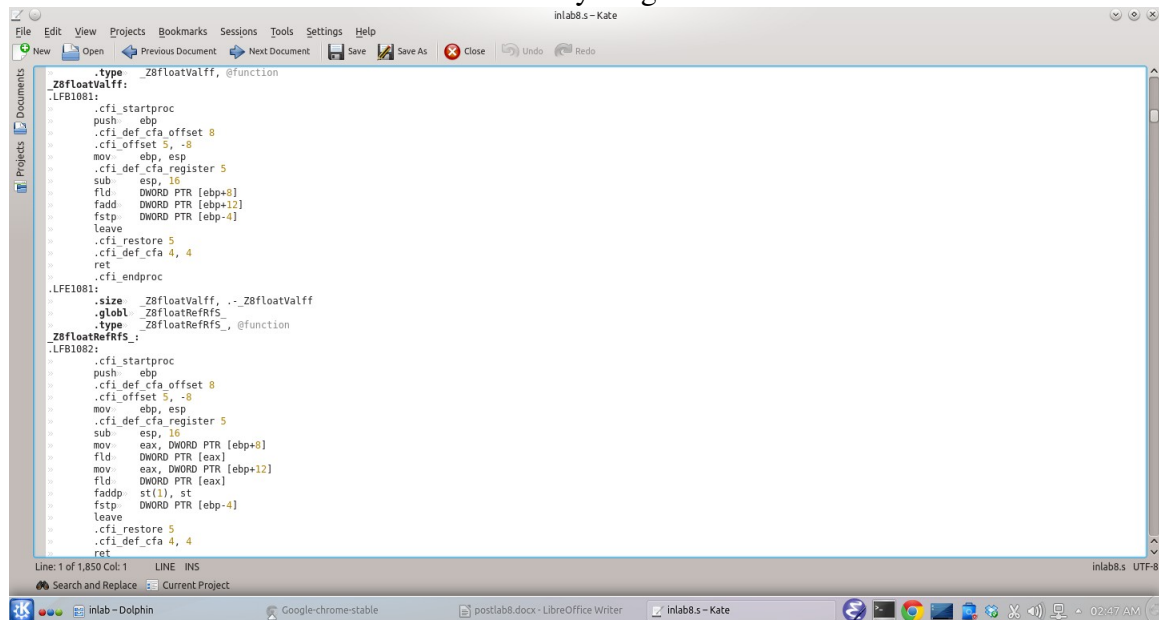


Maddie Stigler  
mgs4ff  
11/4/14  
postlab8.pdf

## Parameter Passing

In order to examine how ints, chars, pointers, floats and objects that contain more than one data member are passed into subroutines, I created different functions that took in these data types as parameters and performed small transitions to them before returning back to the main method.

Passing ints, chars, and floats into the callee all functioned very similarly. The caller saved the stack pointer before pushing the parameters on the stack and calling the callee. However, there were some differences within the callee subroutine that differed due to whether the parameter was passed by value or by reference. There was a difference between passing by value and passing by reference when passing char data types. When passing a char by value, more of the x86 code pushes parameters onto the stack using the mov and push calls. When passing by reference, addresses of the parameters are stored on the stack and the code for passing by reference tends to be longer and spends time looking up the address for the referenced variable. In fact, this is also the case when passing ints and floats as well. Another difference just between passing the different data types was that the subroutine spent more time on the float parameter using fladd, fld, and fst. These instructions did not appear in the int and char methods because it didn't need to convert anything for them.



```
.type _Z8floatValff, @function
.LFB1081:
.cfi_startproc
push    ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
mov     ebp, esp
.cfi_def_cfa_register 5
sub     esp, 16
fld     DWORD PTR [ebp+8]
fadd    DWORD PTR [ebp+12]
fstp    DWORD PTR [ebp-4]
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc

.LFE1081:
.size _Z8floatValff, .-_Z8floatValff
.globl _Z8floatRefRf5
.type _Z8floatRefRf5, @function
.LFB1082:
.cfi_startproc
push    ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
mov     ebp, esp
.cfi_def_cfa_register 5
sub     esp, 16
mov     eax, DWORD PTR [ebp+8]
fld     DWORD PTR [eax]
mov     eax, DWORD PTR [ebp+12]
fld     DWORD PTR [eax]
faddp   st(1), st
fstp    DWORD PTR [ebp-4]
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
```

I also examined the difference between passing a pointer and passing a reference. When comparing passing a pointer into the subroutine with passing a reference into the subroutine, both consume the same amount of code and use the same code to achieve the same result. This suggests that neither is more efficient than the other. Both rely on the eax and edx registers to store DWORD PTR values and move them before exiting and

returning the result to the main method. This suggests that determining which way to pass a value depends on the context of the code, not that one is more or less efficient than the other.

To demonstrate how objects that contain more than one data member are passed into subroutines, I used the c++ list. When passing by value, the x86 code used a DWORD PTR to store the values of the list and move them into the eax register and manipulate them from there. When passing by reference, the x86 code delivers the same result. They are both stored as DWORD PTR in the eax register and manipulated the same way.

In order to replicate passing an array in C++ to x86, I created an array of 10 elements and passed it into a function that simply assigned 5 to array at 0. The x86 code representing the callee accessed the array parameters similarly to how it would access a list or multiple parameters. It adjusted the ebp register to the start of the array and began accessing parameters from there. When calling the callee, the caller simultaneously pushed the array values onto the stack, thus making them accessible to the callee.

As for the interaction between the caller and callee within this program, the caller invokes the callee by using lea and mov before calling it. This saves the parameters and allows the callee access to them. After the callee returns, the caller saves the variables again and alternates between using lea with the eax register and using mov with the DWORD PTR and esp to increment the stack pointer. This remains pretty consistent throughout all data types. However, with the list parameter, the caller simultaneously calls the callee while performing a push\_back, storing the list's parameters on the list.

Ultimately, while the x86 code is certainly not as easily understood as the c++ code, when comparing the two, it is easy to tell that one stems from the other. There were a couple of instructions I didn't understand, but by looking at my cpp program and comparing to the assembly, I was able to piece together what the instructions meant. Most of the caller and callee conventions are similar. While there are slight differences between certain data types, the caller still continues to push the stack point and mov values onto the stack before invoking the callee and the callee continues to pop its' stack values before returning to the main method.

## Objects

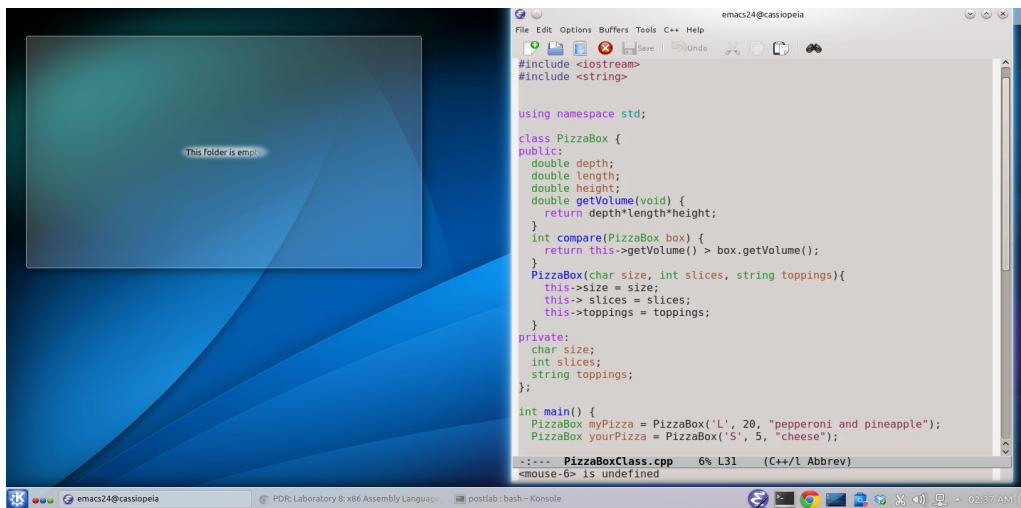
In order to examine objects in assembly, I created a C++ class PizzaBox with 5 fields and two functions. Each PizzaBox object has 3 public double fields: depth, length, and height. Each one also has 3 private fields: char size, int slices, and string toppings. The two public functions are compare that takes in another PizzaBox as a parameter and getVolume that returns the volume using its public fields. I also wrote a constructor for the PizzaBox object that initializes its private fields.

In the assembly for C++, the data for the class is initially skipped over and the assembly code begins with the first function, getVolume(), followed by the next function, compare(PizzaBox), and the constructor. Following are the two PizzaBox objects and the main method. The getVolume() function accesses the fields of the PizzaBox by pushing the object onto the stack, passing its address, and then accessing its various stored fields

using an offset. The compare function pushes onto the stack, calls getVolume, also uses st(#), st to compare values. The PizzaBox constructor stores fields on the stack and calls certain fields as subroutines to the stack. The assembly uses the stack to contain values of fields and objects while also relying on labels to store things such as functions and some aspects of the data objects to access variables through the calling convention.

In respect for the data layout for my sample PizzaBox C++ class, when compiled with g++ and converted into assembly, the data objects are all stored sequentially beginning with the two functions followed by the main method call. The function and main respective data members all reside within the data layout associated with the data objects. One thing I noticed was that in declaring fields public in comparison to private, there seemed to be no difference in the assembly. This leads me to think there would be no effect on how the fields are laid out in memory. Another interesting aspect of the data layout during function calls is what happens when an object is passed in as a parameter. In my compare(PizzaBox) function, a PizzaBox object is passed in as a parameter to be compared to the PizzaBox the function is being called on. When this occurs, in assembly, the code uses an offset to access the parameter's address and fields. While the parameter itself isn't detailed until further along in the assembly, the assembly code already has a way to access its data.

When considering how data members are accessed from inside a member function as compared to from outside a member function, I used the compare function to illustrate both. The compare function utilizes a "this" pointer to access the volume of the caller and compares it to the volume of another PizzaBox object. The compare function uses two different objects and also calls a separate function within itself. First, the "this" pointer is displaced by an offset so it can access the PizzaBox in memory. Then, the getVolume method is called on the pointer and stored in a register. Next, the second PizzaBox object is accessed using an offset of the stack and its volume is found the same way. Data members are accessed the same regardless of whether they are inside the member function or outside the member function. The following screenshots illustrate the "this" pointer.

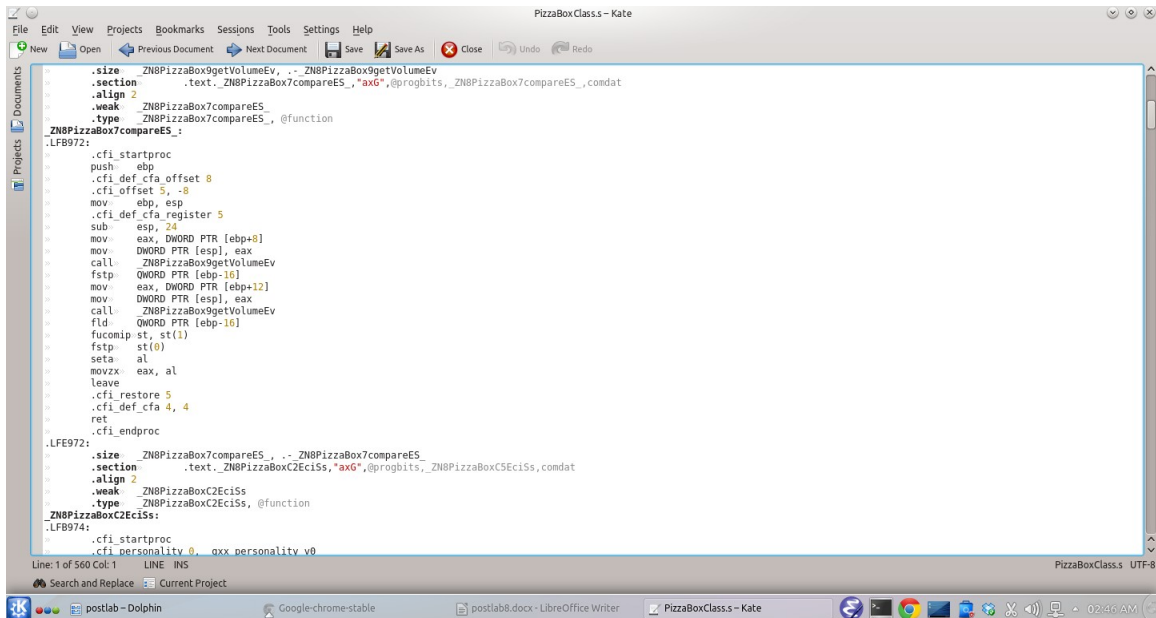


```
#include <iostream>
#include <string>

using namespace std;

class PizzaBox {
public:
    double depth;
    double length;
    double height;
    double getVolume(void) {
        return depth*length*height;
    }
    int compare(PizzaBox box) {
        return this->getVolume() > box.getVolume();
    }
    PizzaBox(char size, int slices, string toppings){
        this->size = size;
        this->slices = slices;
        this->toppings = toppings;
    }
private:
    char size;
    int slices;
    string toppings;
};

int main() {
    PizzaBox myPizza = PizzaBox('L', 20, "pepperoni and pineapple");
    PizzaBox yourPizza = PizzaBox('S', 5, "cheese");
    // ... PizzaBoxClass.cpp 6% L31 (C++/L Abbrev)
    // mouse-6> is undefined
```



```
.size _ZN8PizzaBox9getVolumeEv, .- _ZN8PizzaBox9getVolumeEv
.section .text._ZN8PizzaBox7compareES_, "axG", @progbits, _ZN8PizzaBox7compareES_, comdat
.align 2
.weak _ZN8PizzaBox7compareES_
.type _ZN8PizzaBox7compareES_, @function
_ZN8PizzaBox7compareES_1:
.LFB972:
.cfi_startproc
push    ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
mov     ebp, esp
.cfi_def_cfa_register 5
sub     esp, 24
mov     eax, DWORD PTR [ebp+8]
mov     DWORD PTR [esp], eax
call    _ZN8PizzaBox9getVolumeEv
fstp    QWORD PTR [ebp-16]
mov     eax, DWORD PTR [ebp+12]
mov     DWORD PTR [esp], eax
call    _ZN8PizzaBox9getVolumeEv
fld     QWORD PTR [ebp-16]
fucomip st, st(1)
fstp    st(0)
seta    al
movzx   eax, al
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE972:
.size _ZN8PizzaBox7compareES_, .- _ZN8PizzaBox7compareES_
.section .text._ZN8PizzaBox7compareES_, "axG", @progbits, _ZN8PizzaBox7compareES_, comdat
.align 2
.weak _ZN8PizzaBox7compareES_
.type _ZN8PizzaBox7compareES_, @function
_ZN8PizzaBox7compareES_:
.LFB974:
.cfi_startproc
.cfi_personality 0, _qxx_personality_v0
Line: 1 of 560 Col: 1
```

When examining the implementation, storage, access, passing, and updating of the this pointer, I look at the constructor for my PizzaBox object as well as the compare(PizzaBox) function. Both make use of the this pointer in order to access current fields and outputs from the getVolume() method. The pointer is accessed and stored through the esp register which sends the memory to the two functions to be able to access the pointer's fields and value. In this sense, the pointer is implemented through the stack as well. The assembly also doesn't show that the pointer is being updated. This may be because the functions are not modifying the fields and functions of the pointer, but using them to access a different value.