pointer, so the address stored in *arr* cannot be changed
`q = p + 2;`	`p = p + q;`	pointer arithmetic allows integers to be added to pointers, but does not allow pointers to be added together

Loops

Arrays and Iteration

- Python iteration
  - `for i in range (n):`
  - `while <condition>:`
  - break
  - continue

- C iteration
  - `for(int i=0; i < n; i++) {}`
  - `while (<condition>) {}`
  - `do{} while(<condition>);`
  - break
  - continue

The main difference in syntax is in the for loop

For Factorial

```
// Prints the factorials from 1 to n
int main()
{
      long long factorial = 1;
      int n;
      printf("Enter an integer: ");
      scanf("%d", &n);
      for (int i = 1; i <= n; ++i){
            factorial = factorial * i;
            printf("%d! = %lld\n", i, factorial);
      }
}
```

The loop control statement consists of three statements

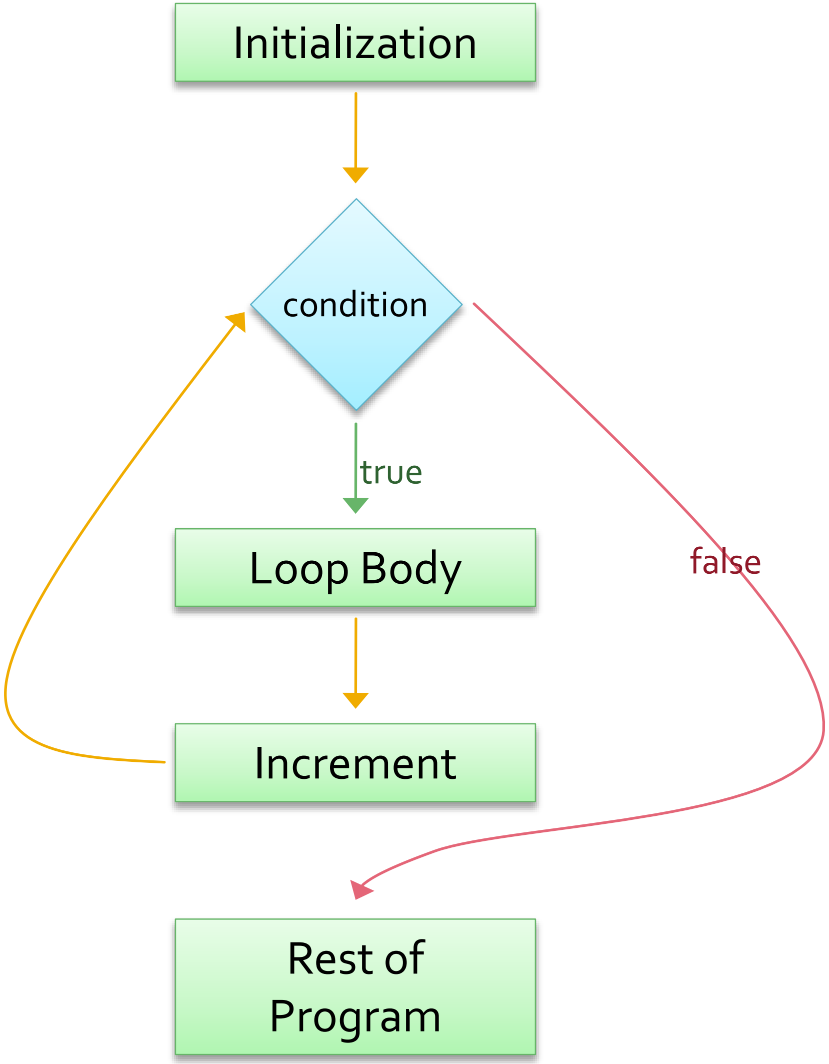
initialization

condition

increment

In this example the loop control variable is also declared in the initialization statement

Controlling For Loops



- `for` statements consist of three expressions
  - Separated by `;`s
- Initialization
  - Executed only once
- Condition
  - Tested before each iteration
    - The last time the condition is tested there is no iteration
    - Since the test returns false
- Increment
  - Applied after each iteration

Common Errors

- Adding an extra semi-colon

empty loop body

```
for (int i = 1; i <= n; ++i); {
    result = result * i;
    printf("%d! = %lld\n", i, factorial);
}
```

- Forgetting opening and closing brackets

```
for (int i = 1; i <= n; ++i)
    result = result * i;
    printf("%d! = %lld\n", i, factorial);
```

not included in loop body

- It is good style to always use brackets even if the loop body only contains one statement

While Loops

- Python

```
def gcd(a, b):
  while b != 0:
    temp = b
    b = a % b
    a = temp
  return a
```

- C

```
int gcd(int a, int b){
  while (b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

While loops in C and Python are very similar

Conditions in C and Python are the same, 0 is treated as false and non-zero as true

Running Time of Loops

- What follows is a few lines of code but it could take a long time to run

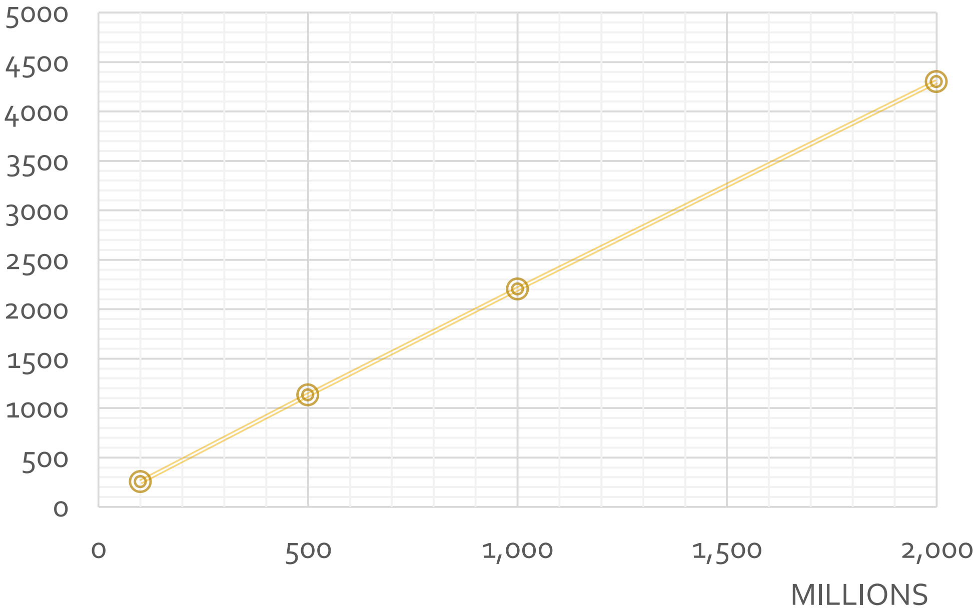
```
long long total = 0;
for (int i = 1; i <= n; i++) {
    total += i;
}
```

  - The running time depends on the value of *n*
- As *n* increases the running time increases
  - If we add one to *n*, then the loop iterates one more time
  - The relationship between *n* and running time is *linear*

Empirical Measurements

- We can time programs using the Linux time command
  - time ./a.out

n	time
100,000,000	253
500,000,000	1,132
1,000,000,000	2,204
2,000,000,000	4,300



Array Comparison

What's Wrong Here?

```
int main ( ) {
    int password[3] = {1, 2, 3};
    int answer[3];

    for (int i = 0; i < 3; i++) {
        printf("Enter digit %d: ", i+1);
        scanf("%d", answer+i);
    }
    if (password != answer) {
        printf("Incorrect password!\n");
    }
}
```

logic error

compares the pointer values (addresses) not the array elements

Array Comparison

- A function to compare two arrays
  - Needs to compare pairs of elements from each array

```
int arrCompare(int arr1[], int arr2[], int length) {
    for (int i = 0; i < length; i++) {
        if (arr1[i] < arr2[i]) {
            return -1;
        } else if (arr1[i] > arr2[i]) {
            return 1;
        }
    }
    return 0;
}
```

Strings

C Strings

- In C a string is just an array of characters
  - A char is a single byte
  - That stores an ASCII code
  - An array of characters forms a string
- The end of the string is marked with a *null character*
  - '\0' or the ASCII code 0
  - So the array must be large enough to hold all of the characters plus one

Characters are Integers

- It’s easy to print the ASCII code for a character
  - `char ch = 'x';`
  - `printf("code for %c = %d", ch, ch);`
- The first placeholder prints the letter that the code represents
- The second placeholder prints the code
- C will also allow arithmetic to be performed on char variables
  - The underlying numeric codes are operated on

Arithmetic and Char

- Let’s say that we want to print all of the letters from A to Z
  - We could write 26 *printf* statements
    - `printf('A');`
    - `printf('B');`
    - `...`
- Or we could do this

```
char ch = 'A';
while(ch < 'A' + 26){
    printf("%c\n", ch);
    ch++;
}
```

Name and Age Program

```c
int main()
{
  char name[20];
  int age;

  printf("What is your name? ");
  scanf("%s", name);
  printf("What is your age? ");
  scanf("%d", &age);

  printf("Your name is %s, and your age is %d\n",
      name , age);
  return 0;
}
```

things to note

its an array

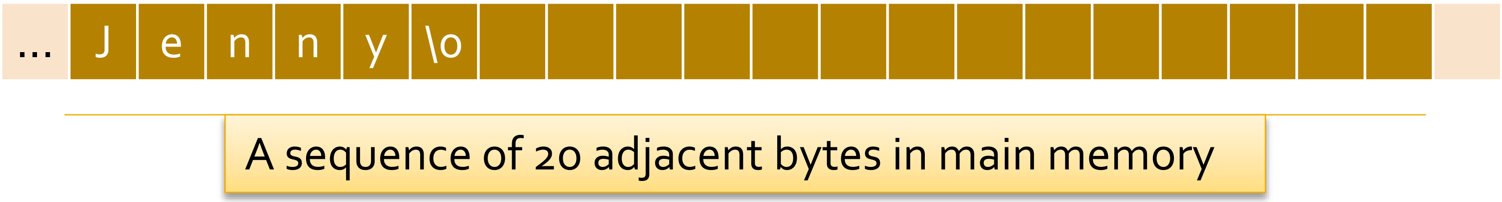
no &

```
What is your name? Jenny
What is your age? 11
Your name is Jenny, and your age is 11
```

Character Arrays

- The line char name[20]; declares an array of 20 characters
  - A sequence in main memory with enough space for twenty characters
  - An array is an ordered sequence of data elements of one type
- The brackets identify *name* as an array rather than a single character
  - And 20 indicates the size of the array

Arrays and Memory



…	J	e	n	n	y	\0															

A sequence of 20 adjacent bytes in main memory

Why 20 bytes?

Because each character is stored in one byte

What 's the \0 in the 6th. byte?

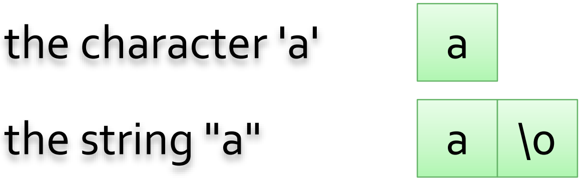
A *null character* to indicate the end of the string

Why is this necessary?

Because memory locations can't be empty, so it is important to distinguish between characters we want to store and garbage values

Characters and Strings

- A string consists of an array containing the words in the string and the null character
- Consequently, a string containing a single character is not the same as a *char* variable



the character 'a'

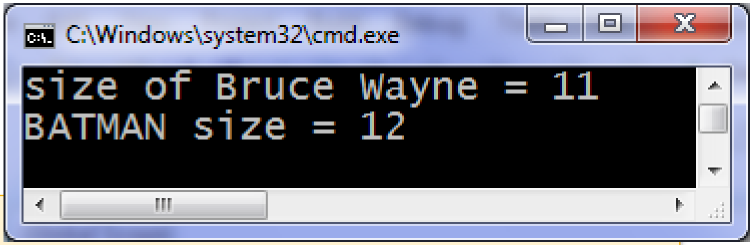
the string "a"

- The type of a character *array* is not the same as the type of a character
  - That is *char name[20]* declares an *array* not a *char*

String Sizes

- What gets printed?

The difference is the null character



```
size of Bruce Wayne = 11
BATMAN size = 12
```

```c
#define BATMAN "Bruce Wayne"
int main()
{
   printf("size of %s = %d\n", BATMAN, strlen(BATMAN));
   printf("BATMAN size = %d\n\n", sizeof(BATMAN));
   return 0;
}
```

String Comparison

```
#include <stdio.h>
#include <string.h>

int main ( ) {
      char password[4] = "abc";
      char answer[4];

      printf("Enter 3 character code: ");
      scanf("%s", answer);

      if (strcmp(password, answer) != 0) {
            printf("Incorrect password!\n");
      }
}
```

string functions

no &

returns 0 if the same; < 0 if first < last; >0 if first > last

Other String Functions

- `void strcpy(char dest[], char src[])`
  - copies source to destination
  - The variable name is *not* preceded by an &
- `void strcat(char dest[], char src[])`
  - appends source to destination
- Actual function header in libraries may differ
  - Both these functions are potentially unsafe
  - Why?