

Math / Stat.

P1.

(Update, Thanks to Ji Yang's idea)

For any girl, the next one to her (clockwise) is a boy with probability $B/(B+G-1)$;

For any boy, the next one to him (clockwise) is a girl with probability $G/(B+G-1)$.

So the total expected number of neighbors is $G \cdot B/(B+G-1) + B \cdot G/(B+G-1) = 2GB/(B+G-1)$.

P2.

Uncorrelated random variables are defined as random variables which have zero correlation. If for two random variables, the conditional distribution of one on the other equals the distribution of itself, these two random variables are independent with each other. Obviously, independence is stronger statement.

Example of random variables that are uncorrelated but not independent:

Assume random variable X satisfies $f(X) = f(-X)$, set $Y = |X|$. We have $\text{Cor}(X, Y) = E(X)E(Y) - E(XY) = 0 - 0 = 0$. Then X and Y are uncorrelated. But obviously, X and Y are not independent.

P3.

This is a typical dynamic programming problem, appeared in the Green book page 123, Dice Game.

If I am given only 1 time to toss, the value is $1/6 \cdot (1+2+3+4+5+6) = 3.5$

If given 2 times, the value is $1/6*(4+5+6+3*3.5) = 4.25$ considering that I will continue if the first toss gets less than 3.5, the expected value of the next toss if I do not stop after the first.

If given 3 times, the value is $1/6*(5+6+4*4.25) = 14/3$ considering that I will continue if the first toss gets less than 4.25, the expected value of the next two tosses if I do not stop after the first.

P5

covariant matrix is symmetric positive semidefinite. According to Sylvester's criterion, its principal minors are greater than or equal to 0.

$$\Omega = \begin{vmatrix} 1 & r & 0 \\ r & 1 & r \\ 0 & r & 1 \end{vmatrix}, |\Omega| = 1 - r^2 > 0, \begin{vmatrix} 1 & r \\ r & 1 \end{vmatrix} > 0 \implies -1 < r < 1$$

P6

S_n as a symmetric random walk. (Where $S_n = X_1 + X_2 + X_3 \cdots + X_n$ with n being the stopping time). We know that both S_n and $S_n^2 - n$ are martingales.

a) Let $P(-1)$ be the probability that the drunk man go through the left door, and N be the stopping time. $E[S_N] = P(-1)*(-1) + (1-P(-1))*99 = S_0 = 0$, $E[S_N^2 - N] =$

$$E[P(-1)*1 + (1-P(-1))*99^2] - E[N] = S_0^2 - 0 = 0, \implies P(-1) = 99/100, N = 99$$

b) if left door is locked, the drunk man need to walk right for the next step at the left door, and exit from the right door. we could use symmetric analysis, it's equivalent that the drunk man could exit at either 99 or -101. Use the same method, we get $P(99) = 0.505$, $N = 9999$

P7&8. What is Ridge regression and Lasso regression?

Ridge and Lasso are two shrinkage methods. They shrink the regression coefficients by imposing a penalty on their size. There are two motivations to use Ridge and Lasso regressions:

Motivation 1: too many predictors

♣ It is not unusual to see the number of input variables greatly exceed the number of observations, e.g. micro-array data analysis, environmental pollution studies.

♣ With many predictors, fitting the full model without penalization will result in large prediction intervals, and LS regression estimator may not uniquely exist.

Motivation 2: correlated variables X in a linear regression model

♣ Because the LS estimates depend upon $(X^T X)^{-1}$, we would have problems in computing if $X^T X$ were singular or nearly singular.

♣ In those cases, small changes to the elements of X lead to large changes in β . In other words, due to the correlated variables, the coefficients are poorly determined and exhibit high variance.

The Ridge and Lasso coefficients minimize a penalized residual sum of squares:

Here λ is a complexity parameter that controls the amount of shrinkage: the larger the value of λ , the greater the amount of shrinkage. The coefficients are shrunk toward zero.

The difference between Ridge and Lasso regression is that when the estimated coefficients have sharp boundary, e.g. the coefficients are piece-wise, Lasso can capture this sharp boundary better than Ridge.

P9.

Assumes the wealth of the player is w and the ticket price is c . The player would only want to play if the expected exponential rate of growth $g(w,c)$ is positive:

where $D_k = 2^k$ is the payoff at round k and $p_k = 1/(2^k)$ is the probability.
 $g(w,c)$ is a decreasing function of c . The constraint $g(w,c) > 0$ gives the upper bound of c , which is the maximum price the player would want to pay for the game. Numerically, for $w=100$, $c \approx 3$; for $w = 1\text{million}$, $c \approx 8$; for $w=100\text{million}$, $c \approx 12$.

Programming

P10.

The use of 'default' in C++ is a class' member functions automatically generated by the compiler if not declared. They include constructor, copy constructor, copy assignment operator, and destructor. All these functions will be both public and inline. (see item 5 of Effective C++" third version).

You should generally write a default constructor for every class you define, to guarantee the state of any "default constructed" variable. If you don't declare a default constructor, the compiler will supply one for you; however, since it does not know much about your class, it won't be able to guarantee very much about the initial state of one of your variables. In fact, all the native types will be left in random state, as though they were declared but not initialized; this is an undesirable condition.

Why did I say 'generally', but not always? Because there are some times when you do not want to allow an object to be created unless the 'real' data is available.

If you want to make it impossible to create an object via compiler-generated default constructor, you can declare a private default constructor that will cause a compiler error in any user code that tried to define an object of that class without specifying an initial value. You do not have to implement this constructor, because a program that tried to use it won't compile. (see page 328 of C++: A Dialogue)

P11.

<http://stackoverflow.com/questions/2576022/efficient-thread-safe-singleton-in-c>

- A. This [Meyers/Alexandrescu paper](http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf) (www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf) explains why - but that paper is also widely misunderstood. It started the 'double checked locking is unsafe in C++' meme - but its actual conclusion is that double checked locking in C++ can be implemented safely, it just requires the use of memory barriers in a non-obvious place.

The paper contains pseudocode demonstrating how to use memory barriers to safely implement the DLCP, so it shouldn't be difficult for you to correct your implementation.

- B. If you are using C++11, here is a right way to do this:

`Foo& getInst()`

```
{  
    static Foo inst(...);  
    return inst;  
}
```

According to new standard there is no need to care about this problem any more. Object initialization will be made only by one thread, other threads will wait till it complete. Or you can use `std::call_once`. (more info [here](#))

- C. ACE singleton implementation uses double-checked locking pattern for thread safety, you can refer to it if you like. You can find source code [here](#).

```
00001 // Singleton.cpp,v 4.54 2005/10/28 16:14:55 ossama Exp  
00002  
00003 #ifndef ACE_SINGLETON_CPP  
00004 #define ACE_SINGLETON_CPP  
00005  
00006 #include "ace/Singleton.h"  
00007  
00008 #if !defined (ACE_LACKS_PRAGMA_ONCE)  
00009 # pragma once  
00010 #endif /* ACE_LACKS_PRAGMA_ONCE */  
00011  
00012 #if !defined (__ACE_INLINE__)  
00013 #include "ace/Singleton.inl"  
00014 #endif /* __ACE_INLINE__ */  
00015  
00016 #include "ace/Object_Manager.h"  
00017 #include "ace/Log_Msg.h"  
00018 #include "ace/Framework_Component.h"  
00019 #include "ace/Guard_T.h"
```

```

00020
00021 ACE_RCSID (ace,
00022     Singleton,
00023     "Singleton.cpp,v 4.54 2005/10/28 16:14:55 ossama Exp")
00024
00025
00026 ACE_BEGIN_VERSIONED_NAMESPACE_DECL
00027
00028 template <class TYPE, class ACE_LOCK> void
00029 ACE_Singleton<TYPE, ACE_LOCK>::dump (void)
00030 {
00031     #if defined (ACE_HAS_DUMP)
00032         ACE_TRACE ("ACE_Singleton<TYPE, ACE_LOCK>::dump");
00033
00034     #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00035         ACE_DEBUG ((LM_DEBUG, ACE_LIB_TEXT ("instance_ = %x"),
00036             ACE_Singleton<TYPE, ACE_LOCK>::instance_i ());
00037         ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
00038     #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00039 #endif /* ACE_HAS_DUMP */
00040 }
00041
00042 template <class TYPE, class ACE_LOCK> ACE_Singleton<TYPE, ACE_LOCK> *&
00043 ACE_Singleton<TYPE, ACE_LOCK>::instance_i (void)
00044 {
00045     #if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00046         // Pointer to the Singleton instance. This works around a bug with
00047         // G++ and it's (mis-)handling of templates and statics...
00048         static ACE_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;
00049
00050         return singleton_;
00051     #else
00052         return ACE_Singleton<TYPE, ACE_LOCK>::singleton_;
00053     #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00054 }
00055
00056 template <class TYPE, class ACE_LOCK> TYPE *
00057 ACE_Singleton<TYPE, ACE_LOCK>::instance (void)
00058 {
00059     ACE_TRACE ("ACE_Singleton<TYPE, ACE_LOCK>::instance");
00060
00061     ACE_Singleton<TYPE, ACE_LOCK> *&singleton =
00062         ACE_Singleton<TYPE, ACE_LOCK>::instance_i ();

```

```

00063
00064 // Perform the Double-Check pattern...
00065 if (singleton == 0)
00066 {
00067     if (ACE_Object_Manager::starting_up () ||
00068         ACE_Object_Manager::shutting_down ())
00069     {
00070         // The program is still starting up, and therefore assumed
00071         // to be single threaded. There's no need to double-check.
00072         // Or, the ACE_Object_Manager instance has been destroyed,
00073         // so the preallocated lock is not available. Either way,
00074         // don't register for destruction with the
00075         // ACE_Object_Manager: we'll have to leak this instance.
00076
00077         ACE_NEW_RETURN (singleton, (ACE_Singleton<TYPE, ACE_LOCK>), 0);
00078     }
00079     else
00080     {
00081 #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00082         // Obtain a lock from the ACE_Object_Manager. The pointer
00083         // is static, so we only obtain one per ACE_Singleton
00084         // instantiation.
00085         static ACE_LOCK *lock = 0;
00086         if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
00087             // Failed to acquire the lock!
00088             return 0;
00089
00090         ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
00091
00092         if (singleton == 0)
00093         {
00094 #endif /* ACE_MT_SAFE */
00095             ACE_NEW_RETURN (singleton, (ACE_Singleton<TYPE, ACE_LOCK>), 0);
00096
00097             // Register for destruction with ACE_Object_Manager.
00098             ACE_Object_Manager::at_exit (singleton);
00099 #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00100         }
00101 #endif /* ACE_MT_SAFE */
00102     }
00103 }
00104
00105 return &singleton->instance_;

```

```

00106 }
00107
00108 template <class TYPE, class ACE_LOCK> void
00109 ACE_Singleton<TYPE, ACE_LOCK>::cleanup (void *)
00110 {
00111     delete this;
00112     ACE_Singleton<TYPE, ACE_LOCK>::instance_i () = 0;
00113 }
00114
00115 #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00116 // Pointer to the Singleton instance.
00117 template <class TYPE, class ACE_LOCK> ACE_Singleton<TYPE, ACE_LOCK> *
00118 ACE_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;
00119
00120 template <class TYPE, class ACE_LOCK> ACE_Unmanaged_Singleton<TYPE,
ACE_LOCK> *
00121 ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;
00122 #endif /* !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES) */
00123
00124 template <class TYPE, class ACE_LOCK> void
00125 ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::dump (void)
00126 {
00127     #if defined (ACE_HAS_DUMP)
00128         ACE_TRACE ("ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::dump");
00129     #endif
00130     #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00131         ACE_DEBUG ((LM_DEBUG, ACE_LIB_TEXT ("instance_ = %x"),
00132             ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i ()));
00133         ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
00134     #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00135     #endif /* ACE_HAS_DUMP */
00136 }
00137
00138 template <class TYPE, class ACE_LOCK>
00139 ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *&
00140 ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i (void)
00141 {
00142     #if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00143         // Pointer to the Singleton instance. This works around a bug with
00144         // G++ and it's (mis-)handling of templates and statics...
00145         static ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;
00146     #endif
00147     return singleton_;

```

```

00148 #else
00149     return ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::singleton_;
00150 #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00151 }
00152
00153 template <class TYPE, class ACE_LOCK> TYPE *
00154 ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance (void)
00155 {
00156     ACE_TRACE ("ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance");
00157
00158     ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *&singleton =
00159         ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i ();
00160
00161     // Perform the Double-Check pattern...
00162     if (singleton == 0)
00163     {
00164         if (ACE_Object_Manager::starting_up () ||
00165             ACE_Object_Manager::shutting_down ())
00166         {
00167             // The program is still starting up, and therefore assumed
00168             // to be single threaded. There's no need to double-check.
00169             // Or, the ACE_Object_Manager instance has been destroyed,
00170             // so the preallocated lock is not available. Either way,
00171             // don't register for destruction with the
00172             // ACE_Object_Manager: we'll have to leak this instance.
00173
00174             ACE_NEW_RETURN (singleton, (ACE_Unmanaged_Singleton<TYPE,
00175                                     ACE_LOCK>),
00176                             0);
00177         }
00178     }
00179     else
00180     {
00181         #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00182             // Obtain a lock from the ACE_Object_Manager. The pointer
00183             // is static, so we only obtain one per
00184             // ACE_Unmanaged_Singleton instantiation.
00185             static ACE_LOCK *lock = 0;
00186             if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
00187                 // Failed to acquire the lock!
00188                 return 0;
00189             ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
00190             #endif /* ACE_MT_SAFE */
00191         return singleton;
00192     }
00193 }

```



```

00190
00191     if (singleton == 0)
00192         ACE_NEW_RETURN (singleton,
00193             (ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>),
00194             0);
00195     }
00196 }
00197
00198 return &singleton->instance_;
00199 }
00200
00201 template <class TYPE, class ACE_LOCK> void
00202 ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::close (void)
00203 {
00204     ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *&singleton =
00205     ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i ();
00206
00207     if (singleton)
00208     {
00209         singleton->cleanup ();
00210         ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i () = 0;
00211     }
00212 }
00213
00214 template <class TYPE, class ACE_LOCK> void
00215 ACE_TSS_Singleton<TYPE, ACE_LOCK>::dump (void)
00216 {
00217     #if defined (ACE_HAS_DUMP)
00218         ACE_TRACE ("ACE_TSS_Singleton<TYPE, ACE_LOCK>::dump");
00219
00220     #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00221         ACE_DEBUG ((LM_DEBUG, ACE_LIB_TEXT ("instance_ = %x"),
00222             ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ());
00223         ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
00224     #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00225     #endif /* ACE_HAS_DUMP */
00226 }
00227
00228 template <class TYPE, class ACE_LOCK> ACE_TSS_Singleton<TYPE, ACE_LOCK>
*&
00229 ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i (void)
00230 {
00231     #if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)

```

```

00232 // Pointer to the Singleton instance. This works around a bug with
00233 // G++ and it's (mis-)handling of templates and statics...
00234 static ACE_TSS_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;
00235
00236 return singleton_;
00237 #else
00238 return ACE_TSS_Singleton<TYPE, ACE_LOCK>::singleton_;
00239 #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00240 }
00241
00242 template <class TYPE, class ACE_LOCK> TYPE *
00243 ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance (void)
00244 {
00245     ACE_TRACE ("ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance");
00246
00247     ACE_TSS_Singleton<TYPE, ACE_LOCK> *&singleton =
00248         ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ();
00249
00250     // Perform the Double-Check pattern...
00251     if (singleton == 0)
00252     {
00253         if (ACE_Object_Manager::starting_up () ||
00254             ACE_Object_Manager::shutting_down ())
00255         {
00256             // The program is still starting up, and therefore assumed
00257             // to be single threaded. There's no need to double-check.
00258             // Or, the ACE_Object_Manager instance has been destroyed,
00259             // so the preallocated lock is not available. Either way,
00260             // don't register for destruction with the
00261             // ACE_Object_Manager: we'll have to leak this instance.
00262
00263             ACE_NEW_RETURN (singleton, (ACE_TSS_Singleton<TYPE, ACE_LOCK>),
00264                             0);
00265         }
00266     }
00267     else
00268     {
00269         #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00270
00269         // Obtain a lock from the ACE_Object_Manager. The pointer
00270         // is static, so we only obtain one per ACE_Singleton instantiation.
00271         static ACE_LOCK *lock = 0;
00272         if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
00273             // Failed to acquire the lock!

```

```

00274         return 0;
00275
00276         ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
00277
00278         if (singleton == 0)
00279         {
00280 #endif /* ACE_MT_SAFE */
00281             ACE_NEW_RETURN (singleton, (ACE_TSS_Singleton<TYPE, ACE_LOCK>),
00282                             0);
00283
00284             // Register for destruction with ACE_Object_Manager.
00285             ACE_Object_Manager::at_exit (singleton);
00286 #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00287         }
00288 #endif /* ACE_MT_SAFE */
00289     }
00290 }
00291
00292 return ACE_TSS_GET (&singleton->instance_, TYPE);
00293 }
00294
00295 template <class TYPE, class ACE_LOCK> void
00296 ACE_TSS_Singleton<TYPE, ACE_LOCK>::cleanup (void *)
00297 {
00298     delete this;
00299     ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i () = 0;
00300 }
00301
00302 template <class TYPE, class ACE_LOCK> void
00303 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::dump (void)
00304 {
00305 #if defined (ACE_HAS_DUMP)
00306     ACE_TRACE ("ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::dump");
00307
00308 #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00309     ACE_DEBUG ((LM_DEBUG, ACE_LIB_TEXT ("instance_ = %x"),
00310                ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ()));
00311     ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
00312 #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00313 #endif /* ACE_HAS_DUMP */
00314 }
00315
00316 template <class TYPE, class ACE_LOCK>

```

```

00317 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *&
00318 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i (void)
00319 {
00320 #if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00321 // Pointer to the Singleton instance. This works around a bug with
00322 // G++ and it's (mis-)handling of templates and statics...
00323 static ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;
00324
00325 return singleton_;
00326 #else
00327 return ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::singleton_;
00328 #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00329 }
00330
00331 template <class TYPE, class ACE_LOCK> TYPE *
00332 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance (void)
00333 {
00334 ACE_TRACE ("ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance");
00335
00336 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *&singleton =
00337 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ();
00338
00339 // Perform the Double-Check pattern...
00340 if (singleton == 0)
00341 {
00342 if (ACE_Object_Manager::starting_up () ||
00343 ACE_Object_Manager::shutting_down ())
00344 {
00345 // The program is still starting up, and therefore assumed
00346 // to be single threaded. There's no need to double-check.
00347 // Or, the ACE_Object_Manager instance has been destroyed,
00348 // so the preallocated lock is not available. Either way,
00349 // don't register for destruction with the
00350 // ACE_Object_Manager: we'll have to leak this instance.
00351
00352 ACE_NEW_RETURN (singleton,
00353 (ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>),
00354 0);
00355 }
00356 else
00357 {
00358 #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00359 // Obtain a lock from the ACE_Object_Manager. The pointer

```

```

00360 // is static, so we only obtain one per
00361 // ACE_Unmanaged_Singleton instantiation.
00362 static ACE_LOCK *lock = 0;
00363 if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
00364 // Failed to acquire the lock!
00365 return 0;
00366
00367 ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
00368 #endif /* ACE_MT_SAFE */
00369
00370 if (singleton == 0)
00371 ACE_NEW_RETURN (singleton,
00372                 (ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>,
00373                  0);
00374 }
00375 }
00376
00377 return ACE_TSS_GET (&singleton->instance_, TYPE);
00378 }
00379
00380 template <class TYPE, class ACE_LOCK> void
00381 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::close (void)
00382 {
00383 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *&singleton =
00384 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ();
00385
00386 if (singleton)
00387 singleton->cleanup ();
00388 }
00389
00390 #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00391 // Pointer to the Singleton instance.
00392 template <class TYPE, class ACE_LOCK> ACE_TSS_Singleton <TYPE, ACE_LOCK>
00393 *
00394 ACE_TSS_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;
00395
00396 template <class TYPE, class ACE_LOCK>
00397 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *
00398 ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;
00399 #endif /* !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES) */
00400 /*****
00401

```

```

00402 #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00403 // Pointer to the Singleton instance.
00404 template <class TYPE, class ACE_LOCK> ACE_DLL_Singleton_T<TYPE,
ACE_LOCK> *
00405 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::singleton_ = 0;
00406 #endif /* !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES) */
00407
00408 template <class TYPE, class ACE_LOCK> void
00409 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::dump (void)
00410 {
00411 #if defined (ACE_HAS_DUMP)
00412 ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::dump");
00413
00414 #if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00415 ACE_DEBUG ((LM_DEBUG, ACE_LIB_TEXT ("instance_ = %x"),
00416 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i ());
00417 ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
00418 #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00419 #endif /* ACE_HAS_DUMP */
00420 }
00421
00422 template <class TYPE, class ACE_LOCK>
00423 ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *&
00424 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i (void)
00425 {
00426 ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i");
00427
00428 #if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
00429 // Pointer to the Singleton instance. This works around a bug with
00430 // G++ and it's (mis-)handling of templates and statics...
00431 static ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *singleton_ = 0;
00432
00433 return singleton_;
00434 #else
00435 return ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::singleton_;
00436 #endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
00437 }
00438
00439 template <class TYPE, class ACE_LOCK> TYPE *
00440 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance (void)
00441 {
00442 ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance");
00443

```

```

00444 ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *&singleton =
00445     ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i ();
00446
00447 // Perform the Double-Check pattern...
00448 if (singleton == 0)
00449 {
00450     if (ACE_Object_Manager::starting_up () ||
00451         ACE_Object_Manager::shutting_down ())
00452     {
00453         // The program is still starting up, and therefore assumed
00454         // to be single threaded. There's no need to double-check.
00455         // Or, the ACE_Object_Manager instance has been destroyed,
00456         // so the preallocated lock is not available. Either way,
00457         // don't register for destruction with the
00458         // ACE_Object_Manager: we'll have to leak this instance.
00459
00460         ACE_NEW_RETURN (singleton, (ACE_DLL_Singleton_T<TYPE, ACE_LOCK>),
00461             0);
00462     }
00463     else
00464     {
00465 #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
00466         // Obtain a lock from the ACE_Object_Manager. The pointer
00467         // is static, so we only obtain one per
00468         // ACE_Unmanaged_Singleton instantiation.
00469         static ACE_LOCK *lock = 0;
00470         if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
00471             // Failed to acquire the lock!
00472             return 0;
00473
00474         ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
00475 #endif /* ACE_MT_SAFE */
00476
00477         if (singleton == 0)
00478             ACE_NEW_RETURN (singleton,
00479                 (ACE_DLL_Singleton_T<TYPE, ACE_LOCK>),
00480                 0);
00481     }
00482     //
ACE_REGISTER_FRAMEWORK_COMPONENT(ACE_DLL_Singleton<TYPE,ACE_LOCK>,
singleton);
00483     ACE_Framework_Repository::instance ()->register_component
00484     (new ACE_Framework_Component_T<ACE_DLL_Singleton_T<TYPE,
ACE_LOCK> > (singleton));

```

```

00485     }
00486
00487     return &singleton->instance_;
00488 }
00489
00490 template <class TYPE, class ACE_LOCK> void
00491 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close (void)
00492 {
00493     ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close");
00494
00495     ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *&singleton =
00496         ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i ();
00497
00498     delete singleton;
00499     singleton = 0;
00500 }
00501
00502 template <class TYPE, class ACE_LOCK> void
00503 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close_singleton (void)
00504 {
00505     ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close_singleton");
00506     ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close ();
00507 }
00508
00509 template <class TYPE, class ACE_LOCK> const ACE_TCHAR *
00510 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::dll_name (void)
00511 {
00512     return this->instance ()->dll_name ();
00513 }
00514
00515 template <class TYPE, class ACE_LOCK> const ACE_TCHAR *
00516 ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::name (void)
00517 {
00518     return this->instance ()->name ();
00519 }
00520
00521
00522 /*****
00523
00524 template <class TYPE> const ACE_TCHAR*
00525 ACE_DLL_Singleton_Adapter_T<TYPE>::dll_name (void)
00526 {
00527     // @todo make this a constant somewhere (or it there already is one

```



```

00528 // then use it.
00529 return ACE_TEXT("ACE");
00530 }
00531
00532 ACE_END_VERSIONED_NAMESPACE_DECL
00533
00534 #endif /* ACE_SINGLETON_CPP */

```

P13.

There are several ways to generate random numbers. One of the most common PRNG is the linear congruential generator, which uses the recurrence

to generate numbers, where a , b and m are large integers, and X_{n+1} is the next in X as a series of pseudo-random numbers.

Testing

Assume that f is a function taking any finite sequence of zeros and ones, and returning a non-negative real value. Then, given a sequence of independent and uniformly distributed random variables X_n , applying f to the finite sequence of random variables (X_1, \dots, X_n) yields a new random variable, Y_n . This new variable has a certain cumulative probability distribution

$F_n(x) = \mathbb{P}(Y_n \leq x)$, which in some cases approaches a function F as n grows large. This limit function F can be seen as the cumulative probability distribution of a new random variable, Y , and in these cases Y_n is said to converge in distribution to Y .

P15.

Exclusive or operation is a logical operation that outputs true only when both inputs differ. In this problem we could sum all elements in the array using exclusive or operation. All elements that appear twice would sum to zero and the final result would be the element that appears only once.

P16.

We flip a coin n times to generate a binary sequence of size n . This number is in the range of $0 - (2^n - 1)$. If we treat binary sequence $000 \dots 0$ as value 2^n , we get a variable ranging from $1 - 2^n$. This number mod 6 will generate output $0 - 5$ equally if we reject number $2^n > 6^m$ given we want m die flips, r is the rejection rate.

$$m = \lceil n \log_3 2 \rceil, r = 1 - \frac{3^m}{2^n}$$

P17.

Basically, we move both robots in the same direction, the robot which gets to zero first stays at zero, the other robot reverts its direction until it gets to zero as well. Two robots will meet at zero.

```
#include<iostream> using namespace std ;

class robot{

private : int pos;

public : robot(int x){ pos = x; }

void Go Left (){ pos -=1; }

void Go Right (){ pos +=1; }

bool at zero (){
return ( pos == 0 );

} };

int main(){

robot robot1(-8); robot robot2 (7);

int count=0;
while( !robot1.at zero() || !robot2.at zero() ){

if ( robot1.at zero() || robot2.at zero() ){ if( robot1.at zero() ) {

robot2 . Go Left (); count++;

}else{ robot1 . Go Left ();

count++;

}

}else{

robot1.Go Right();

robot2.Go Right(); count++;

}

}

cout << count << endl ;

return 0; }
```