1 Soluciones del primer seminario

2. ¿Qué significan por valor, por puntero y por referencia en C++? ¿Cómo funciona esto en memoria?

   The difference between passing by reference and passing by value is pretty important. When you pass by value, you're really passing a copy of the value in question. No matter what happens in the function, the value in the caller will remain unchanged. So, say you've got a function that adds one to an int and returns an int:

   ```
   int addOne(int theNumber)
   {
       theNumber += 1;
       return theNumber;
   }
   ```

   Here, you're passing by value. You'd call it like this:

   ```
   int a = 10;
   int b = addOne(a); // b gets 11, but a remains the same
   ```

   If you want to pass by reference instead, the function would look like this:

   ```
   int addOne(int &theNumber)
   {
       theNumber += 1;
       return theNumber;
   }
   ```

   Note that the body of the function stays the same. Again, you call it like this:

   ```
   int a = 10;
   int b = addOne(a); // b gets 11, but this time a is also changed to 11.
   ```

   The big difference here is that you're passing a reference to `a`. It's really a sort of implicit pointer to `a`, but you can think of it as passing a itself. Since you're passing a instead of copy of the value of `a`, `a` itself will actually be changed by the function.

   The third way, passing the address, looks like this:

   ```
   int addOne(int *theNumber)
   {
       *theNumber = *theNumber + 1;
       return *theNumber;
   }
   ```

This does the same thing as the reference version, but the pointer here is explicit. You use it like this:

```
int a = 10;
int b = addOne(&a); // b gets 11, but this time a is also changed to 11.
```

So, in this case you're explicitly passing the address of `a`, which is to say a pointer to `a`. If you're used to passing by value only, this should be familiar. This is how you pass a by reference in `C` and some other `C`-like languages. It works fine, but you have to do all the pointer stuff yourself. `C++` adds the concept of passing by reference to the language to make this all easier.

A final possibility is to pass a `const` reference, which avoids copying the value, but prohibits changing it in the called function. If a function takes a `const` reference, you can read that as a promise not to change the parameter (and it's a promise that the compiler will enforce). This is particularly useful if the value is more than a few bytes, so that it's desirable to avoid copying the value if you can. Objects are often passed by `const` reference for this reason.

So, as a guideline, pass by value or `const` reference when you don't want the thing the caller is passing to change. Pass by reference when you do want it to change. And don't pass by pointer unless you're dealing with a `C` library or other code that requires it.

3. Constructores:

   a. Default
   b. Copy
   c. Move
   d. Explicit
   e. constexpr
   f. inline

1.0.1 Default

```
// Default constructor
Box() {}
Box(int i = 1) {}
Box() = default;
Box() = delete;
```

Default initialization for classes, structs, and unions is initialization with a default constructor. The default constructor can be called with no initialization expression or

with the new keyword.

```
MyClass mc1;
MyClass* mc3 = new MyClass;
```

1.0.2 Copy

```
Box(Box& other);
Box(const Box& other);
Box(volatile Box& other);
Box(volatile const Box& other);
Box(Box& other, int i = 1);
Box(Box& other) = delete;
```

`volatile` is a type qualifier that you can use to declare that an object can be modified in the program by the hardware.

When you define a copy ctor, you should also define a copy assignment operator (=).

```
ClassName& operator=(const ClassName& x);
```

Tiene que estar claro cuando se llama el constructor copia y cuando el operador copia.

```
ClassName a, b;
a = b;  // operador copia

ClassName c = a;  // constructor copia
```

1.0.3 Move

```
Box(Box&& other);
```