Subject: Re: Difficulty with Class declaration
Posted by mr_ped on Tue, 06 Apr 2010 08:30:09 GMT
View Forum Message <> Reply to Message

about pointers and references...
If you are coming from "higher" language, it should be new for you, because many modern languages hide pointers internally and let you work with references only, so no wonder it does confuse you.

Don't worry, the concept itself is very simple, and U++ will actually support you in *not* using pointers extensively (which is often source of many memory leaks or damaged memory for inexperienced C++ programmers), so learning the concept and learning the U++ way of thinking (everything belongs somewhere) will get you back to productivity quite fast (faster then learning the "good old C").

It's about memory. Let's imagine a street with houses, each house having 1 room capable to store 32bits for example, they are on the street "memory" and each house has different number (going from 0 up).
Now let's have two variable declarations:
int IntegerValue;
int* IntegerPointer;

Both will reserve one house to hold it's value, let's say IntegerValue has house 100, IntegerPointer has house 101.

IntegerValue = 13;
This will tell compiler you want to store value 13 into house 100, but you don't need to remember it was house 100 and to do "memory[100] = 13;", you can use the "IntegerValue" name as an alias, and let compiler to figure it out for you.

IntegerPointer = &IntegerValue;
Notice the left side, there's no asterisk or ampersand, so it's pure value assignment, like in previous case. On the right side there's ampersand. This tells compiler you don't want the "13" value of IntegerValue, but you want it's true address (pointer), so the result will be 100; House 101 holds value 100 now.

int X = *IntegerPointer;
And the asterisk will dereference the IntegerPointer value (100) and use it as an address, so the X will be filled up with value from house 100, and that one holds 13.

int Y = IntegerPointer; would put 100 into Y (and compiler would warn you you are casting pointer to integer value, so it's probably not what you want).

int** pointer = &IntegerPointer;  will put value 101 into "pointer", which is pointer to a pointer to an integer (thus int** data type)

"int*" is data type "pointer to int".

also keep in mind the [] (C arrays) works as dereferencing operator too, so:
int A = IntegerPointer[0];   //A == 13 (value from house 100+0)
int B = IntegerPointer[1];
//WRONG, but will work, we are going out of bounds of original IntegerPointer usage, as it was pointing to single value, not an array ... and in house 100+1 there's value 100, because there's the IntegerPointer itself stored, but that's just coincidence, there could have been anything.

int properarray[10];  //properarray is same type as int*, but the compiler will reserve 10 houses in single range, like houses from 120 to 129. properarray then holds value 120, and using [] operator you can fetch up particular member of array.

uhm... this is probably the shortest possible introduction to C pointers, make sure you fully understand each statement, then we can move to new/delete operators, object instances and why U++ does help you to not bother with them too often (which makes U++ code looks quite like Java, because you don't have to bother with all this pointers stuff most of the time).

I completely omit C/C++ references (ampersands in type definitions and functions calls), because they are simple to explain once you understand pointers. Also I did omit function pointers and other syntax sugar, which is not needed to understand the concept, because in the end, the concept is simple:
- values are numbers
- pointers are addresses to numbers (and address itself is implemented as a simple number too, so for CPU there's no difference between number/pointer, it's just 13 and 100, but compiler+syntax does allow you to use them in different context and warn you when you use it plainly wrong)

update - also note after:
int* IntegerPointer;
we have reserved space in memory, but it's value is not initialized, so int x = *IntegerPointer; is mistake and you will either crash for referencing protected memory, or get bogus value.
This is one of the reasons why the references in C/C++ exist, basically they are identical to pointers, but they can't be uninitialized.
Also the good programming practice is to initialize pointers immediately, like int* IntegerPointer = NULL;
In case you then use it before setting it up to it's true value, you will get crash with access to adress 0, which is better then hunting random bogus numbers and occasional random crashes.