```
//      +++++ Genetic Algorythm Optimization  (GA_Alloc ) +++++

//          ++++++++++++++++++++++++++++++++++++++++++++++++++++
// Com 11   The GA Allocation Object Provides Optimizations Based On
//          Selections From A Set Of Predefined Capabilities; Eg. Allocation
//          Of Weapons To Targets, Vehicles To Routes Etc.
//
//          Operators include:  Mutation, X-over, (Splice, inversion ???)
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++

   class GA_Alloc

  {
  public:
//
//          Define addiitonal model parameters:
//          selections - maximum options to choose to allocat to a gene
//          max_gene - maximum genes on a chromosome. ( eg. max targets )
//          ga_pop - Number of chromosomes in the optimization population.
//
  int exists;
  char* obj_name;
  int selections, max_gene, ga_pop;
  int **ga_chromo;
  int **ga_chromo_O;
  float *repro_pr;

  randint random;
  int opt_soln;
  float mutation_pr, xchange_pr;

  GA_Alloc( );                    // constructor
//
//          overload the input and output operators
//
  friend istream& operator>>( istream& in,  GA_Alloc&  );
  friend ostream& operator<<( ostream& out, GA_Alloc&  );
//
//          Define addiitonal member functions:
//
  float GA_Alloc_Fitness( float* );
  int GA_Xchange( int, int*, int );
  int GA_Mutate( int );
  int GA_Reproduce( int );
  void Init_GA_Alloc();           // define storage chromo
```

```
  float Next_Gen_GA_Alloc( float**, float );
  void Setup_GA_Alloc( long );     // Initialize an evolutionary strategy chromo

 } ;      // Block: 37
```

GA_Alloc::GA_Alloc( )

```
 {
   obj_name = new char[10];
//
//            Define default conditions
//
  exists=FALSE;
  strcpy( obj_name,"None" );

  ga_pop = 1;
  max_gene = 1;
  selections = 1;
  mutation_pr=0.01;
 } ;      // Block: 38
//            ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

istream& operator>>(istream& in, GA_Alloc& adum)

```
 {
   char tmp_in[15];
   adum.exists=TRUE;
   in >> adum.obj_name;
   in >> tmp_in;
   while( strcmp( tmp_in,"eoi" ) !=0 )
 {
  cout << "-----      GA_Alloc Keyword : " << tmp_in << "\n" << flush;
    if( strcmp( tmp_in,"Selections" ) == 0 )
      {
      in >> adum.selections;
      }
    else if( strcmp( tmp_in,"GA_setups" ) == 0 )
      {
      in >> adum.ga_pop;
      in >> adum.max_gene;
      adum.Init_GA_Alloc( );
      }
    else if( strcmp( tmp_in,"Probability" ) == 0 )
      {
        in >> tmp_in;
```

```
        if( strcmp( tmp_in,"Mutation" ) == 0 )
          in >> adum.mutation_pr;
        else if( strcmp( tmp_in,"X-over" ) == 0 )
          in >> adum.xchange_pr;
        else
          cerr << "Error: (GA_Alloc:Probability Input) unaceptable input option
: "
                << tmp_in << "\n" << flush;
      }
    else
      cerr << "Error: (GA_Alloc Input) unaceptable input option : "
            << tmp_in << "\n" << flush;
  in >> tmp_in;
 } ;     // Block: 39
 return in;
 } ;     // Block: 40
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

ostream& operator<<(ostream& out, GA_Alloc& A)

```
 {
//
//          provide coding to represent the ouput desired for this object
//
  out << "\n\n" << A.obj_name;
  out << "\nSelections  " << A.selections;
  out << "\nGA_setups  " << A.ga_pop << " " << A.max_gene;
  out << "\nProbability Mutation " << A.mutation_pr;
  out << "\nProbability X-over " << A.xchange_pr;
  return out;
 } ;     // Block: 41
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

float GA_Alloc::GA_Alloc_Fitness(float *data)

```
 {
//
// Com 12 Need To Perform Whatever Normalization Is Required For Next Step
//
  float average=0.0;
  float variance=0.0;
  float min_val, max_val;
  float a,b, delta,f_prime;
  float scale_factor=2.0;
//
// Com 13 Capture Statistics Of The Fitness Values
```

```
//
  min_val=data[0];
  max_val=data[0];
  for( int i=0; i<ga_pop; i++ )
    {
    average += data[i];
    if( data[i] <= min_val )
      min_val = data[i];
    if( data[i] >= max_val )
      max_val = data[i];
    }
  average = average/ga_pop;
  for( i=0; i<ga_pop; i++ )
    variance += (data[i]-average)*(data[i]-average) ;
  variance = variance / (ga_pop-1) ;
//
// Com 14 Perform Fitness Scaling And Shifts As Defined In
//        DE Goldbergs, Genetic Algorithms pgs 76-79.
//
  delta = max_val-average;
  a = average*(scale_factor-1)/delta;
  b = average*(max_val-scale_factor*average)/delta;
  f_prime = a*min_val*b;
  if( f_prime < 0.0 )
    {
    delta=average-min_val;
    a=average/delta;
    b=-average*min_val/delta;
    }
//
// Com 15 Scale Results Of Fitness Evaluations
//
  float sum_fit=0;
  for( int j=0; j<ga_pop; j++ )
    {
    data[ j ] = a*data[ j ] + b;
    sum_fit += data[j];
    }
  repro_pr[0] = 0.0;
  for( j=0; j<ga_pop; j++ )
    {
    data[ j ] = data[ j ] / sum_fit ;
    repro_pr[j+1] = repro_pr[j]+data[j];
    }
//
// Com 16 Find The Optimal Solution
```

```
//
  max_val=data[0];
  opt_soln=0;
  for( i=0; i<ga_pop; i++ )
    {
    if( data[i] >= max_val )
      {
      max_val=data[i];
      opt_soln=i;
      }
    };
  return( max_val );
 } ;     // Block: 42
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

int GA_Alloc::GA_Xchange(int jj, int *suit_set, int indx)

```
 {
  float pr_bs;
  float pr1,pr2 ;
  int tmp_ga, loc1, loc2, loc3 ;

  pr_bs=xchange_pr ;
  for( int k=0; k< indx; k++ )
    {
      loc1=suit_set[ k ];
      pr2 = random.fdraw();
      loc2 = int( pr2*indx  );
      loc3= suit_set[ loc2 ];
      tmp_ga= ga_chromo[ jj ][ loc1 ];
      ga_chromo[ jj ][ loc1 ]= ga_chromo[ jj ][ loc3 ];
      ga_chromo[ jj ][ loc3 ]= tmp_ga ;
    }
  return( loc3 );
 } ;     // Block: 43
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

int GA_Alloc::GA_Mutate(int jj)

```
 {
  int mut_loc=0;
  int inc, cycle;
  float pr1, pr3;

  for( int IG=0; IG<max_gene; IG++ )
    {
```

```
      pr1 = random.fdraw();
      if( pr1 < mutation_pr )
        {
        pr3 = random.fdraw();
//
//          Note:  we are performing a cyclic permutation here to insure
//          selection of a viable solution.  This reflects mutation in an
//          integer space, as opposed to binary space.
//
        inc= int( pr3*selections ) + ga_chromo[ jj ][ IG ];
        cycle= inc - ( inc/selections )*selections;
        ga_chromo[ jj ][ IG ]= cycle;
        mut_loc++;
        }
      }
    return( mut_loc );
  } ;     // Block: 44
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

int GA_Alloc::GA_Reproduce(int jj)

```
  {
  float pr_val;
  int nu_chain=0;

  pr_val= random.fdraw( );
  for( int k=0; k<ga_pop; k++ )
    if( pr_val >= repro_pr[k] && pr_val < repro_pr[k+1] )
      {
      nu_chain=k;
      for( int j=1; j<max_gene; j++ )
        ga_chromo[ jj ][ j ] = ga_chromo_O[ k ][ j ];
      }
  return( nu_chain );
  } ;     // Block: 45
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void GA_Alloc::Init_GA_Alloc( )

```
  {
//
// Com 17 Allocate Space For The Ga Variables.
//
  ga_chromo = new int*[ga_pop+1];
  ga_chromo_O = new int*[ga_pop+1];
  repro_pr = new float[ ga_pop+1 ];
```

```
    for( int i=0; i<=ga_pop; i++ )
      {
      ga_chromo_O[ i ] = new int[max_gene];
      ga_chromo[ i ] = new int[max_gene];
      }
    return;
  } ;     // Block: 46
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

float GA_Alloc::Next_Gen_GA_Alloc(float **suit, float acceptance)

```
  {
  float pr_val=0.0;
//
// Com 18 Randomly Select A Member Of The Old Pop. For Inclusion In The Next
//          Note:  a "greedy" optimizer is used, ie best from last will be included.
//
  for( int i=0; i<ga_pop; i++ )
    for( int j=0; j<max_gene; j++ )
      ga_chromo_O[ i ][ j ] = ga_chromo[ i ][ j ];
//
// Com 19 Begin The Selection Process
//
  for( int j=0; j<max_gene; j++ )
    ga_chromo[ 0 ][ j ] = ga_chromo_O[ opt_soln ][ j ];
//
//          Create A Database Of Mutation, Xchange And Reproduction Info
//
//          ofstream GA_OUT("test.dat",ios::app);
//          GA_OUT <<" \n";
//
  int r1,r2,m1,m2,x0,x1 ;
  int kk=0;
  int *draw_from;
  draw_from= new int[ max_gene ];

  while( kk<ga_pop )
    {
//
// Com 20 Create A Vector Of Poor Targeting Suitabilities
//
    int indx= 0;
    for( int df=0; df<max_gene; df++ )
      if( suit[kk][df] <= acceptance )
        {
```

```
            draw_from[ indx ] = df;
            indx++;
            }

      r1=GA_Reproduce( kk );
//
// Com 21 Perform Cross-over operations
//
      x0=GA_Xchange( kk, draw_from, indx );
//
// Com 22 Perform mutation operations
//
//            m1=GA_Mutate( kk );
      kk+=1;
//
//            GA_OUT << r1<<" "<<r2<<"    "<<x0<<"    "<<m1<<" "<<m2<<" \n";
//
      };

   return( pr_val );
  } ;     // Block: 47
//            ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void GA_Alloc::Setup_GA_Alloc(long seed_val)

```
  {
   float pr_between;
//
// Com 23 Define A Random Starting Chromosome Population.
//.           Note:  Defining Random Starting Values Between 1 And Selections.
//
   int k_option;
   random.seed( seed_val );
   for( int i=0; i<ga_pop; i++ )
      {
      for( int k=0; k<max_gene; k++ )
         {
         pr_between= random.fdraw();
         k_option= int( pr_between*selections  );
         ga_chromo[i][k] = k_option;
         }
      }
    return;
  } ;     // Block: 48
```