PL-Assignment-6

1. (30%) What is reference counting? How can it help solve the garbage collection problem? What is its limit in solving garbage collection problem?

1. 在Reference counting 下，每一個物件的field內都會有一個變數reference count，隨時記錄有多少個pointer指向此物件。
2. 在Reference counting 下，如果一個object的reference count為0，那就可以確定他不被任何pointer所reference，此時可以安全的釋出object所占記憶體。
3. 對於cyclic data-structures例如doubly linked lists 或 non-simple graphs，可能產生已經沒有pointer可以存取物件，但物件reference count仍不為0的狀況。可能造成memory leak，需要其他的garbage collection機制輔助處理。

2. (50+20%) How does the reference counting work in Python? You need to give an example to explain it. You may refer to the following articles for this problem. And, explain the differences of shallow copy and deep copy in Python.

1. 在Reference counting 下，每一個物件的field內都會有一個變數reference count，隨時記錄有多少個pointer指向此物件。當有物件被刪除，就會減少他對應的reference count，如果一個object的reference count為0，那就可以確定他不被任何pointer所reference，此時可以安全的釋出object所占記憶體。舉例來說，假如宣告兩個變數為同一個值:

```
x = 10
y = x
```

Python virtual machine 會讓y指向和x同一個記憶體位置的10。如此一來可以減少多一份的記憶體占用。但當其中一個變數改變時:

```
x+=1
```

Python就必須再為x分配一個記憶體空間11，和原先被y所指的10區隔開來。
2. 假設對於一個list，L = [[1],2]進行兩種copy。shallow copy會對第一層的物件複製產生新的記憶體位置，但對於內部的mutable object依然會和原來的L共用address。比如:

```
import copy
L = [[1],2]
shallowCopy = copy.copy(L)
L[0][0] = 3
L[1] = 4
```

則此時，L為[[3],4]，shallowCopy為[[3],2]。
而另一邊，deepcopy會recursively的對當中每一個元素複製產生新的記憶體位置，比如:

```
import copy
L = [[1],2]
deepCopy = copy.deepcopy(L)
L[0][0] = 3
L[1] = 4
```

則此時，L為[[3],4]，deepCopy為[[1],2]。

3. (bonus: 30%) What are smart pointers in C++? Please give at one to three examples to illustrate how to use them in C++.

Smart pointer是c++的一個wrapper class，其中的object可以像原來的pointer一樣被使用，除此之外，還有增加一些機制方便記憶體管理，比如auto-deallocation、reference counting、防止dangling pointers。
舉例來說:

auto-deallocation:

```
TeaShopOwnder* CreateOwner();
{
    TeaShopOwner* the_owner = CreateOwner();
    // Do something with the_owner
    delete the_owner;
}
```

傳統pointer的寫法會衍生兩個問題要解決:
1. CreateOwner 回傳的物件應由呼叫端刪除嗎?
2. 使用 delete 能正確刪除 CreateOwner 回傳的物件嗎?
假如問題沒有被處理好，很可能就會發生memory leak。而利用 std::unique_ptr來幫忙管理物件生命週期，比如:

```
std::unique_ptr<TeaShopOwner> CreateOwner();

{
    std::unique_ptr<TeaShopOwner> the_owner = CreateOwner();
    // Do something with the_owner
}
```

一旦離開 scope，則自動刪除所管理的物件。這樣的行為，即使函數內部發生 Exception，也能保證該物件被正確刪除，並釋放佔用的記憶體。

Reference counting:

```
void fun(std::shared_ptr<int> sp)
{
    std::cout << "fun: sp.use_count() == " << sp.use_count() << '\n';
}

int main()
{
    auto sp1 = std::make_shared<int>(5);
    std::cout << "sp1.use_count() == " << sp1.use_count() << '\n';
    fun(sp1);
    std::cout << "sp1.use_count() == " << sp1.use_count() << '\n';
}
```

這段程式碼運用std::shared_ptr，允許多個owner，從輸出:

```
sp1.use_count() == 1
fun: sp.use_count() == 2
sp1.use_count() == 1
```

中可以看到，sp1的reference count可以透過use_count()追蹤，再被新的owner reference時增加，在owner的scope結束時減少。

dangling pointers:

```
std::shared_ptr<int> sptr;

    sptr.reset(new int);
    *sptr = 30;
    std::weak_ptr<int> weak1 = sptr;
    sptr.reset(new int);
    *sptr = 7;
    std::weak_ptr<int> weak2 = sptr;

    if(auto tmp = weak1.lock())
        std::cout << *tmp << '\n';
    else
        std::cout << "Sorry, weak1 is no longer valid!\n";

    if(auto tmp = weak2.lock())
        std::cout << *tmp << '\n';
    else
        std::cout << "Sorry, weak2 is is no longer valid!\n";
```

從輸出:

```
Sorry, weak1 is no longer valid!
7
```

中可以看出，weak_ptr可以透過lock()檢查防止pointer因為指到已經被刪除的資料出錯。