Smart Pointer

**Unique\_ptr:** simple smart pointer.

Unique\_ptr<T>:

* Trỏ tới một đối tượng kiểu T trong heap
* Nó là con trỏ unique- chỉ có duy nhất một unique\_ptr<T> trỏ tới một đối tượng trong heap.
* Owns cái mà nó trở tới.
* **Không** thể **gán** hoặc **copy**
* Có thể **move**
* Khi mà con trỏ bị destroyed, cái mà nó trở tới cũng tự động được xóa.

#include <iostream>  
#include <memory>  
#include <vector>  
using namespace std;  
 class Test{  
 private:  
 int data;  
 public:  
 Test():data{0}{  
 cout<<"Test Constructor("<<data<<")"<<endl;  
 }  
 Test(int data):data{data}{  
 cout<<"Test Constructor("<<data<<")"<<endl;  
 }  
 int get\_data()const {  
 return data;  
 }  
 ~Test(){  
 cout<<"Test destructor("<<data<<")"<<endl;  
 }  
 };  
  
int main() {  
  
  
 unique\_ptr<Test>t1{new Test{1000}};  
 unique\_ptr<Test>t2=make\_unique<Test>(1000);  
  
 unique\_ptr<Test>t3;  
 t3=move(t1);  
 // t3=move(t1);  
 vector<unique\_ptr<Test>> vec;  
 vec.push\_back(move(t1));  
 vec.push\_back(move(t2));  
  
 for( auto it=vec.begin();it!=vec.begin();it++ ){  
 (\*it)->get\_data();  
 }  
  
  
  
  
 return 0;  
}

Share\_ptr:

* Cung cấp quyền sở hữu chung các đối tượng trong heap.
* Shared\_ptr<T>
* Trỏ tới một đối tượng kiểu T trong heap
* Nó không là duy nhất- có thể có nhiều con trỏ shared\_ptr trỏ tới cùng một đối tượng trong heap
* Thiết lập mối quan hệ chia sẻ chung
* Có thể gán hoặc copy
* Có thể move
* Không hỗ trợ quản lí mảng mặc định
* Khi mà số lượng sử dụng là 0, thì Object được quản lí trong heap sẽ được phá hủy.
* Shared\_ptr : some other useful methods
* Use\_count: số lượng của con trỏ shared\_ptr đang quản lí object trong heap
* Make\_shared - tạo ra một con trỏ shared\_ptr

shared\_ptr<int> p1{new int{100}};  
shared\_ptr<int> p2{p1};  
cout<<"Use count: "<<p1.use\_count()<<endl;  
  
cout<<"\n=========================================================\n";  
shared\_ptr<Test> ptr=make\_shared<Test>(1000);  
func(ptr);  
  
cout<<"Use count: "<<ptr.use\_count()<<endl;  
{  
 shared\_ptr<Test> ptr1=ptr;  
 cout<<"Use count"<<ptr.use\_count()<<endl;  
 {  
 shared\_ptr<Test> ptr2=ptr;  
 cout<<"Use count"<<ptr.use\_count()<<endl;  
 ptr.reset();  
 }  
 cout<<"Use count"<<ptr.use\_count()<<endl;  
}  
cout<<"Use count"<<ptr.use\_count()<<endl;  
  
  
vector<shared\_ptr<Test>> vec;  
vec.push\_back(make\_shared<Test>(1000));  
vec.push\_back(make\_shared<Test>(2000));  
  
for(const auto &ptr:vec){  
   
}

weak\_ptr

* Cung cấp một reference non-owning “weak”

Weak\_ptr<T>

* Trỏ tới một đối tượng kiểu T trên heap
* Không tham gia vào owning relationship
* Luôn luôn được tạo ra từ một shared\_ptr
* Không tăng hoặc giảm reference sử dụng count
* Thường ngăn cản vòng lặp strong reference cái mà có thể ngăn cản object được xóa.

#include <iostream>  
#include <memory>  
#include <vector>  
using namespace std;  
  
class B;  
class A{  
 shared\_ptr<B> b\_ptr;  
public:  
 void set\_B(shared\_ptr<B> &b){  
 b\_ptr=b;  
 }  
 A(){  
 cout<<"A constructor"<<endl;  
 }  
 ~A(){  
 cout<<"A destructor"<<endl;  
 }  
};  
  
class B{  
 weak\_ptr<A> a\_ptr;  
public:  
 void set\_A(shared\_ptr<A> &a){  
 a\_ptr=a;  
 }  
 B(){  
 cout<<"B constructor"<<endl;  
 }  
 ~B(){  
 cout<<"B destructor"<<endl;  
 }  
};  
  
int main(){  
 shared\_ptr<A> a=make\_shared<A>();  
 shared\_ptr<B> b=make\_shared<B>();  
  
 a->set\_B(b);  
 b->set\_A(a);  
 cout<<a.use\_count();  
}

Custom Deleters:

* Đôi khi chúng ta destroy một smart pointer chúng ta cần nhiều hơn là chỉ phá huỷ object trong heap
* C++ smart pointer cho phép bạn cung cấp custom deleters
* Có nhiều cách như là :
* Functions
* Lambdas
* Others…