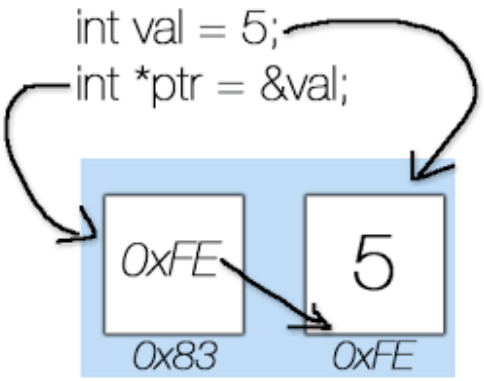
Lab 9. Pointers



Introduction

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.

The usefulness of addresses is pretty obvious in real life: the **Hunter College campus** is a huge multi-building complex. It is really big. So big that you cannot move it around, and you definitely cannot put the whole **Hunter College campus** in your pocket.

On the other hand, its address, `695 Park Ave, New York, NY 10065`, is much smaller than the building itself, you can write it down on a piece of paper to pass it around to other people or to keep it in your notes without taking much space.

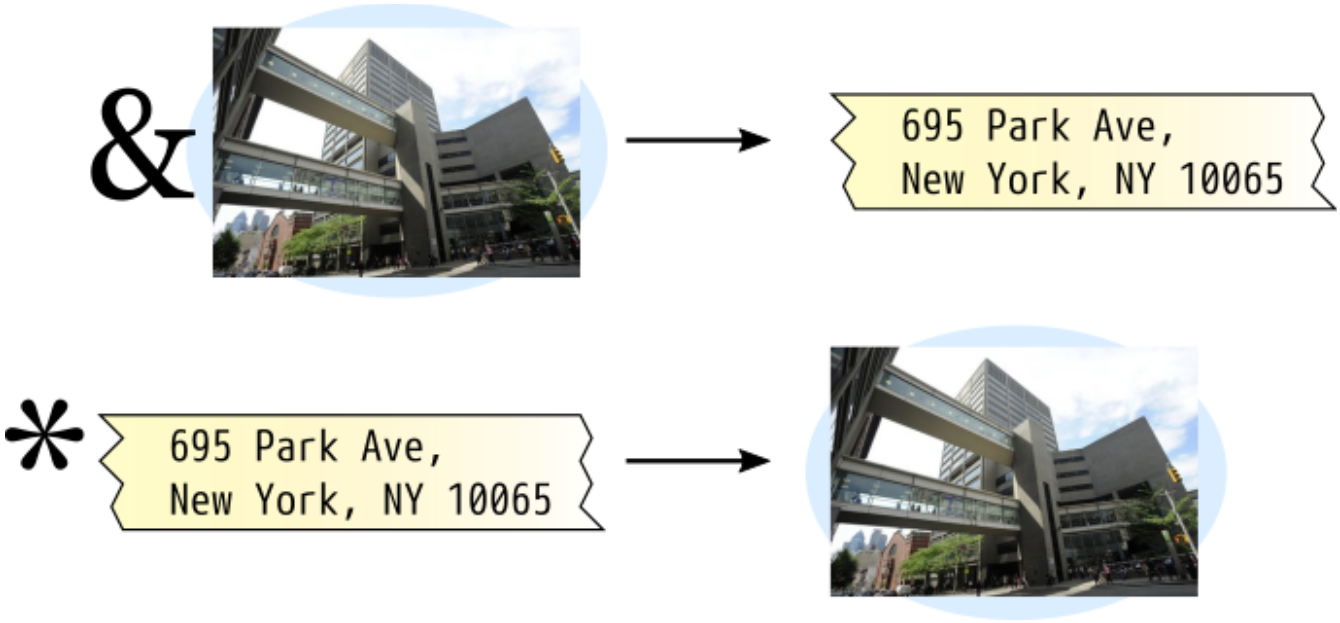
So, a pointer variable is pretty much like this piece of paper with the address. It takes a little bit of memory (several bytes, the exact number can vary depending on your system and the type of pointer), and lets you refer to other (potentially much bigger) objects.

Operators `&` and `*`:

`&x` is the address of the object `x`

`*p` is the object at the address pointed by `p`

Action of these operators in the Hunter campus analogy:



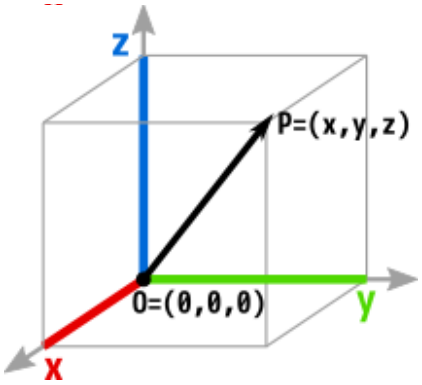
Example:

```
string book = "In a hole in the ground there lived a hobbit...";

string *p;   // declaring a pointer.
p = &book;   // storing the address of 'book' in the pointer

cout << *p << endl; // dereferecing the pointer for printing
                    // the book on the screen

cout << p << endl;  // However, if you print the pointer itself,
                    // you will see a hexadecimal code like:
                    // 0x0A274B3A
```

Task A. Length and distance in 3D space

A **point** in 3D space can be represented using three coordinates. In Cartesian coordinate system

these coordinates are called *(x,y,z)*, describing the position of the point along the three orthogonal axes:



The **origin** of the coordinate system is denoted by *O* and has coordinates *(0,0,0)*.

A point *P=(x,y,z)*, together with the origin, defines a 3D **vector** *OP*. The distance from *O* to *P*, or in other words, the length of the vector *OP* can be computed using the euclidean distance formula:

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

We are provided with a class type that represents coordinates in 3D:

```
class Coord3D {
public:
    double x;
    double y;
    double z;
};
```

Write a program `3d-space.cpp`, in which you define a function `length()` that receives the coordinates of a point *P* passed as a pointer, and computes the distance from the origin to the point *P*:

```
double length(Coord3D *p);
```

A usage example:

```
int main() {
    Coord3D pointP = {10, 20, 30};
    cout << length(&pointP) << endl; // would print 37.4166
```

```
}
```

Notice that we pass the memory address `&pointP`, where the object of this class is located. The function should *dereference* this address to get the corresponding fields `x`, `y`, and `z` for computing the length.

Task B. Farther from the origin?

In the same file `3d-space.cpp`, add a function

```
Coord3D * fartherFromOrigin(Coord3D *p1, Coord3D *p2);
```

Which receives the coordinates of two points (passed as pointers), and returns the pointer of the point that is farther away from the origin.

A usage example:

```
int main() {
    Coord3D pointP = {10, 20, 30};
    Coord3D pointQ = {-20, 21, -22};

    cout << "Address of P: " << &pointP << endl;
    cout << "Address of Q: " << &pointQ << endl << endl;

    Coord3D * ans = fartherFromOrigin(&pointP, &pointQ);

    cout << "ans = " << ans << endl; // So which point is farther?
}
```

When testing your code, look at the reported address of the answer `ans` and **determine whether it reports P or Q**. You can use a calculator or WolframAlpha) to check the numbers. Try other coordinates of points P and Q to confirm that the program works.

Task C. Velocity and moving objects

Let's consider an object moving in 3D space. We already know how to describe its position. (We will be assuming metric system, thus distances will be implicitly measured in meters and time in seconds.)

The object's **velocity** can be represented in the same 3D coordinate system as its displacement per second in the coordinates x, y, and z:

```
Coord3D vel = {5, -3, 1}; // x, y, z components of the velocity
```

When moving at constant velocity `vel`, the object's new position after the elapsed time `dt` can be computed as

```
x' = x + vel.x * dt;

y' = y + vel.x * dt;

z' = z + vel.x * dt;
```

Let's implement it. In the same program, write a function

```
void move(Coord3D *ppos, Coord3D *pvel, double dt);
```

which gets the position and the velocity of an object and has to compute object's new coordinates after the time interval `dt`. The function does not return any values, instead, it should **update** the object's position `ppos` with its new position coordinates.

Because we pass the coordinates `Coord3D * ppos` as a **pointer**, all changes to the fields of the class pointed by `ppos`, will affect the original object you pass into the function, not its local copy. Example:

```
int main() {
    Coord3D pos = {0, 0, 100.0};
    Coord3D vel = {1, -5, 0.2};
```

```
    move(&pos, &vel, 2.0); // object pos gets changed
    cout << pos.x << " " << pos.y << " " << pos.z << endl;
    // prints: 2 -10 100.4
}
```

Notice that we are not passing anything by reference: We pass the *address*, `&pos`, and the function manipulates the original object `pos`, because it knows its address in the memory.

On dynamic 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**.

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 only locally inside the function, after exiting the function, the **memory allocated for this string gets claimed and freed by the program**. 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.

The keyword `new`

> **Question(?):**
> Can we 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?

Yes we can, using the keyword `new`:

```
string * createAPoemDynamically() {
    string *ppoem;      // declare a pointer to string
                        // (it will 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
      "   Those             \n"
      "   who can write     \n"
      "   have a            \n"
      "   lot to            \n"
      "   learn from those  \n"
      "   bright            \n"
      "   enough            \n"
      "   not to.           \n";

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

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;
}
```

The keyword `delete`

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.

When we know that we don't need a dynamically allocated memory anymore, we have to delete it with keyword `delete`:

```
// allocate an integer dynamically
int *p = new int;
*p = 1234;          // we are using it
cout << *p << endl; //  ''  ''  ''  ''

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

Allocating arrays dynamically (operators `new ... []` and `delete[]`)

Dynamic memory allocation of arrays uses special operators with square brackets, we have to specify how many elements we want to get:

```
int n = 100; // how many elements we need
// Getting memory
string * lines = new string[n]; // array of 100 strings

lines[0] = "Roses are red";
lines[1] = "Violets are blue";

// ... Keep using the array ...

// Don't need it anymore - delete
delete[] lines;
```

When it is not needed, use operator `delete[]` to release the memory.

Task D. Fix the program so that it does not crash your computer:

You are provided with the following program `poem.cpp`. All is good, and the memory is allocated dynamically, but it crashes your computer, because it runs out of memory really quickly:

```
#include <iostream>
using namespace std;

string * createAPoemDynamically() {
    string *p = new string;
    *p = "Roses are red, violets are blue";
    return p;
}

int main() {
    while(true) {
        string *p;
        p = createAPoemDynamically();
```

```
            // assume that the poem p is not needed at this point

        }
    }
```

Fix this program `poem.cpp`. It should still keep creating poems, but all dynamically allocated memory should get deleted when it is not needed. (The program can be stopped with `Ctrl+C` in the terminal.)

Task E. Creating and deleting objects dynamically

In the program `3d-space.cpp`, add functions that create, delete, and coordinate objects dynamically:

```
// allocate memory and initialize
Coord3D* createCoord3D(double x, double y, double z);

// free memory
void deleteCoord3D(Coord3D *p);
```

A usage example:

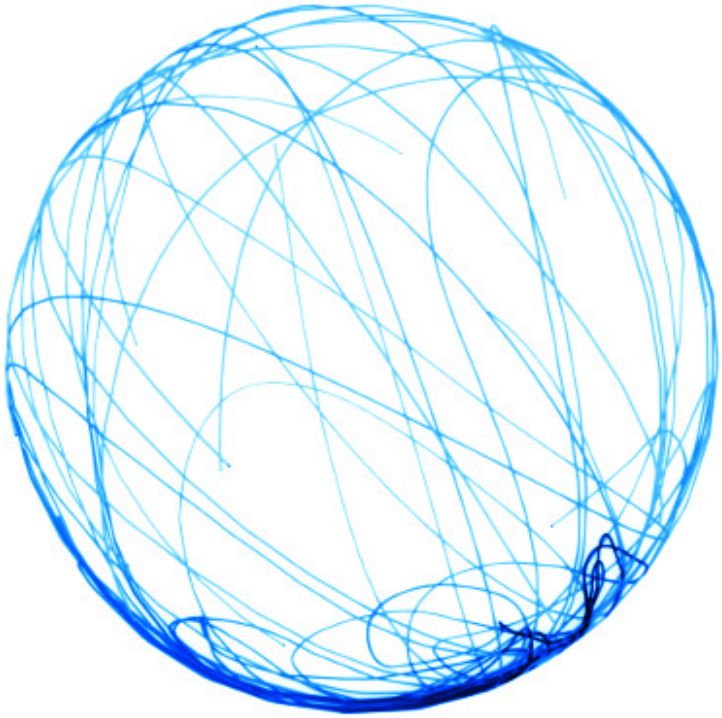
```
int main() {
    double x, y, z;
    cout << "Enter position: ";
    cin >> x >> y >> z;
    Coord3D *ppos = createCoord3D(x,y,z);

    cout << "Enter velocity: ";
    cin >> x >> y >> z;
    Coord3D *pvel = createCoord3D(x,y,z);

    move(ppos, pvel, 10.0);
```

```
    cout << "Coordinates after 10 seconds: "
         << (*ppos).x << " " << (*ppos).y << " " << (*ppos).z << endl

    deleteCoord3D(ppos); // release memory
    deleteCoord3D(pvel);
}
```

Expected output:

```
$ ./a.out
Enter position: 10 20 30
Enter velocity: 5.5 -1.4 7.77
Coordinates after 10 seconds: 65 6 107.7
```

Task F (Bonus Task). Making your own class `Particle`



Write a new program `particle.cpp` (you may copy anything you want from `3d-space.cpp` if need be).

The program should declare a new `class Particle`, which

- stores **position** and **velocity** of the particle, and
- provides the following programming interface:

```
// dynamically allocate memory for a particle and initialize it
Particle* createParticle(double x, double y, double z,
                         double vx, double vy, double vz);
// set its velocity to (vx, vy, vz)
void setVelocity(Particle *p, double vx, double vy, double vz);
// get its current position
Coord3D getPosition(Particle *p);
// update particle's position after elapsed time dt
void move(Particle *p, double dt);
// delete all memory allocated for the particle passed by pointer
void deleteParticle(Particle *p);
```

Implement these five functions. `deleteParticle` must delete all dynamically allocated memory that is created by `createParticle` function.

Here is a usage example of the class and its programming interface. It models accelerated motion of a particle, which is done by making the velocity of the particle increase on each time step.

```
int main() {
    // make new particle
    Particle *p = createParticle(1.0, 1.5, 2.0, 0.1, 0.2, 0.3);
    double time = 0.0;
    double dt = 0.1;
    while(time < 3.0) {
        // update its velocity
        setVelocity(p, 10.0 * time, 0.3, 0.1);

        // move the particle
```

```
            move(p, dt);
            time += dt;

            // reporting its coordinates
            cout << "Time: " << time << " \t";
            cout << "Position: "
                 << getPosition(p).x << ", "
                 << getPosition(p).y << ", "
                 << getPosition(p).z << endl;
        }
        // remove the particle, deallocating its memory
        deleteParticle(p);
    }
```

Expected output:

```
    $ ./a.out
    Time: 0.1       Position: 1, 1.53, 2.01
    Time: 0.2       Position: 1.1, 1.56, 2.02
    Time: 0.3       Position: 1.3, 1.59, 2.03
    Time: 0.4       Position: 1.6, 1.62, 2.04
    Time: 0.5       Position: 2, 1.65, 2.05
    Time: 0.6       Position: 2.5, 1.68, 2.06
    Time: 0.7       Position: 3.1, 1.71, 2.07
    Time: 0.8       Position: 3.8, 1.74, 2.08
    Time: 0.9       Position: 4.6, 1.77, 2.09
    Time: 1         Position: 5.5, 1.8, 2.1
    Time: 1.1       Position: 6.5, 1.83, 2.11
    Time: 1.2       Position: 7.6, 1.86, 2.12
    Time: 1.3       Position: 8.8, 1.89, 2.13
    Time: 1.4       Position: 10.1, 1.92, 2.14
    Time: 1.5       Position: 11.5, 1.95, 2.15
    Time: 1.6       Position: 13, 1.98, 2.16
    Time: 1.7       Position: 14.6, 2.01, 2.17
    Time: 1.8       Position: 16.3, 2.04, 2.18
```

```
Time: 1.9       Position: 18.1, 2.07, 2.19
Time: 2         Position: 20, 2.1, 2.2
Time: 2.1       Position: 22, 2.13, 2.21
Time: 2.2       Position: 24.1, 2.16, 2.22
Time: 2.3       Position: 26.3, 2.19, 2.23
Time: 2.4       Position: 28.6, 2.22, 2.24
Time: 2.5       Position: 31, 2.25, 2.25
Time: 2.6       Position: 33.5, 2.28, 2.26
Time: 2.7       Position: 36.1, 2.31, 2.27
Time: 2.8       Position: 38.8, 2.34, 2.28
Time: 2.9       Position: 41.6, 2.37, 2.29
Time: 3         Position: 44.5, 2.4, 2.3
```

How to submit your programs

Each program should be submitted through Gradescope

Write separate programs for each part of the assignment.
Submit only the source code (.cpp) files, not the compiled executables.
Each program should start with a comment that contains your name and a short program description, for example:

```
/*
Author: your name
Course: CSCI-136
Instructor: their name
Assignment: title, e.g., Lab1A

Here, briefly, at least in one or a few sentences
describe what the program does.
*/
```