Assignment_12-08-2024

Author: ThanhTH10
Date: 13/08/2024

1. pre increment vs post increment. Which one is the faster one? Overload pre increment and post increment. Explain the difference with respect to assembly code by using objdump command

Characteristic	**Pre-increment (++x)**	**Post-increment (x++)**
Value of the expression	Incremented value of the variable	Original value of the variable
Order of operations	Variable is incremented before the expression is evaluated	Variable is incremented after the expression is evaluated
Return value	Reference to the incremented variable	Original value of the variable
Assembly code	`incl (%rax)`	`mov (%rax),%eax`, `incl (%rax)`
Performance	Slightly faster	Slightly slower
Use case	Preferred when the incremented value is needed immediately	Preferred when the original value needs to be used first

As for which one is faster, it largely depends on the specific use case and the underlying hardware/architecture. Generally, pre-increment is considered slightly faster because it doesn't require storing the original value before incrementing it.

To demonstrate the difference in assembly code, we can use the objdump command, which disassembles the compiled machine code.

```
#include <stdio.h>

int main() {
    int x = 0;
    printf("Pre-increment: %d\n", ++x);
    x = 0;
    printf("Post-increment: %d\n", x++);
    return 0;
}
```

Compiling the code and disassembling it using objdump -d gives us the following:

```
0000000000001119 <main>:
    1119:   55                      push   %rbp
    111a:   48 89 e5                mov    %rsp,%rbp
    111d:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    1124:   48 8d 45 fc             lea    -0x4(%rbp),%rax
    1128:   ff 00                   incl   (%rax)
    112a:   8b 45 fc                mov    -0x4(%rbp),%eax
    112d:   89 c6                   mov    %eax,%esi
    112f:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 1136 <main+0x1d>
    1136:   b8 00 00 00 00          mov    $0x0,%eax
    113b:   e8 00 00 00 00          callq  1140 <printf>
    1140:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    1147:   48 8d 45 fc             lea    -0x4(%rbp),%rax
    114b:   8b 00                   mov    (%rax),%eax
    114d:   89 c6                   mov    %eax,%esi
    114f:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 1156 <main+0x3d>
    1156:   b8 00 00 00 00          mov    $0x0,%eax
    115b:   e8 00 00 00 00          callq  1160 <printf>
    1160:   b8 00 00 00 00          mov    $0x0,%eax
    1165:   5d                      pop    %rbp
    1166:   c3                      retq
```

In the pre-increment case (++x), we can see that the value is incremented before it is used (incl   (%rax)), whereas in the post-increment case (x++), the original value is first used (mov (%rax),%eax), and then the variable is incremented (incl   (%rax)).

Regarding overloading pre-increment and post-increment, we can do so by defining the corresponding operator overload functions in your class. The pre-increment operator overload function would have the signature T& operator++(), and the post-increment operator overload function would have the signature T operator++(int) (the int parameter is just a dummy parameter to differentiate it from the pre-increment overload).

```cpp
class MyClass
{
    int value_ = 0;

public:
    MyClass &operator++()
    {
        // Implement pre-increment logic here
        ++value_;
        return *this;
    }
    MyClass operator++(int)
    {
        // Implement post-increment logic here
        MyClass temp = *this;
        ++value_;
        return temp;
    }
};
```

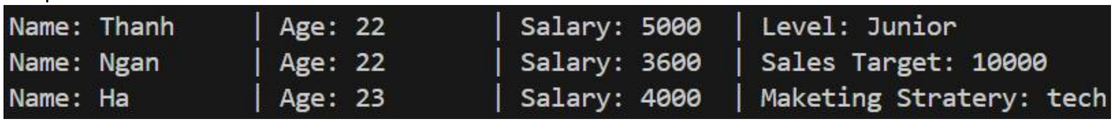
The main difference between the two is that the pre-increment operator overload returns a reference to the modified object, while the post-increment operator overload returns a copy of the original object before the increment operation.

2. Implement hierarchical inheritance for the employee base class with derived classes it team, sales team, marketing team

```cpp
#include <iostream>
using namespace std;
class Employee
{
protected:
    string name;
    int age;
    double salary;

public:
    Employee(string _name, int _age, double _salary) : name(_name), age(_age), salary(_salary)
{}
    void display()
    {
        cout << "Name: " << name << "\t| Age: " << age << "\t| Salary: " << salary;
    }
};
class ITTeam : public Employee
{
    string it_level;
public:
    ITTeam(string _name, int _age, double _salary, string _it_level) : Employee(_name, _age,
_salary), it_level(_it_level) {}
    void display()
    {
        Employee::display();
        cout << "\t| Level: " << it_level << endl;
    }
};
class SalesTeam : public Employee
{
    double salesTarget;
public:
    SalesTeam(string _name, int _age, double _salary, double _salesTarget) : Employee(_name,
_age, _salary), salesTarget(_salesTarget) {}
    void display()
    {
        Employee::display();
        cout << "\t| Sales Target: " << salesTarget << endl;
    }
};
class MarketingTeam : public Employee
{
    string marketingStrategy;
public:
    MarketingTeam(string _name, int _age, double _salary, string _marketingStrategy) :
Employee(_name, _age, _salary), marketingStrategy(_marketingStrategy) {}
    void display()
    {
        Employee::display();
        cout << "\t| Maketing Stratery: " << marketingStrategy << endl;
    }
};
int main()
{
    ITTeam it_team("Thanh", 22, 5000, "Junior");
    it_team.display();
    SalesTeam sales_team("Ngan", 22, 3600, 10000);
    sales_team.display();
    MarketingTeam maket_team("Ha", 23, 4000, "tech");
    maket_team.display();
    return 0;
}
```

Output:



```
Name: Thanh     | Age: 22        | Salary: 5000 | Level: Junior
Name: Ngan      | Age: 22        | Salary: 3600 | Sales Target: 10000
Name: Ha        | Age: 23        | Salary: 4000 | Maketing Stratery: tech
```