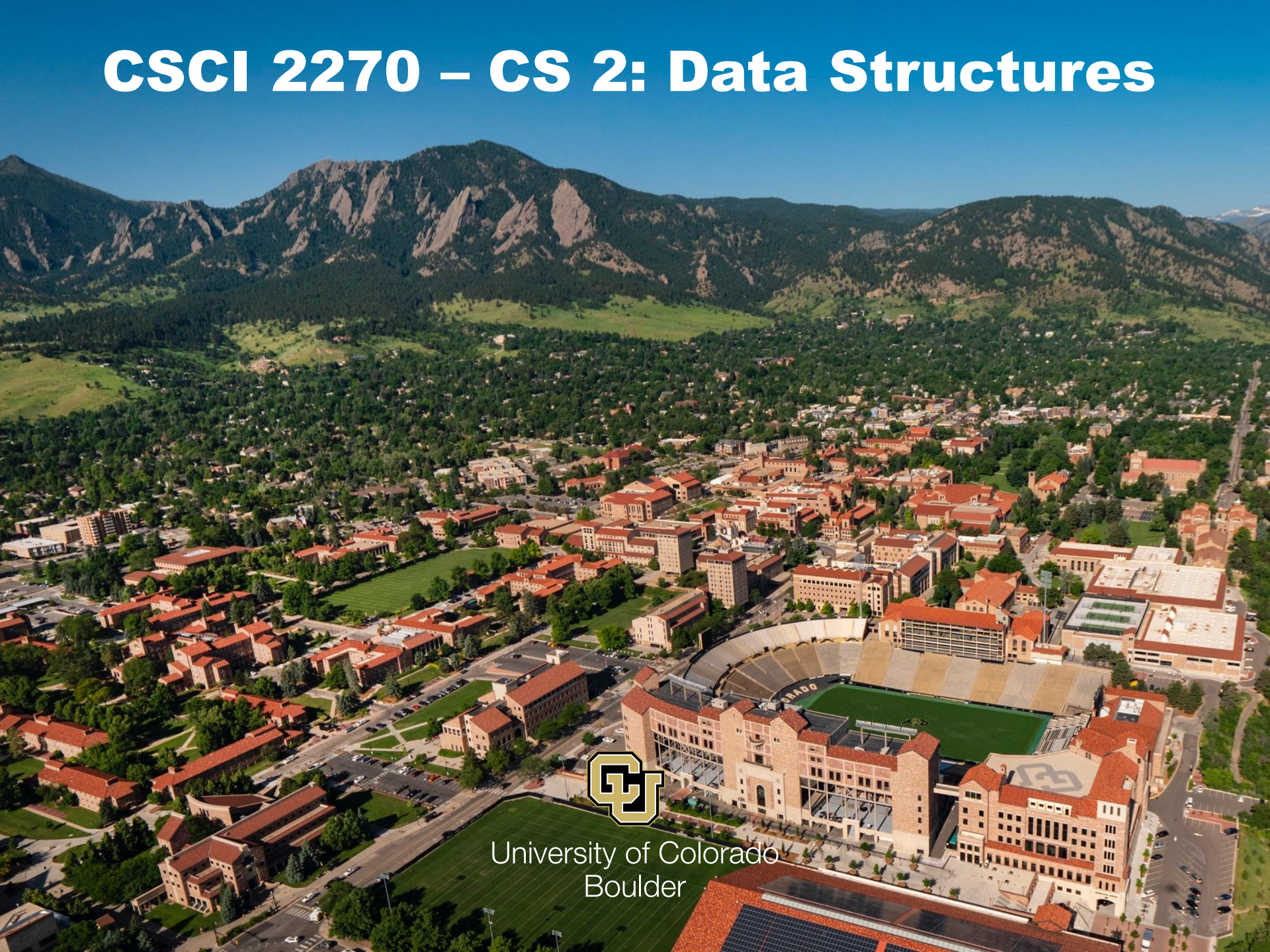
CSCI 2270 – CS 2: Data Structures



University of Colorado
Boulder

Reminders

Topics

- Streams (File I/O)
- Arrays
- Structs
- Command Line Arguments

Streams

- C++ input/output is based on streams.
- To access a file, use a file stream.
- To read or write files, you use variables of type
  - ifstream (input)
  - ofstream (output)
  - fstream (input and output)
- Include <fstream> header

Streams – Opening a Stream

- When opening a file stream, you supply the name of the file stored on disk.

```
in_file.open("input.dat");
```

All streams are objects!

`~/homework/input.dat` (UNIX)
`C:\homework\input.dat` (Windows)

```
in_file.open("C:\\homework\\input.dat");
```

Streams – Opening a Stream

```
e.g. 1
cout << “Please enter the file name:”;
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename);

e.g. 2
in_file.open(“input.dat”);
if(in_file.fail())
   cout << “Cannot read file input.dat”;
```

Don’t forget to close the file stream!

```
in_file.close();
```

Streams – Reading from a File

- Read from a file stream with the same operations that you use with cin.
- Assume a file we want to read from contains:

```
Asa 100
```

- To read the above from the file, use:

```
string name;
double value;
in_file >> name >> value;
```

Now, `name` contains `Asa` and `value` contains `100`.

Streams – Reading from a File

```
#include <fstream>

ifstream in_file;
string name;
double value;
in_file.open(filename);
in_file >> name >> value;
```

Streams – Reading Words

- What really happens when reading a **string**?

```
string word;
in_file >> word;
```

1. Any whitespace is skipped (whitespace is: '**\t**' '**\n**' ' ').
2. The first character that is not white space is added to the string **word**. More characters are added until either another white space character occurs, or the end of the file has been reached.

Streams – Reading Characters

```
char ch;
in_file.get(ch);

// gets all the characters in a file
while (in_file.get(ch))
{
   // process the character ch
}
```

- Note: get() will retrieve white characters

Streams – Reading Lines

- You can read a line of input with the `getline` function and then process it further.

```
string line;
ifstream in_file("myfile.txt");
getline(in_file, line);

// reads entire file
while (getline(in_file, line))
{
   // Process line
}
```

Streams – Writing to a File

- Write to a file stream with the same operations that you use with `cout`.

```
#include <fstream>

ofstream out_file;
out_file.open(“C:\\output.txt”);
out_file << “Asa” << “ ” << 100 << endl;
```

Streams – Writing Text Output

- To write a single character to a stream, use:

```
out_file.put(ch);
```

- Use the setw manipulator to <u>set the width of the next output</u>

```
out_file << setw(10);
```

- Pad numbers using setfill()
  - e.g. 09:01

```
out_file << setfill('0') << setw(2) << hours
<< ":" << setw(2) << minutes << setfill(' ');
```

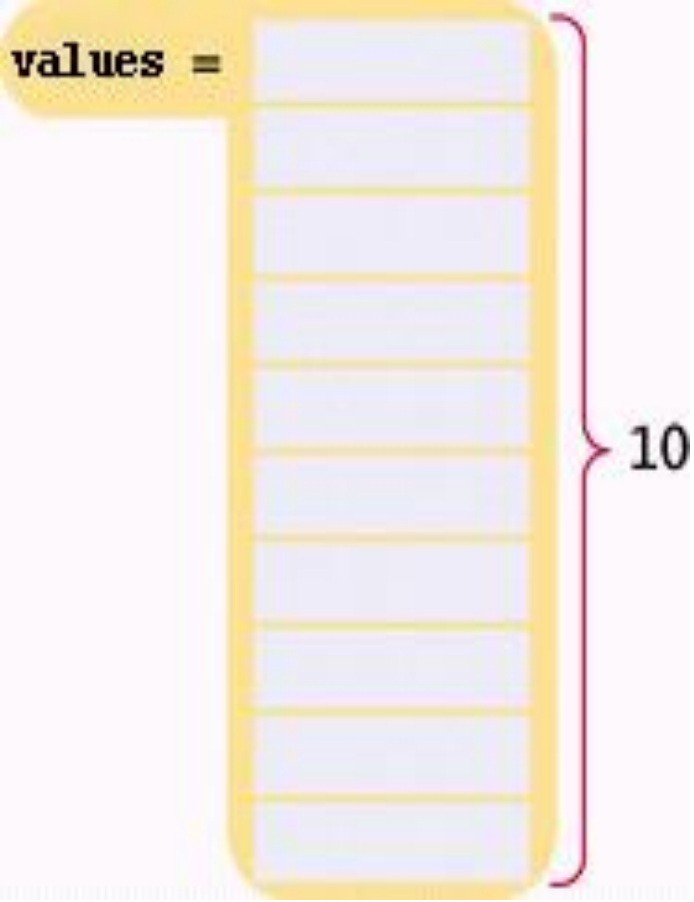
Arrays

- An array is a data type
- Fundamental mechanism in C++ for collecting multiple values
- Use an array to collect a sequence of values of the same type
- Assume we want to store 10 double values.
- Option 1: create 10 different variables
- Option 2: use an array

```
double values[10];
```

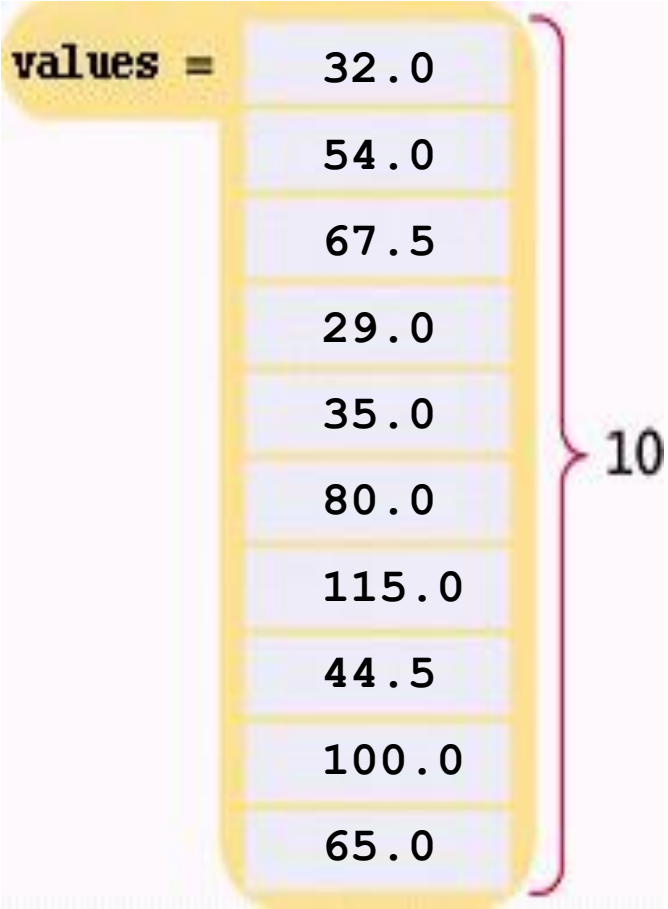
Arrays

- Arrays store data with a single name and a subscript.
- Ten elements of double type stored under one name as an array.



Arrays

```
double values[] = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```



Arrays

`int numbers[10];`	An array of ten integers.
`const int SIZE = 10;` `int numbers[SIZE];`	It is a good idea to use a named constant for the size.
`int size = 10;` `int numbers[size];`	**Caution:** the size must be a constant. This code will not work with all compilers.
`int squares[5] =` `{ 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`int squares[] =` `{ 0, 1, 4, 9, 16 };`	You can omit the array size if you supply initial values. The size is set to the number of initial values.
`int squares[5] =` `{ 0, 1, 4 };`	If you supply fewer initial values than the size, the remaining values are set to 0. This array contains 0, 1, 4, 0, 0.
`string names[3];`	An array of three strings.

Arrays

An array element can be used like any variable.

To access an array element, you use the notation:

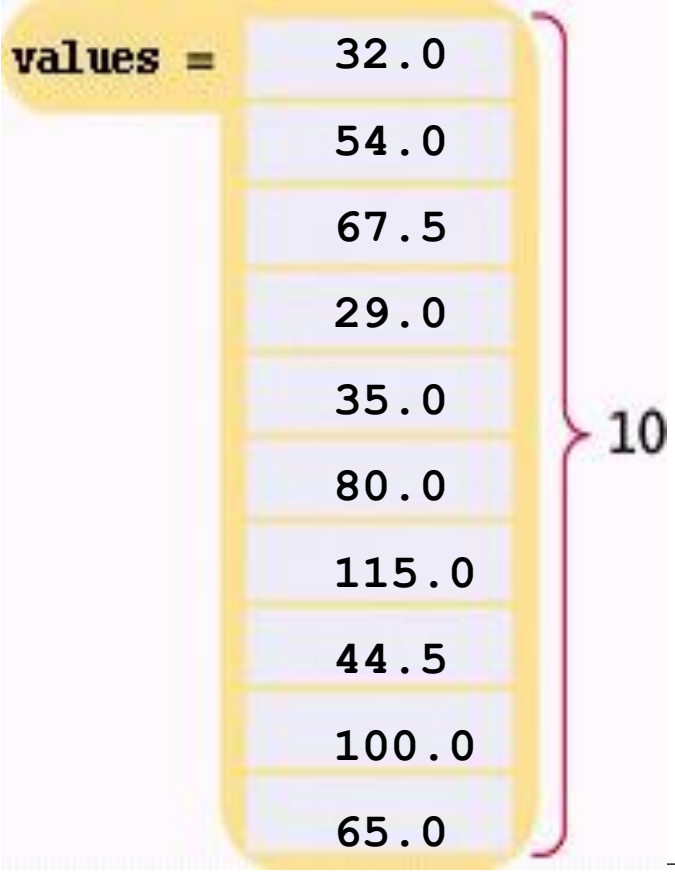
values[i]

where **i** is the *index*.

The first element in the array is at index i=0, *NOT at i=1.*

Arrays

- To access the element at index 4 using this notation:
  values[4]
  4 is the *index*.




```
double values[10];
...
cout << values[4] << endl;
```

The output will be **35.0**.
(Again because the first subscript is 0, the output for index=4 is the 5th element)

Arrays

- That is, the legal elements for the **values** array are:

  values[0], the ***first*** element
  values[1], the second element
  values[2], the third element
  values[3], the fourth element
  values[4], the fifth element
  ...
  values[9], the tenth ***and last legal*** element
  recall: **double values[10];**

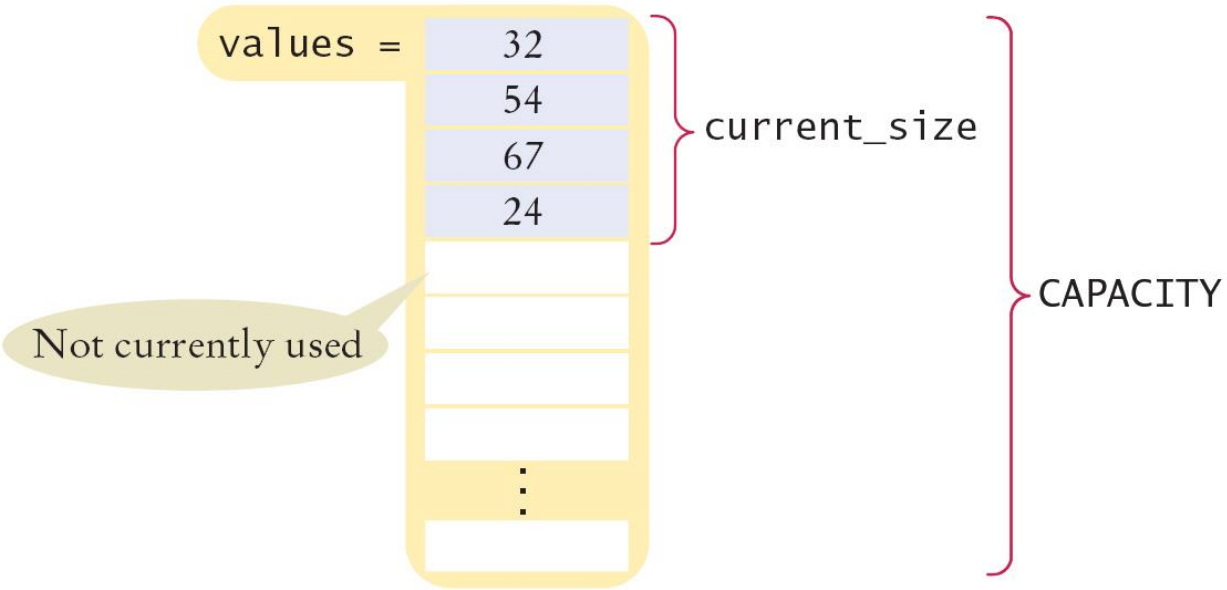
The index must be **>= 0** and **<= 9**.
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is 10 numbers.

Arrays – Partially-Filled

- What is the capacity of an array?

```
const int CAPACITY = 100;
double values[CAPACITY];
```

Assume we only add 4 values to the array.



Illegally Accessing an Array Element – Bounds Error

A *bounds* error occurs when you access an element outside the legal set of indices:

```
double values[10];
values[10] = 5.4;
cout << values[10]; //error! 9 is the last valid index
```

Doing this can corrupt data or cause your program to terminate.

Arrays and Functions

- When passing an array to a function, also pass the size of the array.
- Here is the **sum** function with an array parameter:
  Notice that to pass one array, it takes two parameters.

```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + data[i];
   }
   return total;
}
```

Arrays and Functions

- When you call the function, supply both the name of the array and the size, BUT NO SQUARE BRACKETS!!

```
double NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES]
   = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

- You can also pass a smaller size to the function:

```
double partial_score = sum(scores, 5);
```

- This will sum over only the first five **double**s in the array.

Arrays and Functions

- Array parameters are always reference parameters.
- When you pass an array into a function, the contents of the array can *always* be changed. An array name is actually a reference, that is, a memory address:

```
// function to scale all elements in array by a factor
void multiply(double values[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      values[i] = values[i] * factor;
   }
}
```

But never use an & with an array parameter – that is an error.

Arrays and Functions

- A function's return type cannot be an array.
- However, the function can modify an input array, so the function definition must include the result array in the parentheses if one is desired.

```
void squares(int n, int result[])
{
   for (int i = 0; i < n; i++)
   {
      result[i] = i * i;
   }
}
```

Structures

- A Structure is a collection of related data items, possibly of different types.
- A structure type in C++ is called struct.
- A struct is heterogeneous in that it can be composed of data of different types.
- In contrast, array is homogeneous since it can contain only data of the same type.

```
struct student {
        string name;
        string address;
        string discipline;
        float GPA;
};
```

Structures

- Define a structure type with the `struct` reserved word.

```
struct student {
        string name;
        string address;
        string discipline;
        float GPA;
};
```

```
struct StreetAddress //has 2 members
{
   int house_number; // first member
   string street_name; // second member
};

StreetAddress white_house; //defines a variable of the type

// You use the “dot notation” to access members
white_house.house_number = 1600;
white_house.street_name = "Pennsylvania Avenue";
```

Structures

Use the = operator to assign one structure value to another. All members are assigned simultaneously.

```
StreetAddress dest;
dest = white_house;
```

is equivalent to

```
dest.house_number = white_house.house_number;
dest.street_name = white_house.street_name;
```

However, you cannot compare two structures for equality.

```
if (dest == white_house) // Error
```

You must compare individual members, in order to compare the whole struct:

```
if (dest.house_number == white_house.house_number
&& dest.street_name == white_house.street_name) // Ok
```

Structures - Initialization

Structure variables can be initialized when defined, similar to array initialization:

```
struct StreetAddress
{
   int house_number;
   string street_name;
};

StreetAddress white_house = {1600, "Pennsylvania Avenue“};
// initialized
```

The initializer list must be in the same order as the structure type definition.

Structures - Initialization

```
student std1 = {"ADIL", "PAK", "BSCS",3.5};
```

```
struct student {
        string name;
        string address;
        string discipline;
        float GPA;
};
```

Structures – Functions

Structures can be function arguments and return values.

For example:

```
void print_address(StreetAddress address)
{
   cout << address.house_number << " " << address.street_name;
}
```

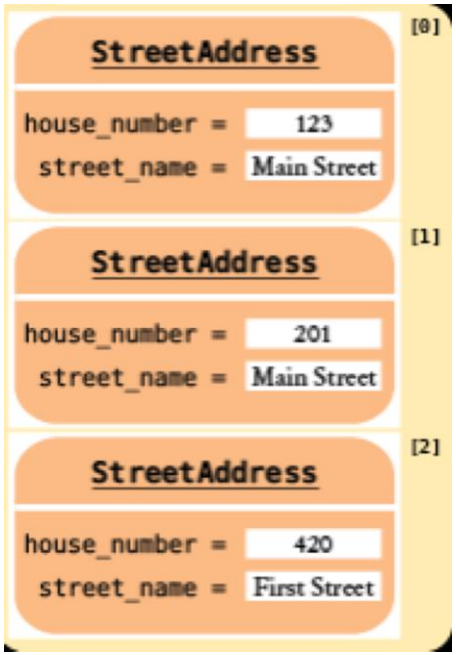
A function can return a structure. For example:

```
StreetAddress make_random_address()
{
   StreetAddress result;
   result.house_number = 100 + rand() % 100;
   result.street_name = "Main Street";
   return result;
}
```

Structures – Arrays

- You can put structures into arrays.
- For example:

```
StreetAddress delivery_route[ROUTE_LENGTH];
delivery_route[0].house_number = 123;
delivery_route[0].street_name = “Main Street”;
```



- You can also access a structure value in its entirety, like this:

```
StreetAddress start = delivery_route[0];
```

- Of course, you can also form vectors of structures:

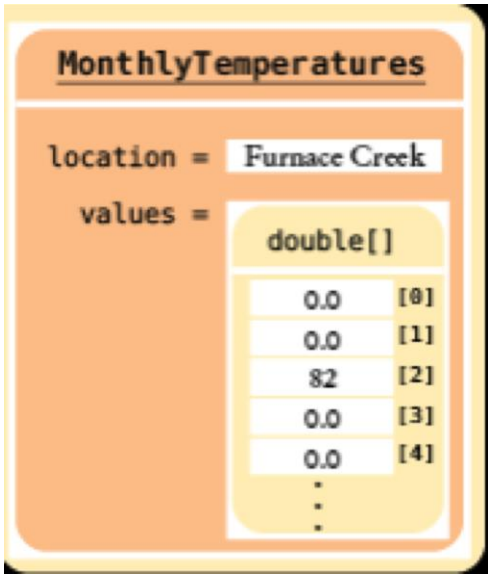
```
StreetAddress white_house;
vector<StreetAddress> tour_destinations;
tour_destinations.push_back(white_house);
```

Structures – Array Members

- Structure members can contain arrays.

For example:

```
struct MonthlyTemperatures
{
   string location;
   double values[12];
}
```



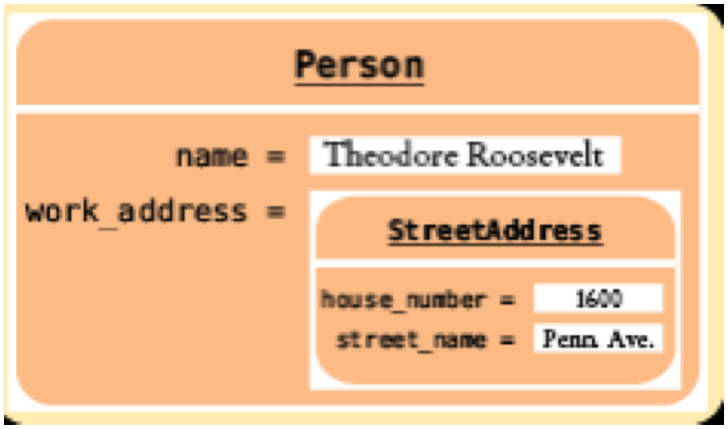
To access an array element, first select the array member with the dot notation, then use brackets:

```
MonthlyTemperatures death_valley_noon;
death_valley_noon.value[2] = 82;
```

Structures – Nested

- A struct can have a member that is another structure. For example:

```
struct Person
{
    string name;
    StreetAddress work_address;
}
```



You can access the nested member in its entirety, like this:

```
Person theo;
StreetAddress white_house;
theo.work_address = white_house;
```

To select a member of a member, use the dot operator twice:

```
theo.work_address.street_name = “Penn. Ave.”;
```

Structures – Arrays in Structs

```
struct AnimalPatient
{
   string name;
   string species;
   int age;
   double weight;
   bool gender;
   int arr[5];
};
```

```
int main()
{
   AnimalPatient p0;
   p0.name = “Steve”;
   p0.species = “Cat”;
   p0.age = 12;
   p0.weight = 14.4;
   p0.gender = 1;
   p0.arr[0] = 100;
   p0.arr[1] = 200;
   p0.arr[2] = 300;
   p0.arr[3] = 400;
   p0.arr[4] = 500;
};
```

Structures – Array of Structs

```
struct student
{
  int roll_no;
  string name;
  int phone_number;
};
```

```
int main(){
   struct student stud[5];
   int i;

   for(i=0; i<5; i++){
      //taking values from user
      cout << "Student " << i + 1 << endl;
      cout << "Enter roll no" << endl;
      cin >> stud[i].roll_no;
      cout << "Enter name" << endl;
      cin >> stud[i].name;
      cout << "Enter phone number" << endl;
      cin >> stud[i].phone_number;
   }
}
```

Structures vs Classes

```
struct student
{
  int roll_no;
  string name;
  int phone_number;
};
```

```
class student
{
   int roll_no;
   string  name;
   int phone_number;
}
```

Command Line Arguments

- We can pass information via “command line arguments”
- These args are passed to the main function
- The cmd shell program calls the main function of your program
- Your execution of the program from the command line is actually calling the main() function!
- It’s a function…

…so we can give that thing some input args

Command Line Arguments

- Example:

```
prog –v input.dat
```

- The program receives two command line arguments:
  - The string “-v”
  - The string “input.dat”
- Typically, the – in –v indicates an option.
- Strings that don’t start with – are usually file names.

Command Line Arguments

- Our program needs to process command line arguments.

```
int main(int argc, char* argv[])
{
   // code goes here
}
```

argc = **arg**ument **c**ount. argc = 1 if the user typed nothing after the program name (1 arg)

argv = **arg**ument **v**ector. Not a real vector, but just a bunch of character pointers (behaves like an array of strings for the arguments you give the program)

Command Line Arguments

```
prog -v input.dat
```

- argc = number of arguments, which is 3
- argv = contains the values of the arguments

  argv[0]: “prog”

  argv[1]: “-v”

  argv[2]: “input.dat”

Note: argv[0] is always the name of the program and argc is always at least 1.

Command Line Arguments

Example: Let's write a program that takes as input from the command line an optional argument to denote whether to use a special greeting (if present) or a default greeting (if not present).

```
int main(int argc, char* argv[])
{
    // If argv[1] is "-g", then argv[2] is the greeting file name
    // --> Read and print that greeting
    // Otherwise,
    // --> Print a default greeting
}
```

Command Line Arguments

```
int main(int argc, char* argv[])
{
   string arg = argv[1]; // coerce the character array into a string type

   if (arg=="-g") {
      ifstream infile;
      string filename = argv[2];
      string line;
      infile.open(filename);
      if (!infile.fail()) {
         getline(infile, line);
         cout << line << endl; // special greeting!
      }
     infile.close();
} else {
  cout << "Hey." << endl; // default greeting
}
   return 0;
}
```

Questions?