Storing an Inheritance

Due Tuesday, February 25 at 8 a.m.

CSE 1325 - Spring 2020 - Homework #5 - 1 - Rev 2

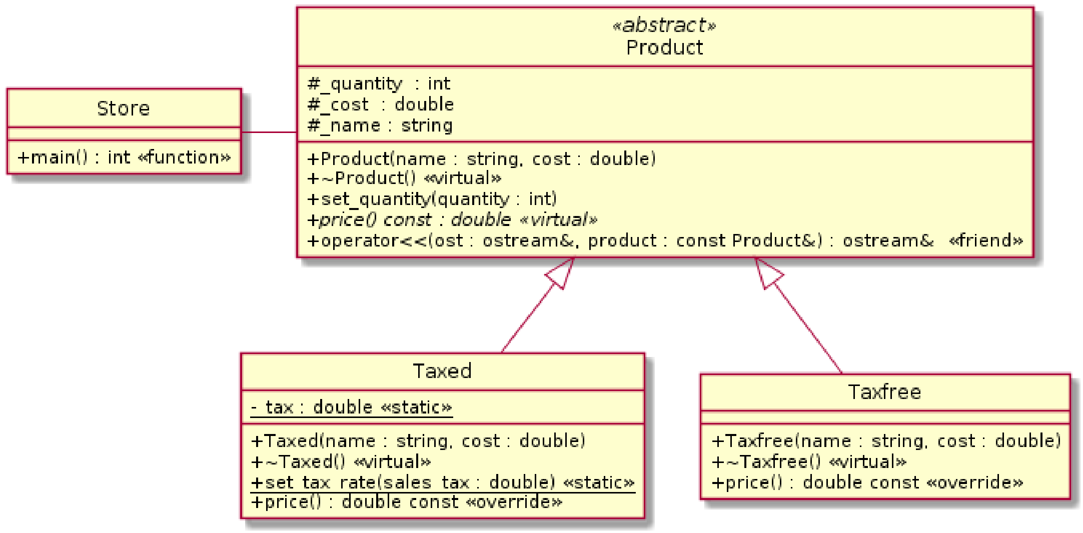
Assignment Overview

Inheritance is one of the three foundations of object-oriented programming, so let's dip our toes gently into the waters with a trip to the local grocery store.

Full Credit

In your git-managed Ubuntu Linux 18.04 directory **cse1325/P05/full_credit**, write an *abstract* Product class matching the following UML class diagram to represent items for sale in a grocery store. (NOTE: A class is *abstract* if it includes at least one *pure virtual* method, in this case, *price*.) Then derive two classes from it - Taxed and Taxfree - representing products for which sales tax is or is not due, respectively.

Note that we've used stereotypes to *emphasize* virtual class members, abstract classes, virtual methods, and overrides in this diagram, but this will not be the case for future diagrams. You should be able to recognize what needs to be virtual, static, and overridden from context and typographic indicators such as italics and underline.



«abstract»
Product
#_quantity : int
#_cost : double
#_name : string
+Product(name : string, cost : double)
+~Product() «virtual»
+set_quantity(quantity : int)
+price() const : double «virtual»
+operator<<(ost : ostream&, product : const Product&) : ostream& «friend»
Store
+main() : int «function»
Taxed
- tax : double «static»
+Taxed(name : string, cost : double)
+~Taxed() «virtual»
+set_tax_rate(sales_tax : double) «static»
+price() : double const «override»
Taxfree
+Taxfree(name : string, cost : double)
+~Taxfree() «virtual»
+price() : double const «override»

The attributes or fields of Product, all of which are immutable (i.e., can't be changed after construction) except _quantity, are as follows:

- **_name** is just a string representation of the product that is for sale, e.g., "Milk" or "Apple".
- **_cost** is the amount of cash in pre-tax dollars necessary to buy the item. It's a double, which we don't use for money in the real world due to rounding errors. But this isn't the real world, is it? :-)
- **_quantity** is a mutable int representing how many are to be purchased.

The executable members of Product are as follows:

- The **constructor** should initialize _name and _cost to the matching parameters, and _quantity to 0. Throw a runtime exception if _cost is negative.

- The **destructor** is always virtual for a base class, but has an empty body. We have not resources that need to be cleaned up when Product deleted.
- **set_quantity** simply assigns its parameter to the _quantity attribute.
- **price** is pure virtual (it has no implementation in this class) and const (it promises not to modify any attributes). It will be overridden in the derived classes.
- **Overload operator<< for Product**, streaming out e.g., "Cheese ($0.99)" if _quantity is 0 or e.g., "Cheese (2 @ $0.99)" if quantity is positive. *Always* show 2 digits to the right of the decimal point for money. Note that since Taxed and Taxfree are derived from this class, they will take advantage of this overload as well!

Taxed adds one additional attribute and one additional method.

- **_tax** is the sales tax rate for this product, e.g., 0.0825. This attribute is *static*, that is, it is a single variable in one memory location for **all** instances of Taxed. Remember that a static attribute must be declared in taxed.h but also separately defined in taxed.cpp.
- The **constructor** simply delegates to Product's constructor. Remember to use parentheses.
- The **destructor** is virtual and has an empty body.
- **set_tax_rate** is a static method that simply assigns its parameter to the _tax static attribute. Remember, static methods can be called from the class without an object, e.g., `Taxed::set_tax_rate(0.0725)`.
- **price** is const, overrides *Product::price()*, and returns the total prices as _quantity x _cost x (1+_tax).

Taxfree adds no additional attributes or methods.

- The **constructor** simply delegates to Product's constructor. Remember to use parentheses.
- The **destructor** is virtual and has an empty body.
- **price** is const, overrides *Product::price()*, and returns the total prices as _quantity x _cost.

(At this point, you may be asking why we didn't simply use a bool to indicate whether a product requires collecting sales tax or not. The answer is: For the educational benefit, of course. You need to practice inheritance!)

Write a main function that predefines several taxed and taxfree grocery items. In the main loop:

- List all of the taxable and taxfree items you predefine (at least 3 of each).
- Ask the user to select an item with an associated quantity.
- **Perform data validation on their input.**
    - If invalid, throw an exception (which you must catch and handle) on e.g., negative quantities or undefined grocery items, then restart the loop.
    - If valid, add the selection to their "cart" (e.g., vectors in main).
- List of both sets of items in the cart and a cart total price (sum of all price() methods).
- Then allow them to "buy" the items, e.g., by entering a quantity of 0 or via a menu selection.

You can find out which products require sales tax in the first paragraph of https://comptroller.texas.gov/taxes/publications/96-280.pdf.

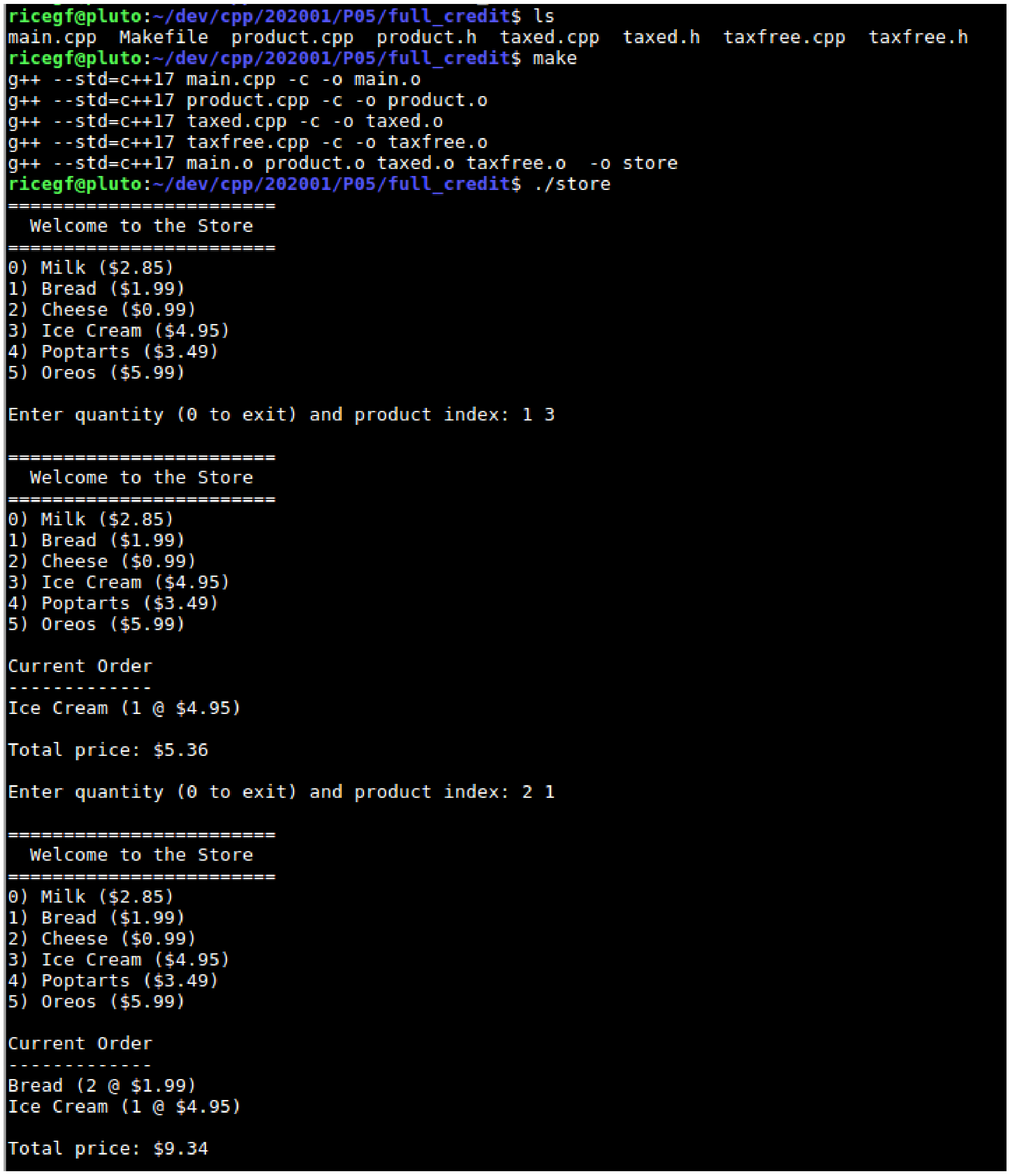
You MUST write a valid Makefile as always.

- `make clean` deletes (at a minimum) your .o and .gch files and your executable(s).

- `make` builds the executable into an executable, with separate compilation of each class with e.g., `make product.o`.

Add, commit, and push all files.

Example Full Credit Output



```
ricegf@pluto:~/dev/cpp/202001/P05/full_credit$ ls
main.cpp  Makefile  product.cpp  product.h  taxed.cpp  taxed.h  taxfree.cpp  taxfree.h
ricegf@pluto:~/dev/cpp/202001/P05/full_credit$ make
g++ --std=c++17 main.cpp -c -o main.o
g++ --std=c++17 product.cpp -c -o product.o
g++ --std=c++17 taxed.cpp -c -o taxed.o
g++ --std=c++17 taxfree.cpp -c -o taxfree.o
g++ --std=c++17 main.o product.o taxed.o taxfree.o  -o store
ricegf@pluto:~/dev/cpp/202001/P05/full_credit$ ./store
========================
  Welcome to the Store
========================
0) Milk ($2.85)
1) Bread ($1.99)
2) Cheese ($0.99)
3) Ice Cream ($4.95)
4) Poptarts ($3.49)
5) Oreos ($5.99)

Enter quantity (0 to exit) and product index: 1 3

========================
  Welcome to the Store
========================
0) Milk ($2.85)
1) Bread ($1.99)
2) Cheese ($0.99)
3) Ice Cream ($4.95)
4) Poptarts ($3.49)
5) Oreos ($5.99)

Current Order
-------------
Ice Cream (1 @ $4.95)

Total price: $5.36

Enter quantity (0 to exit) and product index: 2 1

========================
  Welcome to the Store
========================
0) Milk ($2.85)
1) Bread ($1.99)
2) Cheese ($0.99)
3) Ice Cream ($4.95)
4) Poptarts ($3.49)
5) Oreos ($5.99)

Current Order
-------------
Bread (2 @ $1.99)
Ice Cream (1 @ $4.95)

Total price: $9.34
```

Bonus

In Lecture 06, we discuss error handling and testing. You've demonstrated your skills with exceptions, so let's try a bit of testing!

Write a simple regression test (that is, a second main() function in file test.cpp) that performs 4 tests on our class hierarchy.

- **Instance a Taxfree product, and verify that each of its attributes (both inherited and local) were properly constructed and that operator<< (Product's friend) works with it.**

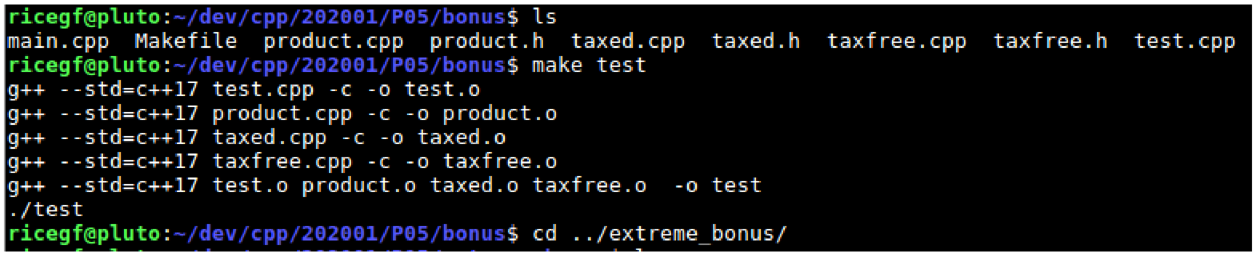
    - With that or a different instance of a Taxfree product, set the quantity to a non-zero value. Verify that the price is correctly calcuatled.

- Do the same two tests for a Taxed product. Remember that you *must* set the static attribute _tax prior to calculating any _price.

Remember: **Regression tests are fully automatic - no human interation! - and print NOTHING unless an error is found.** If one or more errors are found, report each one clearly but concisely on one or more separate lines. If any test fails, the last thing the test should print is simply "FAIL" on a separate line, but if all tests pass, print nothing at all.

Some examples for inspiration or derision (as appropriate) may be found at Lecture 06 slide 51 and **cse1325-prof/06/code_from_slides/test_area.cpp**.

Add, commit, and push all files.



```
ricegf@pluto:~/dev/cpp/202001/P05/bonus$ ls
main.cpp  Makefile  product.cpp  product.h  taxed.cpp  taxed.h  taxfree.cpp  taxfree.h  test.cpp
ricegf@pluto:~/dev/cpp/202001/P05/bonus$ make test
g++ --std=c++17 test.cpp -c -o test.o
g++ --std=c++17 product.cpp -c -o product.o
g++ --std=c++17 taxed.cpp -c -o taxed.o
g++ --std=c++17 taxfree.cpp -c -o taxfree.o
g++ --std=c++17 test.o product.o taxed.o taxfree.o  -o test
./test
ricegf@pluto:~/dev/cpp/202001/P05/bonus$ cd ../extreme_bonus/
```

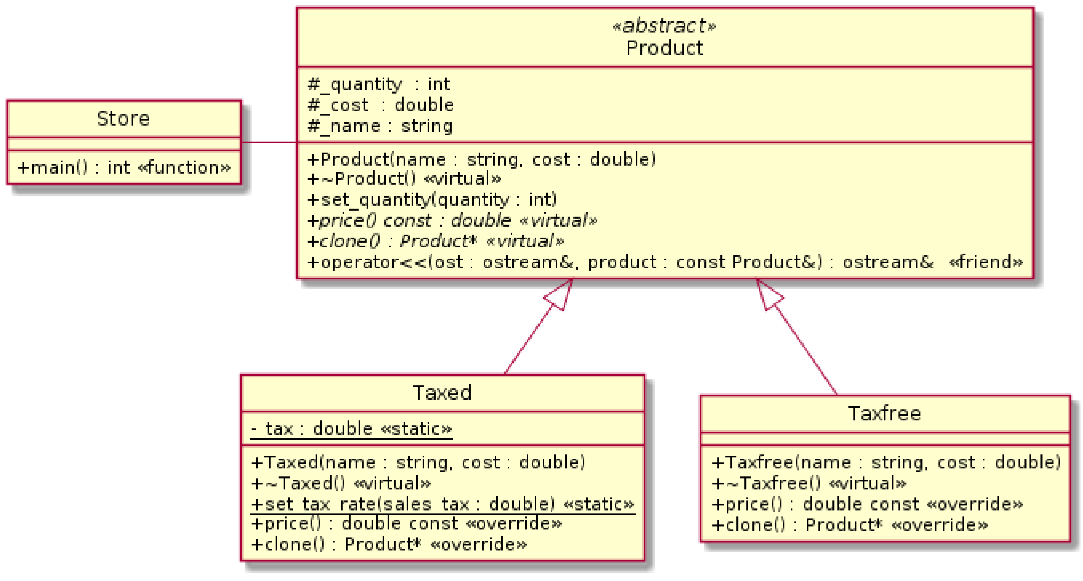
Extreme Bonus

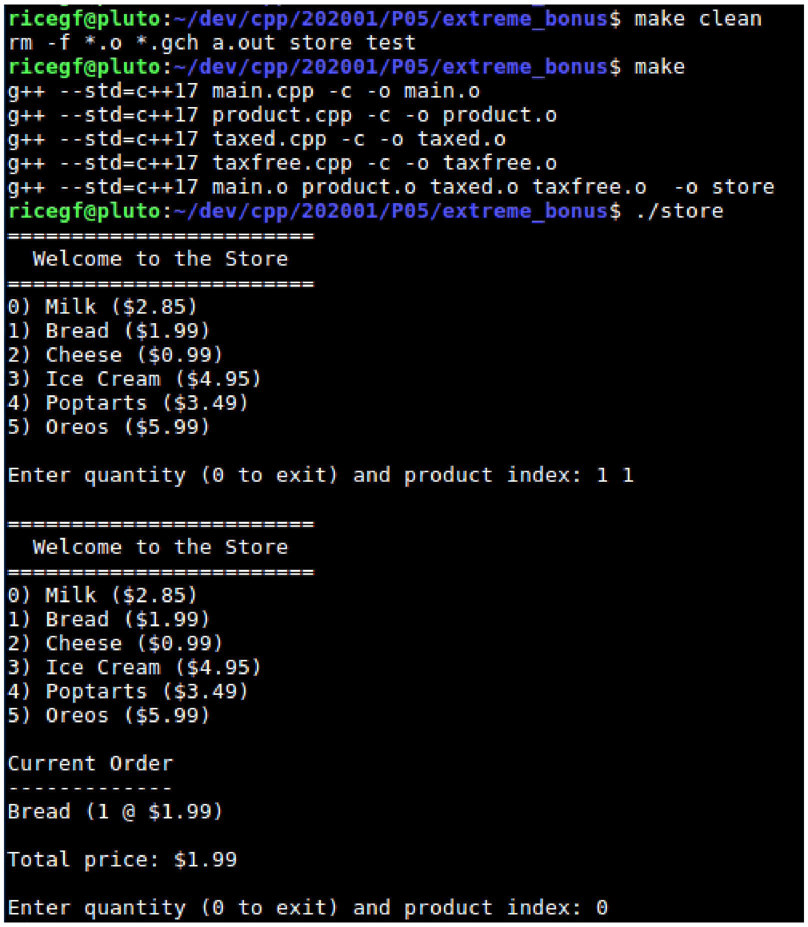
Replace the separate vectors in each of your main programs with a single vector of type Product*. Use polymorphism to invoke each method.

Include screenshots or other output of meld, cloc, diff, or a similar tool demonstrating how this simplifies your implementation.

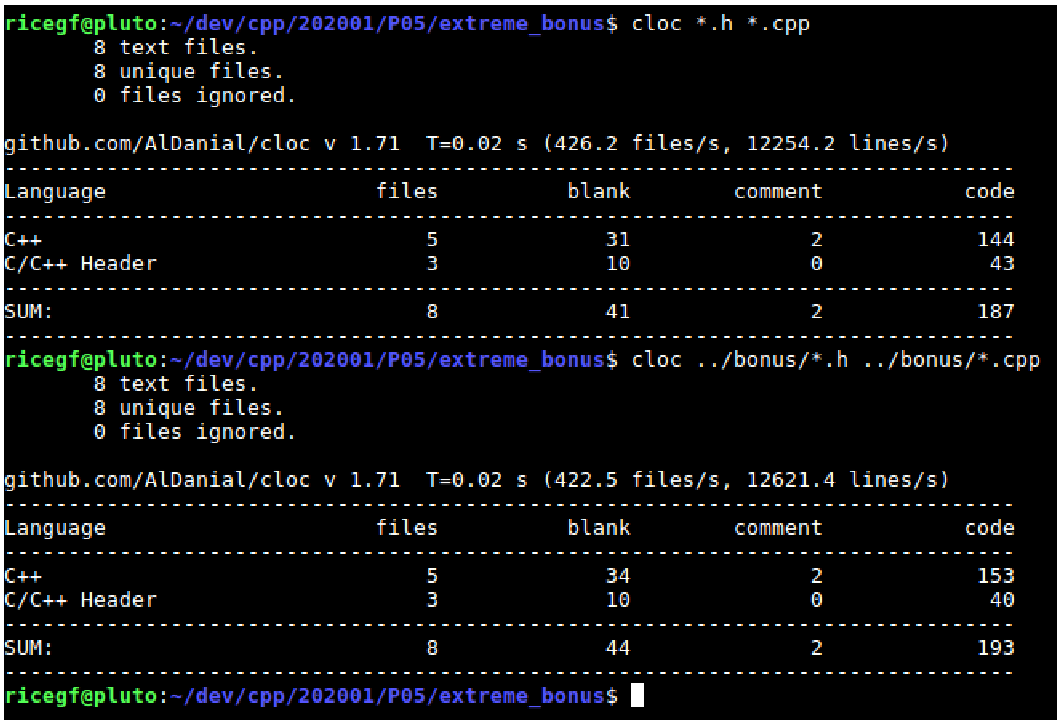
And (wait for it... wait for it...) Add, commit, and push all files.

> **Rev 2 Hint:** Now that your vectors for store products and customer orders are (I presume) *pointers* to instances of Taxed and Taxfree, to enable polymorphism, it's no longer adequate to just push the pointer to the desired product onto your order vector. If you do, you'll just have one instance of each product pointed to by all vectors, and *adding a quantity will affect every order - and your list of products!* (Yes, the original suggested solution had this bug. How careless of me!)
>
> Instead, you need to *clone* the dereferenced object from the vector of pointers to store products into a new location of the heap, and push *its* pointer onto the order vector. That way, they are independent instances, and adding a quantity to the order doesn't affect the store or any other orders.
>
> This is a bit of a trick, actually, but a not uncommon C++ idiom. I've indicated it on the class diagram below. See if you can infer how to do it. If you get stuck, check with me for additional hints. Stack Overflow wasn't as helpful to me the first time I encountered this need (yep - you'll see clone() again when we build our Shape library after Exam #1).


«abstract»
Product
#_quantity : int
#_cost : double
#_name : string
+Product(name : string, cost : double)
+~Product() «virtual»
+set_quantity(quantity : int)
+price() const : double «virtual»
+clone() : Product* «virtual»
+operator<<(ost : ostream&, product : const Product&) : ostream& «friend»
Store
+main() : int «function»
Taxed
- tax : double «static»
+Taxed(name : string, cost : double)
+~Taxed() «virtual»
+set_tax_rate(sales_tax : double) «static»
+price() : double const «override»
+clone() : Product* «override»
Taxfree
+Taxfree(name : string, cost : double)
+~Taxfree() «virtual»
+price() : double const «override»
+clone() : Product* «override»



```
ricegf@pluto:~/dev/cpp/202001/P05/extreme_bonus$ make clean
rm -f *.o *.gch a.out store test
ricegf@pluto:~/dev/cpp/202001/P05/extreme_bonus$ make
g++ --std=c++17 main.cpp -c -o main.o
g++ --std=c++17 product.cpp -c -o product.o
g++ --std=c++17 taxed.cpp -c -o taxed.o
g++ --std=c++17 taxfree.cpp -c -o taxfree.o
g++ --std=c++17 main.o product.o taxed.o taxfree.o  -o store
ricegf@pluto:~/dev/cpp/202001/P05/extreme_bonus$ ./store
========================
  Welcome to the Store
========================
0) Milk ($2.85)
1) Bread ($1.99)
2) Cheese ($0.99)
3) Ice Cream ($4.95)
4) Poptarts ($3.49)
5) Oreos ($5.99)

Enter quantity (0 to exit) and product index: 1 1

========================
  Welcome to the Store
========================
0) Milk ($2.85)
1) Bread ($1.99)
2) Cheese ($0.99)
3) Ice Cream ($4.95)
4) Poptarts ($3.49)
5) Oreos ($5.99)

Current Order
-------------
Bread (1 @ $1.99)

Total price: $1.99

Enter quantity (0 to exit) and product index: 0
```



```
ricegf@pluto:~/dev/cpp/202001/P05/extreme_bonus$ cloc *.h *.cpp
       8 text files.
       8 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.71  T=0.02 s (426.2 files/s, 12254.2 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                              5             31              2            144
C/C++ Header                     3             10              0             43
-------------------------------------------------------------------------------
SUM:                             8             41              2            187
-------------------------------------------------------------------------------
ricegf@pluto:~/dev/cpp/202001/P05/extreme_bonus$ cloc ../bonus/*.h ../bonus/*.cpp
       8 text files.
       8 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.71  T=0.02 s (422.5 files/s, 12621.4 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                              5             34              2            153
C/C++ Header                     3             10              0             40
-------------------------------------------------------------------------------
SUM:                             8             44              2            193
-------------------------------------------------------------------------------
ricegf@pluto:~/dev/cpp/202001/P05/extreme_bonus$
```