Subject: Re: Difficulty with Class declaration
Posted by mr_ped on Tue, 06 Apr 2010 15:30:38 GMT
View Forum Message <> Reply to Message

Sorry, can't afford to study your problem in details, so I will response in dry theory and short introductions to general C/C++ concepts. (I hope other members will fill the gap with your project like they did up till now).

WHY pointers - excellent question and I'm feeling ashamed I didn't even think about explaining that.

Let's get back to stone age of programming to see why pointers did born.
There are generally two memory pools available for PC program, that's the STACK and HEAP. (and GPU memory nowadays, but it's the same stuff as highly specialized heap)

STACK is local to executed code, addressed by "sp" register in CPU. Take for example this piece of ugly code:

```
int ExplainStack1( int arg1 ) {
  int my_x = 13;
  my_x += arg1;
  return my_x;
}

//we start to execute here!
int result = ExplainStack1( 7 );
```

from this code snippet it's impossible to tell where result is stored, so ignore that one.
After execution starts, you want to call function ExplainStack1 with number 7 as parameter.
There's empty stack at the start, you PUSH there the number 7 (stack is LIFO queue, so you push it at top of stack), then you call the function (the return address for CPU is pushed on stack as well).
Inside function you reserve additional stack space for "my_x" variable (like pushing unknown value on top of stack).
So the stack looks like this:
top) my_x //<- here points the CPU "sp"
+1) return address
+2) 7
You fetch value of stack+2 (the 7), add it to top of stack (get resulting 20), put that into stack on position where result is expected by the original caller (let's say it's top+2 replacing the "7" for simplicity reasons).
You POP the local function mess from stack (the my_x and return address), and jump to return address...
Stack looks like this now:
top) 20 //the return address and my_x were above
The main code stores the 20 into "result".

As you can see, there's no absolute address used, everything is relative to top of stack.

This is nice, and some programming languages don't have heap and pointers stuff, just stack and they live well with it. C/C++ is more low level thing, so it's not enough.
Back in x86 days stack was usually limited to 64kB, so doing:

```
void foo() {
  int my_data[100000];
}
```

would not work, exceeding the limits of stack space. Even nowadays the stack is much smaller then heap.
But you have 640kB of RAM, right? (at least Bill told so) And you want to use it, you want just 100k of data. That's what heap is for, you reserve junk of 100k values out of it easily.

Stack is generally for returning addresses between functions calling and little amount of local variables to make things neat and tidy for CPU and execute it fast.

But that means you have to reserve/release big chunks of memory from OS's heap by some means, and that needs absolute addressing, because the OS can't be sure it will have free block of memory on the same relative spot to your code, like always.
That's where pointers come into play:

```
void foo() {
  int *mydata = new int[100000];   //reserve heap memory
  //on local stack only the pointer is allocated (32 or 64bits),
  //the 100k bunch of ints is reserver in heap.

  mydata[32768] = 10;    //work with the heap memory trough pointer

  delete [] mydata;  //release the heap memory by it's pointer
  //if you will not, it will be lost, because pointer to it is on local stack
  //which will be lost after function return => memory leaking
}
```

As you can see, pointers are not popular, because if you use them for allocating heap memory (and that's the most classic usage of them), and you lost the pointer, you can't release the memory later -> it is lost. (OS will recover it after your application does exit, but that's considered bad practice anyway)

... fast forward to modern times and U++ ...
the U++ way of avoiding pointers (and thus common mistakes of dereferencing uninitialized pointer, or not release resources) is to put as much as possible somewhere where it belongs.
I.e. if your widget needs int counter, it's member of it's class. That widget is used by main window, so whole widget is member of main window. Main window is allocated on local stack of main like: MainWindowClass window; (which allocated all the stuff inside in one big chunk on stack, like also that widget with it's counter).
So having int counters[100000]; as class member in widgets would hurt the stack, right? Do you need pointer then?
Depends what you are doing, if you are doing high performance stuff with static aligned memory

pools, you may be better with pointer + manual management of resources.
If you have dynamic app which may need 10k, 30k or 100k of values from some external file, use NTL container instead.
That will make only the container management to occupy the stack space, and the container code will handle the dynamic heap memory allocation and releasing (trough properly written&called destructors of containers).
Like member of class is: Vector<int> counters;
Then in the member function you can do:
counters.Reserve(100000);
counters[32768] = 20;

As you see, no direct pointers and direct explicit memory management.

You just have to design your code and classes in a way that everything belongs somewhere and it's lifetime span is well defined by existence of parent object.
Like the "my_x" is released at the end of ExplainStack1 function implicitly, so C++ destructors and proper usage of classes will make this naturally happening for you.

Pointers are of course very handy whenever you need to address something in absolute way, no matter if you did allocate that resource by yourself, or not. (like accessing video ram of text mode at 0xb800:0000 from C by directly writing byte values there in DOS days)

... I hope this makes some sense, why pointers exists, and why they are source of many coding mistakes and more then a decade was spent to create libraries which help you to NOT use pointers (like STD, NTL and whole U++).

Now yet another Java vs C++ difference to answer your WHY:
void foo(int buffer[100000]) { ... function code ... }

In Java the function does get pointer to buffer which is automagically allocated in heap memory, so the stack is not burdened by push/pop of 100k data and killing the cache on CPU.
This is all hidden in Java, and as programmer you don't have to think about that, you put 100k buffer into function definition, and the java machine will put there just pointer (reference) to actual data.
In C++ it will do what you ask from it, i.e. kill the stack, cache and eventually OS too.
(well, the syntax I did use would actually lead to pointer even in C++ IMHO, but if it would be huge struct, it would do the ugly thing anyway, so I hope you get the idea)
So in C++ you can define the function as:
void foo(int *buffer) { ... function code ... }
And on the stack only pointer will be stored, which you can dereference later for actual values in buffer.

This naturally leads us to C++, classes and references:
1) void foo(WindowClass window) { ... }
2) void foo(WindowClass * window) { ... }
3) void foo(WindowClass & window) { ... }

1) upon calling with some w1 object will CLONE it on the stack space (if WindowClass is memory

huge, you are asking for trouble), and give the function the clone (done by copy constructor) to play with.
2) you have to call it with &w1 (address of w1 object), it will dereference it to have access to actual w1 (not it's copy!).
You can also call it with "NULL", which will make you crash upon dereferencing it.
3) (references - like Java magic) you do call it with foo(w1), but only pointer is passed down on stack. In function you don't have to dereference it as pointer, you work with it like with instance, i.e. window.bar(); What you do with it will affect the upper w1, it's not a copy of it. You can't pass NULL, only valid "WindowClass" instance, which is checked during compilation time.

I.e. references are nice to use when you want to affect actual value of passed argument, or you want to save the stack space and the passed object would be too huge. But you don't want pointers because you don't want to test for NULLs, etc.

Pointers are nice when you have to be fully dynamic, like maybe you get that object because user did want it, or maybe you get just NULL, because you should work without it.
