Topic 7

1. Defining and using pointers
2. Arrays and pointers
3. C and C++ strings
4. Dynamic memory allocation
5. Arrays and vectors of pointers
6. Problem solving: draw a picture
7. Classes of objects
8. Pointers and objects

Clases: User-defined Mixed Data Types

- To group values of a single type together under a shared name, use an array
- To group different types together with one name, use an object of a class (a structured type)
  - Like arrays, pointers prove quite useful with class type objects
- Define a class type with the `class` reserved word:

```
class StreetAddress  //has 2 members {
public:
   int house_number;  //first member
   string street_name;
};

StreetAddress white_house; //defines an object of this class

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

Objects: Assignment, but No Comparisons

Use the = operator to assign one class type object's 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 objects for equality.

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

You must compare individual members to compare the whole `object:`

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

Object Initialization

- Objects of class types can be initialized when defined, similar to array initialization:

```
class StreetAddress {
public:
   int house_number;
   string street_name;
};

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

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

Functions and `class`

Class type objects 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 class instance. For example:

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

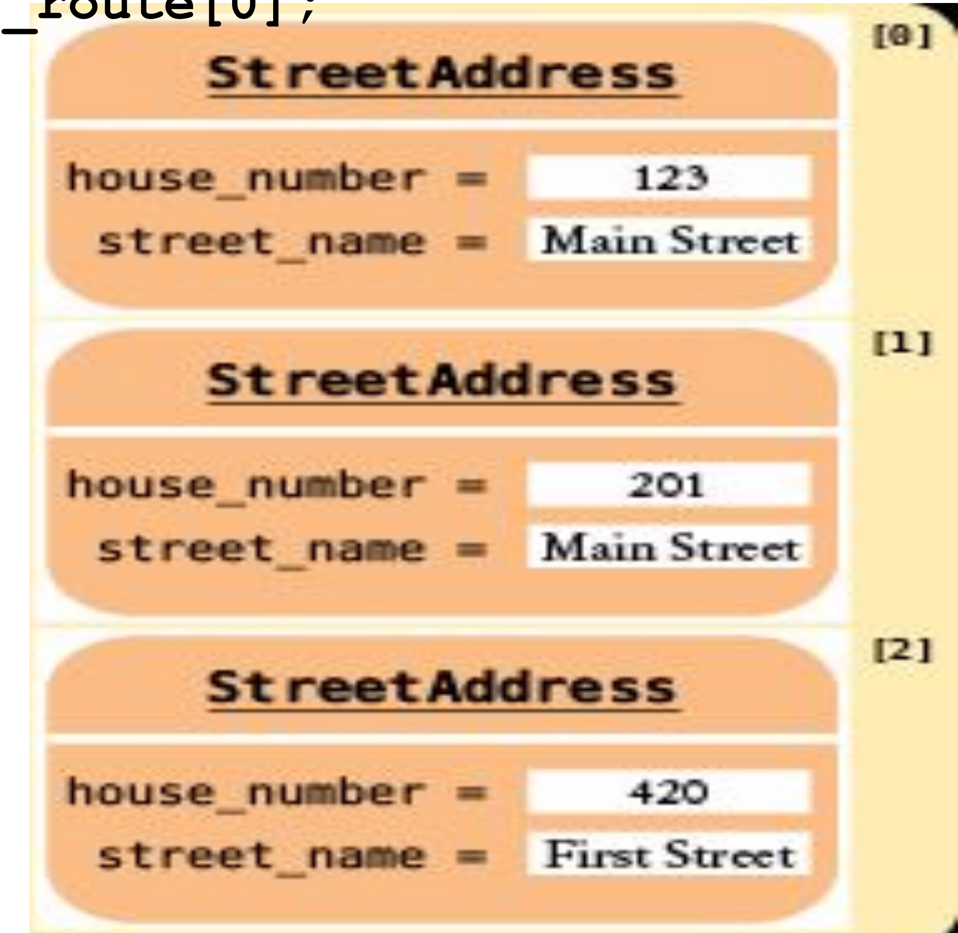
Arrays of Objects

You can put objects 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 an object's value in its entirety, like this:

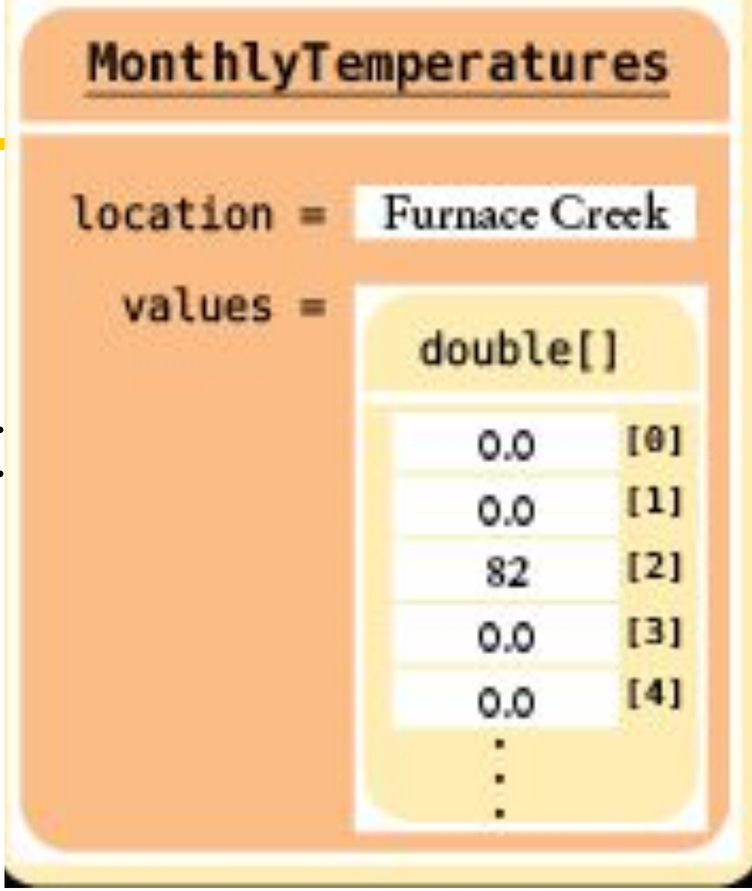
```
StreetAddress start = delivery_route[0];
```



Objects with Array Members

Objects of class types can contain arrays. For example:

```
class MonthlyTemperatures {
public:
    string location;
    double values[12];
};
```



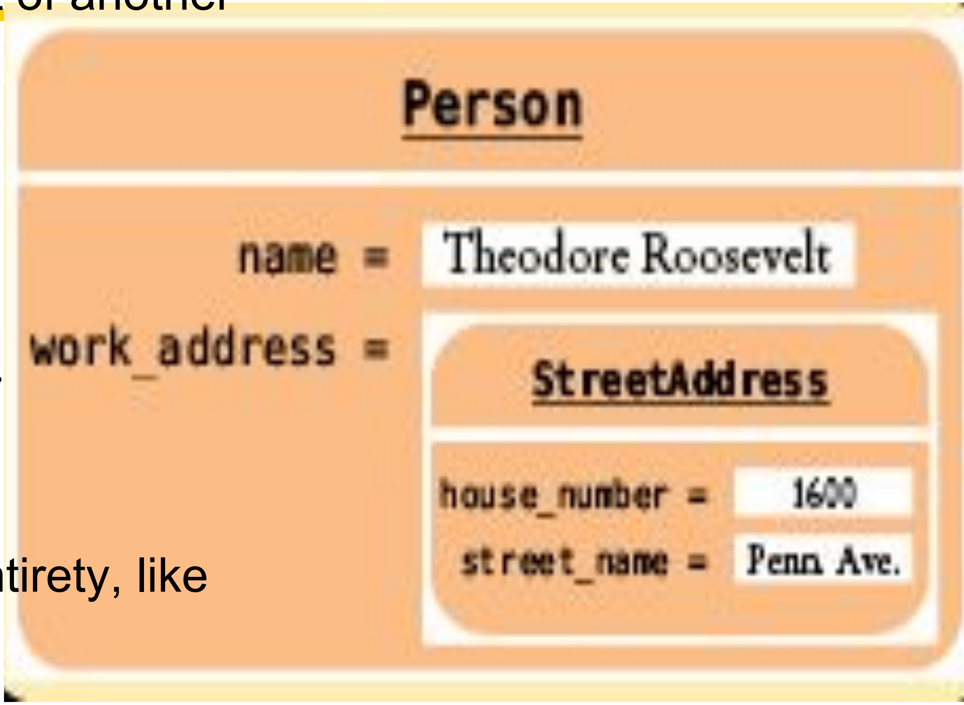
To access an array element, first select the array member with the dot notation, then use brackets:

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

Nested Objects

A class can have a member that is an object of another class. For example:

```
    class Person {
    public:
        string name;
        StreetAddress work_address;
    }
```



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

```
Person theodore;
theodore.work_address = white_house;
```

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

```
theodore.work_address.street_name =
"Pennsylvania Avenue";
```

Practice It: Structures

Write the code snippets to:

1. Declare an object "`a`" of class `StreetAddress.`

2. Set it's house number to 2201.

3. Set the street to "C Street NW".

Topic 8

1. Defining and using pointers
2. Arrays and pointers
3. C and C++ strings
4. Dynamic memory allocation
5. Arrays and vectors of pointers
6. Problem solving: draw a picture
7. Classes of Objects
8. Pointers and Objects

Object Pointers for Dynamic Allocation

As with all dynamic allocations, you use the `new` operator:

```
StreetAddress* address_pointer = new StreetAddress;
```

The following is incorrect syntax for accessing a member of the object:

```
*address_pointer.house_number = 1600; // Error
```

…because the dot operator has a higher precedence than the * operator. That is, the compiler thinks that you mean `house_number` is itself a pointer:

```
*(address_pointer.house_number) = 1600; // Error
```

Instead, you must first apply the * operator, then the dot:

```
(*address_pointer).house_number = 1600; // OK
```

Because this is such a common situation, an arrow operator **->** exists to show `class` member access via a pointer:

```
address_pointer->house_number = 1600; // OK – use this
```

Classes with Pointer Members

Objects may need to contain pointer members. For example,

```
class Employee {
public:
   string name;
   StreetAddress* office;
};
// defining 2 accounting employees:
StreetAddress accounting;
accounting.house_number = 1729;
accounting.street_name = "Park Avenue";

Employee harry;
harry.name = "Smith, Harry";
harry.office = &accounting;
Employee sally;
sally.name = "Lee, Sally";
sally.office = &accounting;
```

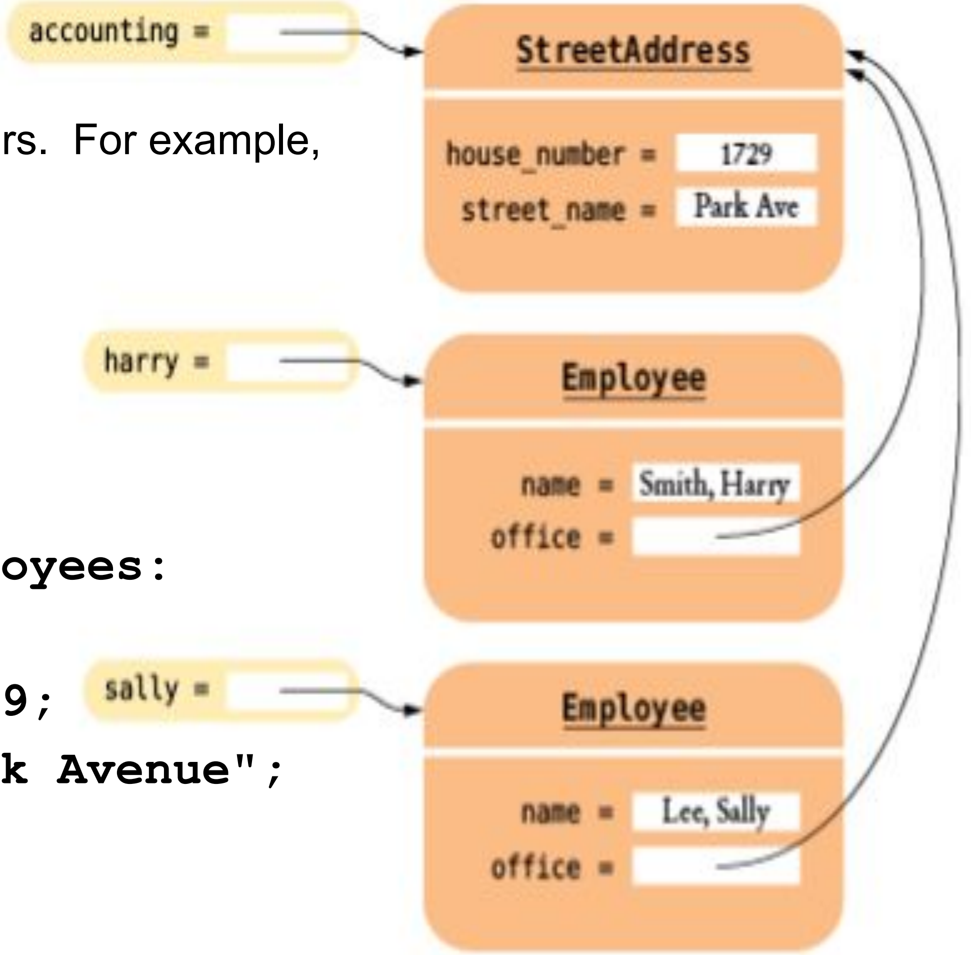


Figure 16 Two Pointers to a Shared Structure

Classes and Pointers: Complete Code Example, Part 1

```
// sec08/streets2.cpp
#include <iostream>
#include <string>
using namespace std;

class StreetAddress {
public:
   int house_number;
   string street_name;
};

class Employee{
public:
   string name;
   StreetAddress* office;
};

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

Structures and Pointers: Complete Code Example, Part 2

```
void print_employee(Employee e)
{
   cout << e.name << " working at ";
   print_address(*e.office);
}
int main()
{
   cout << "A dynamically allocated object" << endl;
   StreetAddress* address_pointer = new StreetAddress;
   address_pointer->house_number = 1600;
   address_pointer->street_name = "Pennsylvania
Avenue";
   print_address(*address_pointer);
   delete address_pointer;
   cout<<endl<< "Two employees in the same office"
<<endl;
   StreetAddress accounting;
   accounting.house_number = 1729;
   accounting.street_name = "Park Avenue";
```

Classes and Pointers: Complete Code Example, Part 3

```
Employee harry;
harry.name = "Smith, Harry";
harry.office = &accounting;

Employee sally;
sally.name = "Lee, Sally";
sally.office = &accounting;

cout << "harry: ";
print_employee(harry);
cout << endl;

cout << "sally: ";
print_employee(sally);
cout << endl;
```

Classes and Pointers: Complete Code Example, Part 4

```
    cout << "After accounting office move" << endl;
    accounting.house_number = 1720;

    cout << "harry: ";
    print_employee(harry);
    cout << endl;
    cout << "sally: ";
    print_employee(sally);
    cout << endl;
    return 0;
}
```

Smart “shared” Pointers (C++ 11 and Later)

C++ 11 introduced a `shared_ptr<>` type that automatically reclaims memory that is no longer used. For example,

```
shared_ptr<StreetAddress> accounting(new StreetAddress);
accounting->house_number = 1729;
accounting->street_name = "Park Avenue";

Employee sally;
sally.name = "Lee, Sally";
sally.office = accounting;
```

Now the `StreetAddress` structure for the accounting office has two shared pointers pointing to it: `accounting` and `sally.office`. When both of these variables go away, then the structure memory is automatically deleted.

We discuss additional strategies for memory management in Chapter 13.

Chapter Summary, Part 1

Define and use pointer variables.

- A pointer denotes the location of a variable in memory.
- The type `T*` denotes a pointer to a variable of type `T`.

  ```
  int* p = nullptr; // can point to an int
  ```

- The `&` operator yields the location of a variable.

  ```
   int i = 0;
   int* p = &i; // p points to i
  ```

- The * operator accesses the variable to which a pointer points.

  ```
   cout << p; // prints value of i, pointed to by p
  ```

- It is an error to use an uninitialized pointer.
- The `nullptr` pointer does not point to any object.
  - Please initialize unknown pointers to `nullptr`

Chapter Summary, Part 2

Understand the relationship between arrays and pointers in C++.

- The name of an array variable is a pointer to the starting element of the array.
- Pointer arithmetic means adding an integer offset to an array pointer, yielding a pointer that skips past the given number of elements.
- The array/pointer duality law:
  - `a[n]` is identical to `*(a + n),` where a is a pointer into an array and n is an integer offset.
- When passing an array to a function, only the starting address is passed.

```
printf(a); //prints array a
```

Chapter Summary, Part 3

Use C++ `string` objects with functions that process character arrays

- A value of type `char` denotes an individual character. Character literals are enclosed in single quotes.
- A literal string (enclosed in double quotes) is an array of `char` values with a zero terminator.
- Many library functions use pointers of type `char`*.
- The `c_str` member function yields a `char`* pointer from a string object.

```
string s = “This is a C++ string object”;
char arr[] = s.c_str(); //copies C++ string to C-string
```

- You can initialize C++ `string` variables with C strings.

```
string t = arr; //copies C-string to C++ string
```

- You can access characters in a C++ `string` object with the `[]` operator.

Chapter Summary, Part 4

Allocate and deallocate memory in programs whose memory requirements aren’t known until run time.

- Use dynamic memory allocation if you do not know in advance how many values you need.
- The `new` operator allocates memory from the free store.

  ```
  int* p = new int[50]; // allocate array of 50 ints
  ```

- You must reclaim dynamically allocated objects with the `delete` or `delete[]` operator.

  ```
  delete[] p; //done using our int array pointed to by p
  p = nullptr; //set p to nullptr to avoid dangling pointer usage
  ```

- Using a dangling pointer (a pointer that points to memory that has been deleted) is a serious programming error.
- Every call to `new` should have a matching call to `delete`.

Chapter Summary, Part 5

Work with arrays of pointers.

- Draw diagrams for visualizing pointers and the data to which they point.
  - Draw the data that is being processed, then draw the pointer variables. When drawing the pointer arrows, illustrate a typical situation.

Use classes to aggregate data items.

- An object of a class combines member values into a single value.
- Use the dot notation to access members of an object.

```
Streetaddress home;
home.house_number = 1234;
```

- When you assign one object value to another, all members are assigned.

Work with pointers to objects.

- Use the -> operator to access an object member through a pointer

```
Streetaddress* p = new Streetaddress;
P->house_number = 1234;
```