Yubin Lim yl2vv

postlab9.pdf

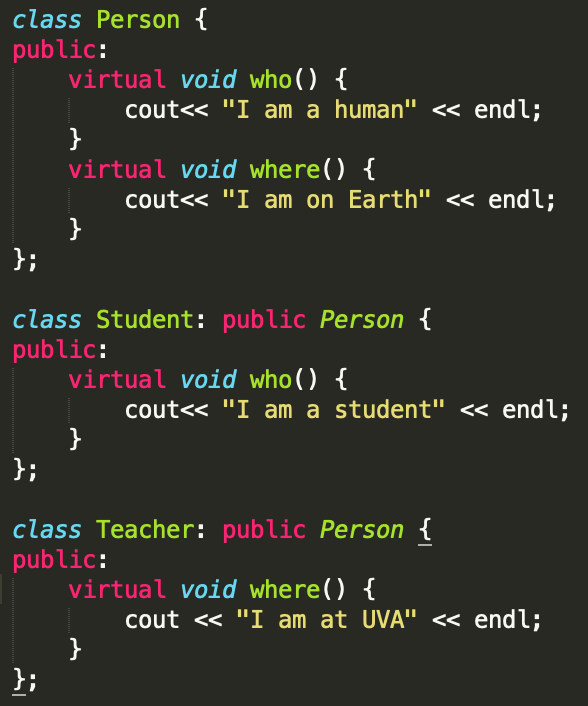
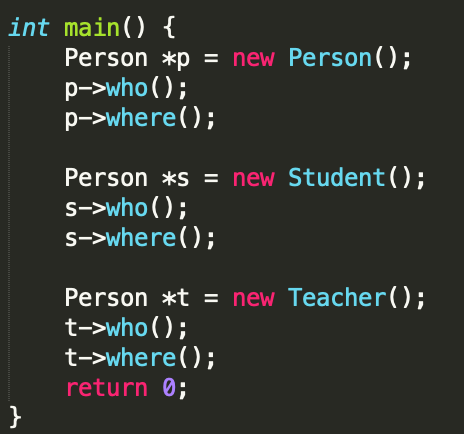
April 11, 2019

CS2150-103

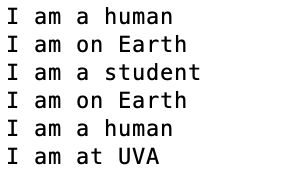
**Dynamic Dispatch**

For dynamic dispatch the decision on which member function to invoke is made using run-time type of an object. This is other type of dispatch is static dispatch, where the decision on which member function to invoke is made using compile-time type of an object. A difference between Java and C++ is that Java uses dynamic dispatch by default, that is, everything is virtual, while C++ supports both types of dispatch. Virtual methods are only needed under certain circumstances, specifically when you want to use polymorphism. The syntax use in C++ for dynamic dispatch is the “virtual” keyword. By using “virtual”, dynamic dispatch is implemented with the use of virtual method tables. The C++ compiler inserts a virtual method table for every class having virtual function or class inherited from the class that has a virtual function. In essence, each object has a pointer to the virtual method table. That table has the addresses of the methods. Any virtual method must follow the pointer to the object, then follow the virtual method table pointer. It proceeds to look up the method pointer and jump to that method.

As an example, I created three classes: Person superclass, and two subclasses called Student and Teacher (as shown by the picture below). I used the virtual keyword to make the functions use dynamic dispatch. In the main method, I created a Person pointer p pointing to Person, a Person pointer s pointing to Student, and a Person pointer t pointing to Teacher.

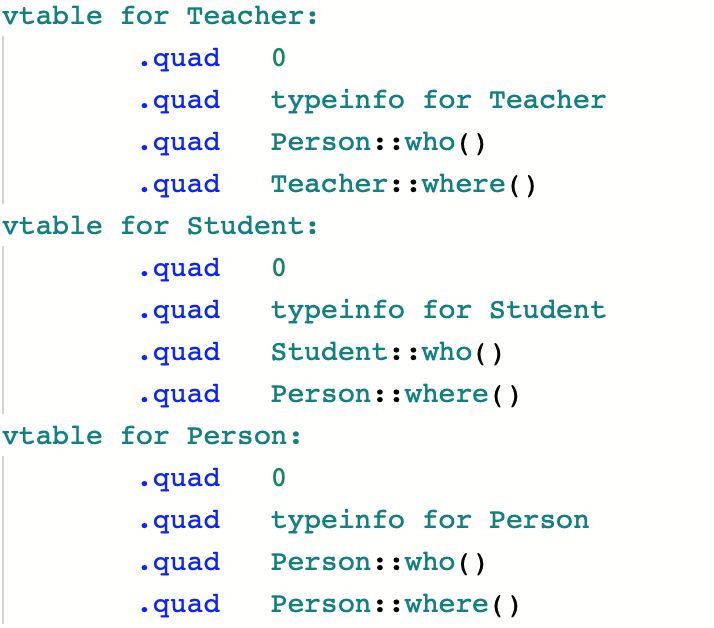
 

Per expected, the code printed the following:

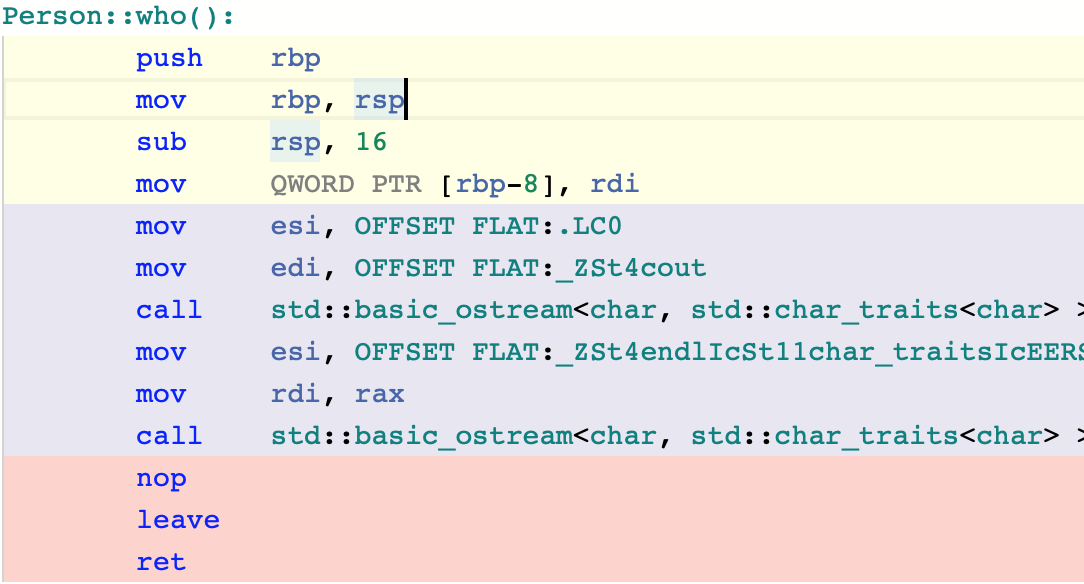
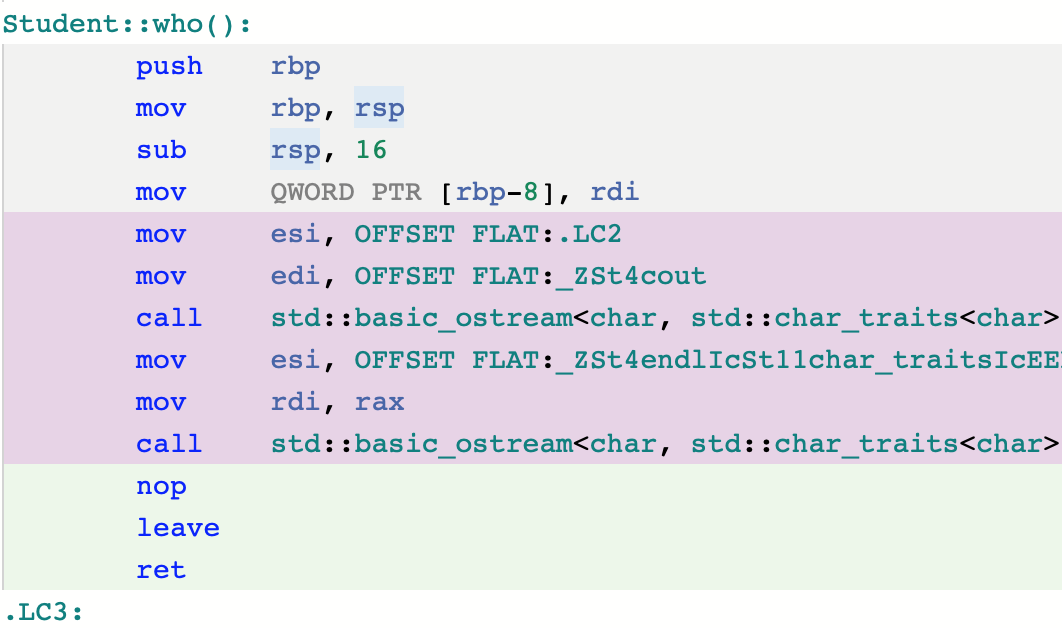


The code was able to decide what member function to call during run time. For example, as the Student Class inherited the Person Class. The vtable of Student Class contains pointer to the who() function of Student class (“I am a student”), but the pointer to where() is of base class, the Person class (“I am on Earth”).

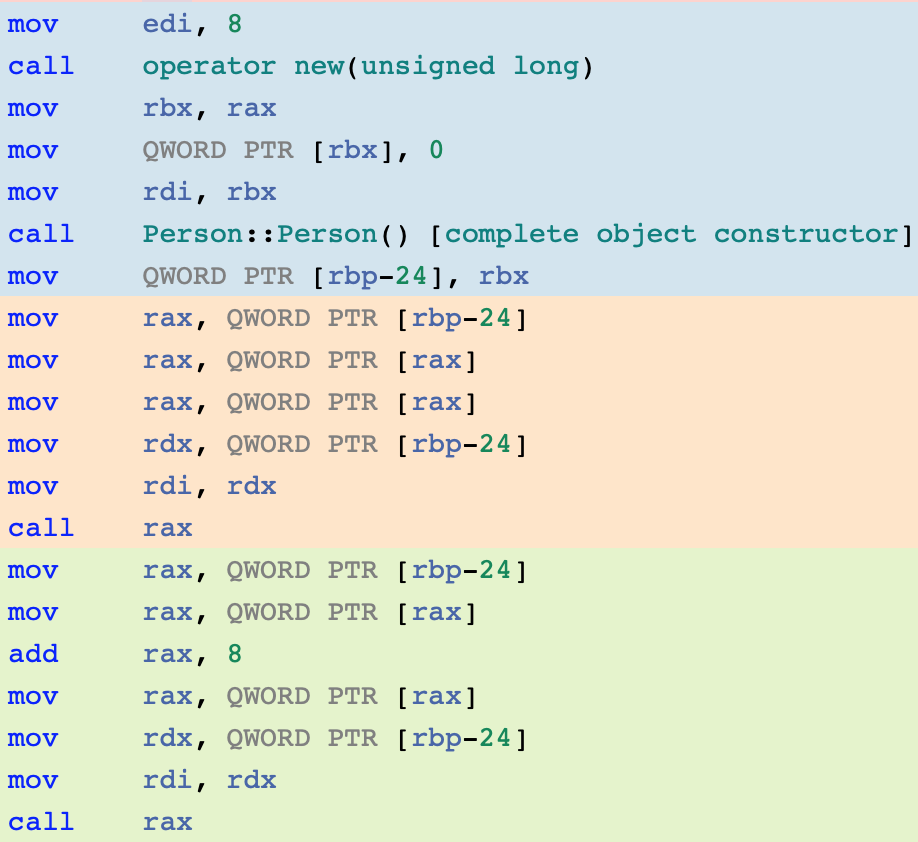
I used godbolt.org to convert my C++ code into assembly. The assembly gave a vtable, which is the virtual table that contains the pointers. The vtable contains the addresses of the appropriate methods as well as the pointers to metadata. As shown in the table, in the virtual table for Teacher, the function who() is called from the Person class while the function where() is called from the Teacher class. In the virtual table for Student, the function who() is called from the Student class while the function where() is called from the Person class.

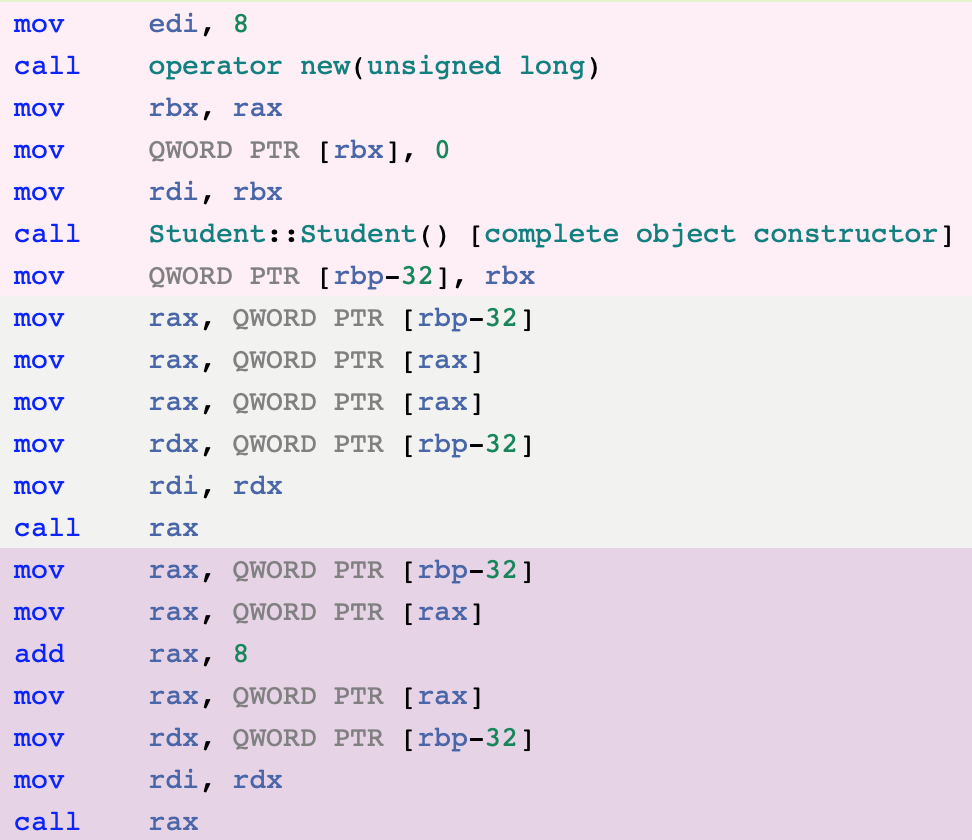


The assembly code for the Person::who() and the Student::who() looks almost identical, it is the difference in the virtual table. QWORD PTR shows the pointer in dynamic dispatch.

The assembly code from the main method is shown below. The first picture is from the pointer p and second picture is the pointer s. The call to the first function, who() has the virtual pointer to the appropriate function. The call to the second function where() has a line that wasn’t in the who() function. It is the assembly line “add rax, 8” which shows that since we are calling the second overridden method, we access an offset of the virtual method pointer by 8 bytes.





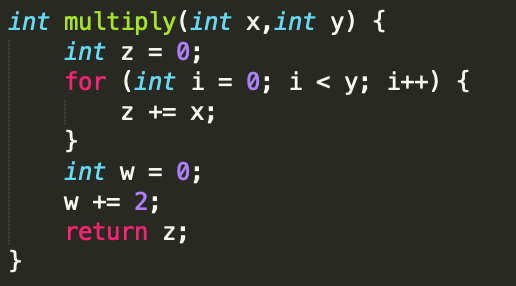
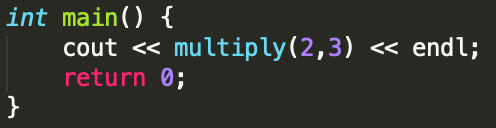
**Sources:**

* https://lukasatkinson.de/2016/dynamic-vs-static-dispatch/
* https://en.wikipedia.org/wiki/Dynamic\_dispatch
* https://pabloariasal.github.io/2017/06/10/understanding-virtual-tables/

**Optimized Code**

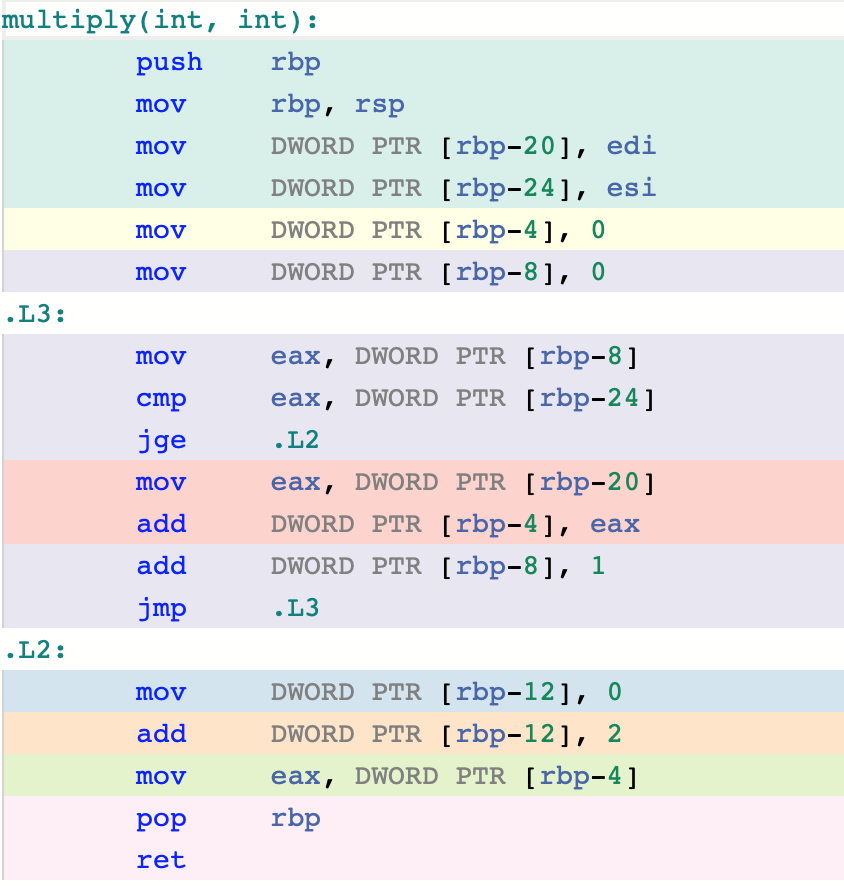
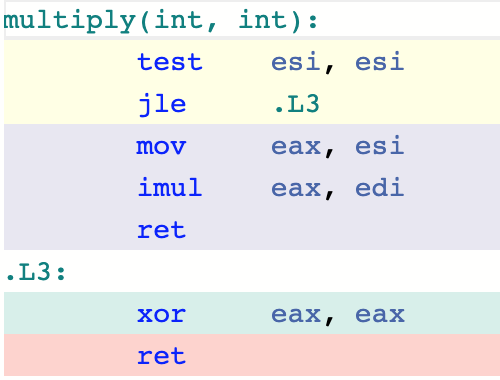
In C++, one can generate optimized code with the –O2 compiler flag. The -O2 flag sets the options that create the fastest code in the majority of cases. –O2 increases both compilation time and the performance of the generated code.

To compare code generated normally to optimized code, I created a function called multiply and a main method that calls the function.

The multiply function takes in two parameters: an integer x and an integer y. Instead of multiplying the two numbers, the function takes the use of a for loop that adds the integer “x” y number of times, so the returning value would be equal to multiplication. At the end of the code, I initialized an integer w and added two to it. This unrelated part was added to see how it would affect my assembly language with optimization. If an instruction is not necessary to the final output of the program, then it may be optimized away completely.

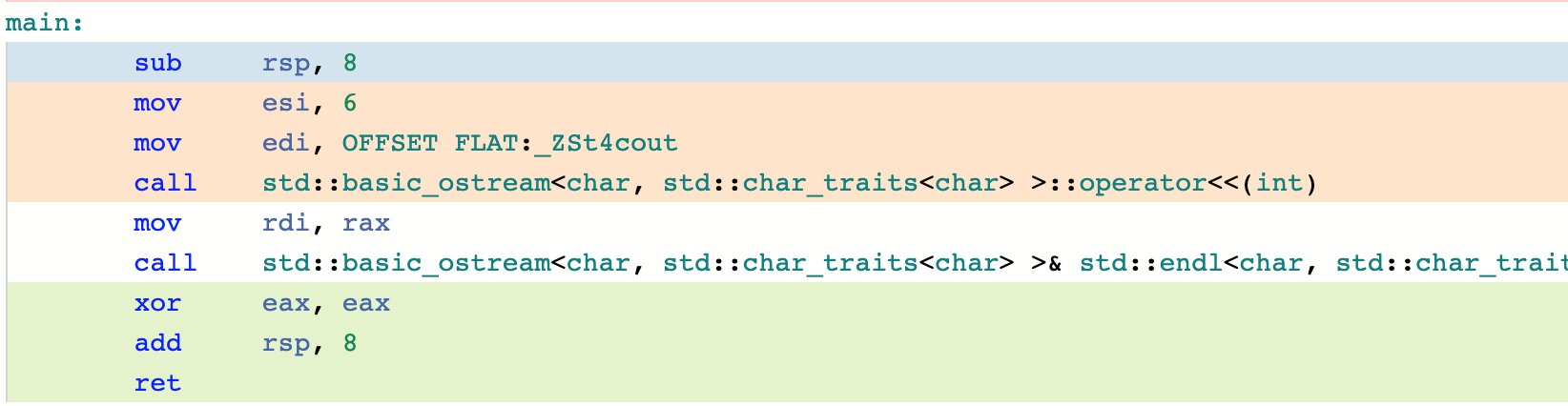
I used godbolt.org to convert my C++ code into assembly language. By just skimming the code, I already noticed a massive change with the use of the –O2 flag, as the one without the optimization flag contained twice as many lines compared to the code with the addition of the flag. It used less code and more efficient code. The two images below are the multiply function. The left is compiled without any compiler options and the right takes use of the –O2 flag.

The results were significant, because by using optimization, the compiler recognized that the purpose of the function is to multiply two numbers and used “imul” instead of looping and adding the first number each time. Branching is not ideal, as it slows down the execution speed of a program. Because a large percentage of the time is spent inside loops, loop optimization has a significant impact. Another difference was that the optimization got rid of the useless code I made with “int w”. The function on the left has a third step called “.L2:” while the one on the right ignored the lines that doesn’t affect the return of “z”. There are no pointers or any memory access with push and pop in the optimized code.

A difference in the main method is shown below.





As expected, the picture on the top is the assembly code without optimization. With the use of optimization, the push and pop were gone, reducing memory access. The optimized code uses fewer x86 instructions.

Optimizing code can be very important, as it can save memory and time. It minimizes the power consumed by a program.

**Sources:**

* https://unix.stackexchange.com/questions/452337/what-does-the-o2-option-for-gcc-do
* https://www.quora.com/What-is-O2-flag-compile-How-do-I-do-it
* https://en.wikipedia.org/wiki/Optimizing\_compiler