Pointers

Lab 9: https://maryash.github.io/135/labs/lab_09.html

- Sadab Hafiz

Introduction to basic Classes

Classes in C++ are user-defined compound data-types that can be used for grouping together several related variables. These “variables” inside a class are called fields (or data-members) of the classes.

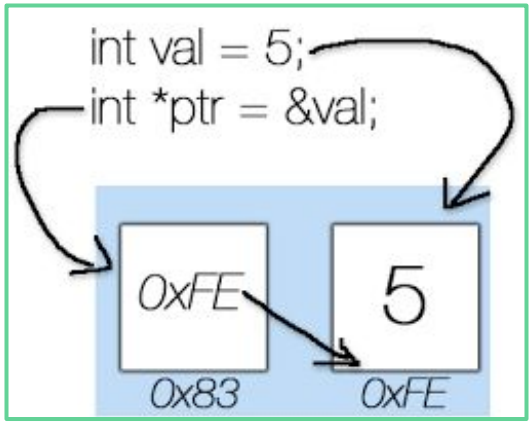
```
// class with three public data members

class Coord3D {

      public:

            // public data members

            double x;

            double y;

            double z;

};
```

Introduction to basic Classes (continued...)

If you create a variable of a class, it can be referred to as an object.

```
. . . class definition of Coord3D from previous slide

int main {

      Coord3D pointP = {10, 20, 30};   // create a `Coord3D` object

      cout << pointP.x << endl;        // prints the value of data-member `x` which is 10

      pointP.y = 50;                   // set data-member `y` of `pointP` equal to 50

      cout << pointP.y << endl;        // will print 50

}
```

Introduction to Pointers



Pointer variables are a feature of systems programming languages, such as C++ or C, that are able to store the memory addresses of other variables and objects used by the program.

Knowing the address of an object can very convenient. You can freely pass it around into functions, if the function needs to operate on that object.

Whenever we need to tell someone the location of some place, we give them the address. We don’t have to give them details about the whole location. The address can be written down in a small piece of paper. Similarly, instead of referring to an entire object or a large variable, using pointers can help us access much larger objects while passing it in a much smaller size.

&

695 Park Ave,
New York, NY 10065

*

695 Park Ave,
New York, NY 10065

Dereferencing a Pointer

Dereferencing refers to the process through which the value that a pointer is pointing to can be accessed using the * operator. For example:

```
. . . class definition of Coord3D from previous slide

int main {

     Coord3D pointP = {10, 20, 30};   // create a `Coord3D` object with x=10, y=20, z=30

     Coord3D * p = &pointP;           // pointer `p` points to `pointP` object

     Coord3D pointPCopy = *p;         // dereference p and make a copy of the object

     cout << pointPCopy.z << endl;    // prints 30

}
```

Introducing the -> operator

Dereferencing and accessing a classes data-members requires us to create a local copy of the object. What if we wanted to access the data-members directly from the pointer? In that case use -> operator:

```
. . . class definition of Coord3D from previous slide

int main {

      Coord3D pointP = {10, 20, 30};      // create a `Coord3D` object with x=10, y=20, z=30

      Coord3D * p = &pointP;              // pointer `p` points to `pointP` object

      cout << p -> y << endl;             // prints 20

      p -> y = p -> z + 10;               // update the value of y data-member (30+10)

      cout << pointP.y << endl;           // prints 40

      // changes made using -> will also make changes to the original object!!!

}
```

Helpful functions from <cmath> library

$$\text{Length of the vector } OP = \sqrt{x^2 + y^2 + z^2}$$

The formula for distance requires us to get the square of x, y, and z. We also need to get the square root of the sum of the squares. In order to perform these operations, we can use some functions from <cmath> library:

pow(a,b): This function call will return a^b

sqrt(a): This function will return √a

Automatic memory allocation

Normally, any variable “lives” only within the block where it is declared, and disappears once the program execution leaves this scope. We know that already, right?

This memory management is called automatic, the program allocates memory for each variable when it enters the scope of the variable, and deletes that memory when leaving that scope.

This is how C++ handles local variables within functions and scope.

A problem with automatic memory allocation

The following function that's supposed to create a poem and return its memory address, will not work (reliably):

```
string * createAPoem() {

      string poem =    // making a string with a poem

      "   Said Hamlet to Ophelia,            \n"

      "   I'll draw a sketch of thee,        \n"

      "   What kind of pencil shall I use?   \n"

      "   2B or not 2B?                      \n";

    return &poem;    // and returning its address

}
```

Since the variable poem exists locally inside the function, after exiting the function, the memory allocated for this string gets claimed and freed.

Even though we returned the address where the poem was, after the function exits, that address may be taken and used by some other part of your program, the poem may be easily overwritten by some other value.

How do we fix this? Using keyword ‘new’

We can allocate a chunk of memory for the poem so that it would remain persistent and would not be claimed by the program after the function exits. Using the keyword `new`:

```
string * createAPoemDynamically() {

     string * ppoem;             // declare a pointer to string to store the address

     ppoem = new string;         // <-- DYNAMICALLY ALLOCATE MEMORY

                                 //     for a poem string and store its address in the pointer

     *ppoem =                    // put a poem there

     "   If you know              \n"

     "   you know                 \n";

     return ppoem;               // return the address where the poem is

}
```

More about keyword `new`

The address of a dynamically allocated memory can be passed around, returned from a function, or stored in another variable, etc. The dynamically allocated memory will remain persistently in the computer memory throughout the program execution:

```
int main() {

      string * p;

      p = createAPoemDynamically();

      // The memory at the address p still stores the poem we put in it during the function call. Neat!

      // At any later moment, you can print it out:

      cout << *p;

                              // You can also save the address into another pointer variable:

      string *p2 = p;         // then both pointers, p and p2, will be pointing to the same poem

      cout << *p2;

}
```

What's the catch? We have to `delete` it.

Dynamically allocated memory is awesome, it stays persistently and does not depend on the variables and their life times. However, it also comes with problems. Once this memory is not needed, it must be manually released to the system, otherwise if we only keep allocating memory and never giving it back, we may run out of memory eventually.
Once a dynamically allocated memory is not needed anymore, we can use the keyword `delete` to get rid of that memory:

```
int *p = new int;                  // allocate an integer dynamically

*p = 1234;                         // we are using it

cout << *p << endl;                // print out 1234

delete p;                          // once it's not needed, delete it:
```

Note: Dynamic arrays can be used to create custom data structures. They have many different uses. Learn how to allocate dynamic arrays from the lab: Dynamic Array Example