ECE 2400 Computer Systems Programming

Topic 13: Generic Programming

http://www.csl.cornell.edu/courses/ece2400
School of Electrical and Computer Engineering
Cornell University

revision: 2025-04-16-11-03

Please do not ask for solutions. Students should compare their solutions to solutions from their fellow students, discuss their solutions with the instructors during lab/office hours, and/or post their solutions on Ed for discussion.

List of Problems

1 Short Answer **2**

1.A Unique Smart Pointer Class 3

1.B Shared Smart Pointer Class 5

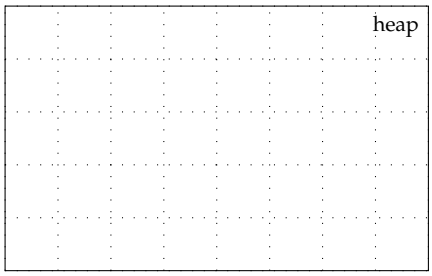
Problem 1. Short Answer

Carefully plan your solution before starting to write your response. Please be brief and to the point; if at all possible, limit your answers to the space provided.

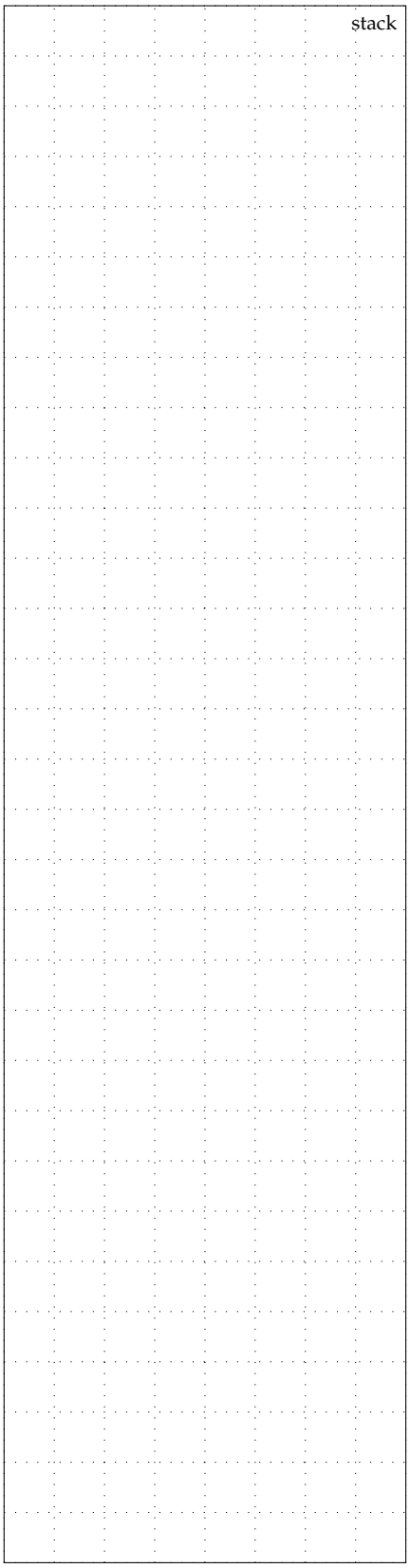
Part 1.A Unique Smart Pointer Class

Consider the C++ program on the following page which defines a new unique pointer class `UniquePtr<T>`. A unique pointer is a kind of "smart pointer" which is a thin wrapper around a regular pointer. A unique pointer provides a destructor that will automatically delete a dynamically allocated variable. Ideally, a unique pointer would ensure that at all times exactly one unique pointer will ever point to the dynamically allocated object (that is why it is unique!). A real unique pointer class would provide additional member functions and overloaded operators to preserve this invariant and to access the private regular pointer. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls. *Hint: There will be many arrows. Plan how to draw your arrows carefully and ensure your diagram is neat and legible. Solutions with confusing arrows will be penalized.*

heap



stack



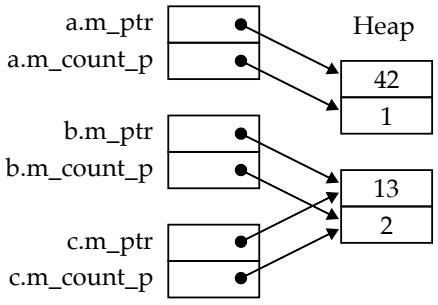
```
1  template < typename T >
2  class UniquePtr
3  {
4   public:
5    UniquePtr()         { m_ptr = nullptr; }
6    UniquePtr( T* ptr ) { m_ptr = ptr;     }
7    ~UniquePtr()        { delete m_ptr;    }
8
9    UniquePtr( const UniquePtr<T>& uptr )
10   {
11     m_ptr      = uptr.m_ptr;
12     uptr.m_ptr = nullptr;
13   }
14
15   UniquePtr<T>&
16   operator=( const UniquePtr<T>& uptr )
17   {
18     if ( this != &uptr ) {
19       delete m_ptr;
20       m_ptr      = uptr.m_ptr;
21       uptr.m_ptr = nullptr;
22     }
23     return *this;
24   }
25
26  private:
27   T* m_ptr;
28 };
29
30 int main( void )
31 {
32   int* ptr = new int;
33   UniquePtr<int> a( ptr );
34   UniquePtr<int> b( a );
35   return 0;
36 }
```

Part 1.B Shared Smart Pointer Class

Consider the interface for a new shared pointer class `SharedPtr<T>` shown below on the left. A shared pointer is a kind of "smart pointer" which is a thin wrapper around a regular pointer. A shared pointer manages dynamic memory by deleting a dynamically allocated variable automatically when appropriate. This specific shared pointer implements *reference counting* where the variable *and* a reference count are stored on the heap. Every time a shared pointer is copied, we increment the reference count since this means there is one more pointer that refers to the shared variable. Every time we destruct a shared pointer, we decrement the reference counter since this means there is one less pointer that refers to the shared variable. If the reference count is zero then we are guaranteed that no other shared pointers point to this variable, and thus the shared pointer is free to delete the variable from the heap. We have provided you the two member fields and the implementation of the overloaded dereference (`*`) and arrow (`->`) operators which essentially enable a smart pointer to act like a regular pointer.

```
1  template < typename T >
2  class SharedPtr
3  {
4   public:
5
6    SharedPtr();
7    SharedPtr( T* ptr );
8
9    ~SharedPtr();
10   SharedPtr( const SharedPtr<T>& sptr );
11
12   SharedPtr<T>&
13   operator=( const SharedPtr<T>& sptr );
14
15   T* operator->() { return m_ptr;  }
16   T& operator*()  { return *m_ptr; }
17
18  private:
19   T*   m_ptr;
20   int* m_count_p;
21 };
```

```
22 int main( void )
23 {
24   SharedPtr<int> a( new int );
25   SharedPtr<int> b( a );
26   *b = 42;
27
28   SharedPtr<int> c( new int );
29   b = c;
30   *b = 13;
31
32   return 0;
33 }
```



An example of a shared pointer in use is shown above on the right. Some visual pseudo-code is also shown on the right that illustrates the state after line 31 and before the `return 0` statement. The shared pointer `a` points to a variable of type `int` on the heap along with a corresponding reference count of one. The shared pointers `b` and `c` both point to another variable of type `int` on the heap along with a corresponding reference count of two (i.e., pointers `b` and `c` are *sharing* this variable on the heap). When these three shared pointers go out of scope (i.e., when `main` returns), their destructors will be called and (if implemented correctly) both variables will be deleted from the heap with no double deletes and no memory leaks.

The implementation for the default and non-default constructors are provided for you below. **Implement the destructor, copy constructor, and assignment operator for this shared pointer.** Your implementation should be correct, efficient, and effectively hide the implementation details from the user. *Hint: Manually verify that your implementation would result in the above visual pseudo-code right before the `return 0` statement.*

```
template < typename T >
SharedPtr<T>::SharedPtr() {
  m_ptr     = nullptr;
  m_count_p = nullptr;
}

template < typename T >
SharedPtr<T>::SharedPtr( T* ptr ) {
  if ( ptr == nullptr ) {
    m_ptr      = nullptr;
    m_count_p  = nullptr;
  }
  else {
    m_ptr      = ptr;
    m_count_p  = new int;
    *m_count_p = 1;
  }
}

template < typename T >
SharedPtr<T>::~SharedPtr() {
```

```
template < typename T >
SharedPtr<T>::SharedPtr( const SharedPtr<T>& sptr ) {
```

```
template < typename T >
SharedPtr<T>& SharedPtr<T>::operator=( const SharedPtr<T>& sptr ) {
```