Options	File (by build order)	Size	Volume
	Runtime.lib	22836	Quercus:…:THINK Pascal 4.…
	Interface.lib	12812	Quercus:…:THINK Pascal 4.…
D N V R	MyGlobals	0	Quercus:…:Arthromorphs (…
D N V R	Error_Alert.Pas	36	Quercus:…:Arthromorphs (…
D N V R	SetupBoxes	1098	Quercus:…:Arthromorphs (…
D N V R	Ted.Pas	13582	Quercus:…:Arthromorphs (…
D N V R	Richard	1332	Quercus:…:Arthromorphs (…
D N V R	InitTheMenus.Pas	132	Quercus:…:Arthromorphs (…
D N V R	Engineering_Windo…	3908	Quercus:…:Arthromorphs (…
D N V R	Genome_Window.Pas	402	Quercus:…:Arthromorphs (…
D N V R	Breeding_Window.P…	966	Quercus:…:Arthromorphs (…
D N V R	Preferences.Pas	1058	Quercus:…:Arthromorphs (…
D N V R	About_Arthromorp…	516	Quercus:…:Arthromorphs (…
D N V R	HandleTheMenus.Pas	424	Quercus:…:Arthromorphs (…
D N V R	Initialize	222	Quercus:…:Arthromorphs (…
D N V R	Brand_New.Pas	1636	Quercus:…:Arthromorphs (…
	Total Code Size	60960	

```
{Arthromorphs   by Richard Dawkins and Ted Kaehler}
{Ted's initial version: 25 Nov 90}
{Current version: 8 Dec 90}
{Since we both are confused by handles and pointers in Pascal, this does not use any of either!}

{There is a Record called Atom that holds a little part of an animal.  It has fields for a Height, }
{a Width, and an Angle.}
{   When it is used to describe a Segment, Height and Width are for the oval,}
{       and Angle is not used}
{   When it is used to describe a Joint, Height is the thickness of the leg-part,}
{       Width is the length, and Angle is the angle from the previous joint}
{   When it is used to describe a Claw,  Height is the thickness of the claw-part,}
{       Width is the length, and Angle is the between the claw halves}

{Remember that the true Joint length is the multiplication of all the factors:}
{The Animal's joint length, this Section's joint length, this Segment's joint length, and the Joint's own joint length.}
{Thus a Segment actually has three parts: its factor for Segment size, its }
{   factor for Joint size, and its factor for Claw size.  Each of these are Atoms.  Thus a}
{   Segment has three Atoms.  They are distinguished by having different kinds: SegmentTrunk,}
{   SegmentJoint, and SegmentClaw.}
{An Animal-record also has three Atoms in it AnimalTrunk, AnimalJoint, and AnimalClaw.}

{How are Atoms hooked together?  Here is a sample Animal.  Each line is an Atom, but I don't}
{   show the values inside it, like Height: 20 Width: 30, etc.}
{AnimalTrunk}
{   AnimalJoint}
{   AnimalClaw}
{       SectionTrunk}
{           SectionJoint}
{           SectionClaw}
{               SegmentTrunk}
{                   SegmentJoint}
{                   SegmentClaw}
{                       Joint}
{                       Joint}
{                       Joint}
{                           Claw}
{               SegmentTrunk}
{                   SegmentJoint}
{                   SegmentClaw}
{                       Joint}
{                       Joint}
{                       Joint}
{                           Claw}

{A Section sets the tone for all segments within it:  Head, Thorax, Abdomen are sections}

{In the above set of Atoms, there are two fields for connecting Atoms together.}
{   NextLikeMe  hooks the atom to the next atom on the same level.}
{   FirstBelowMe  hooks the atom to the first atom on a lower level.}
{Look at the diagram above.  When an atom points to another with NextLikeMe, they}
{have the same level of indentation.  When an atom points to another with }
{FirstBelowMe, the atom is indented one more level.}
{The first SegmentTrunk points way down to the second SegmentTrunk with NextLikeMe.}
{The Joints point to the next with NextLikeMe.  However, the AnimalClaw}
{points to SegmentTrunk using FirstBelowMe.  Note that the three atoms that}
{make up an Animal are split.  AnimalJoint is pointed to with FirstBelowMe even}
{though it is part of the animal description.  I had to do this so that AnimalTrunk could use its}
{NextLikeMe to point at the next animal.  Likewise with Segments.}

{All atoms are stored in a big Array called the BoneYard.  You find an atom}
{by knowing its index (the integer that is its place in the array).  The two "pointers" NextLikeMe }
{and FirstBelowMe are not pointers at all, but simply integers.}
```

```
{An indivudual Animal can have its atoms spread out all over the BoneYard, but }
{each atom in it holds the index of the next atom in it.  Thus we can walk down }
{the parts of an animal very easily.  Atoms that are not being used are labelled Free.}


unit myGlobals;
interface
  const
    MaxBoxes = 15;

  type
    Pressure = (positive, zero, negative);
    Concentration = (FirstSegmentOnly, LastSegmentOnly, AnySegment);

  var
    NRows, NCols: LongInt;
    MidBox:  integer;
    Special, NBoxes, Hot: integer;
    Prect:  rect;
    box: array[0..MaxBoxes]  of rect;
    upregion: RgnHandle;
    centre: array[0..MaxBoxes]  of point;
    BreedWindow:  WindowPtr;
    VerticalOffset, HorizontalOffset, OldVerticalOffset, OldHorizontalOffset, thickscale: integer;
    wantColor, sideways, centring, resizing, startingUp: boolean;
    TrunkMut, LegsMut, ClawsMut, AnimalTrunkMut, AnimalLegsMut, AnimalClawsMut: Boolean;
    SectionTrunkMut, SectionLegsMut, SectionClawsMut, SegmentTrunkMut, SegmentLegsMut, SegmentClawsMut: Boolean;
    WidthMut, HeightMut, AngleMut, DuplicationMut, DeletionMut, AgreeToExit: boolean;
    MutationPressure:  pressure;
    FocusOfAttention:  concentration;
    Overlap:  real;
    BreedingWindow:  WindowPtr;


implementation
end.
```

```
unit  Error_Alert;

{File name:  Error_Alert.Pas  }
{Function: Handle a Alert}
{This is a CAUTION alert, it is used to inform the user that if the current path}
{is taken then data may be lost.  The user can change the present course and}
{save the data.  This is the type of alert used to tell the user that he needs to}
{save the data before going on.}
{This alert is called when:   }
{  }
{The choices in this alert allow for:   }
{  }
{History: 12/12/90 Original by Prototyper.    }
{                  }

interface

  procedure  A_Error_Alert;

implementation

  procedure  A_Error_Alert;
    const
      I_OK = 1;
    var
      itemHit: Integer;         {Get the selection ID in here}

  begin                           {Start of alert handler}

            {Let the OS handle the Alert and wait for a result to be returned}
    itemHit := CautionAlert(6, nil);{Bring in the alert resource}

            {This is a button that may have been pressed.}
            {This is the default selection, when RETURN is pressed.}
    if (I_OK = itemHit) then{See if this item was selected}
      begin                     {Start of handling if this was selected}
      end;                      {End of handling if this was selected}


  end;                            {End of procedure}

end.                              {End of unit}
```

```
unit boxes;
interface
  uses
    myGlobals;
  procedure SetUpBoxes;
  procedure Slide (LiveRect, DestRect: Rect);
  procedure DrawBoxes;

implementation


  function sgn (x: INTEGER): INTEGER;
  begin
    if x < 0 then
      sgn := -1
    else if x > 0 then
      sgn := 1
    else
      sgn := 0
  end; {sgn}


  procedure Slide (LiveRect, DestRect: Rect);
    var
      SlideRect: RECT;
      xDiscrep, yDiscrep, dh, dv, dx, dy, xmoved, ymoved, xToMove, yToMove, distx, disty: INTEGER;
      TickValue: LONGINT;

  begin {PenMode(PatXor); FrameRect(LiveRect); PenMode(PatCopy);}
    xMoved := 0;
    yMoved := 0;
    distx := DestRect.left - LiveRect.left;
    disty := DestRect.bottom - LiveRect.bottom;
    dx := sgn(distx);
    dy := sgn(disty);
    xToMove := ABS(distx);
    yToMove := ABS(disty);
    xMoved := 0;
    yMoved := 0;
    UnionRect(LiveRect, DestRect, SlideRect);
    ObscureCursor;
    repeat
      TickValue := TickCount;
      xDiscrep := xToMove - xMoved;
      if xDiscrep <= 20 then
        dh := xDiscrep
      else
        dh := (xDiscrep) div 2;
      yDiscrep := yToMove - yMoved;
      if Ydiscrep <= 20 then
        dv := yDiscrep
      else
        dv := (yDiscrep) div 2;
      repeat
      until TickValue <> TickCount;
      if (xMoved < xToMove) or (yMoved < yToMove) then
        ScrollRect(SlideRect, dx * dh, dy * dv, upregion);
      xMoved := xMoved + ABS(dh);
      yMoved := yMoved + ABS(dv);
    until (xMoved >= xToMove) and (yMoved >= yToMove);
  end; {Slide}

  procedure DrawBoxes;
    var
      j:  integer;
  begin
    for j := 1 to NBoxes do
      framerect(box[j]);
    PenSize(3, 3);
    FrameRect(box[MidBox]);
    PenSize(1, 1);
  end;

  procedure SetUpBoxes;
    var
      j, l, t, row, column, boxwidth, height, midBox: INTEGER;
      inbox:  rect;

  begin
    Prect := BreedingWindow^.PortRect;
    with Prect do
      begin
        bottom := bottom - 20;
        right := right - 20;
      end;
    EraseRect(Prect);
    j := 0;
    NBoxes := NRows * NCols;
    MidBox := NBoxes div 2 + 1;
    with Prect do
      begin
        boxwidth := (right - left) div ncols;
        height := (bottom - top) div nrows;
        for row := 1 to NRows do
          for column := 1 to NCols do
            begin
              j := j + 1;
              l := left + boxwidth * (column - 1);
              t := top + height * (row - 1);
              setrect(box[j], l, t, l + boxwidth, t + height);
              if j <> MidBox then
                FrameRect(box[j]);
              with box[j] do
                begin
                  Centre[j].h := left + boxwidth div 2;
                  Centre[j].v := top + height div 2
                end;
            end; {row & column loop}
      end; {WITH Prect}
    PenSize(3, 3);
    FrameRect(box[MidBox]);
    PenSize(1, 1);
    with Prect do
      begin
        left := box[1].left;
        right := Box[NBoxes].right;
        top := box[1].top;
        bottom := box[Nboxes].bottom
      end;
    SetRect(Box[0], 261, 28, 483, 320); {Special box for Engineering window}
    with box[0] do
      begin
        boxwidth := right - left;
        height := bottom - top;
        Centre[0].h := left + boxwidth div 2;
        Centre[0].v := top + height div 2
        end;
      end; {setup boxes}

    end.
```

```
unit Ted;
interface
  uses
    MyGlobals, boxes, Error_Alert;
  const
    YardSize = 5000;
    miniSize = 200;
    scale = 10;
{2500 would allow 18 Animals with 15 segments each and 4 joints per segment.}
  type
    AtomKind = (Free, AnimalTrunk, AnimalJoint, AnimalClaw, SectionTrunk, SectionJoint, SectionClaw, SegmentTrunk,
 SegmentJoint, SegmentClaw, Joint, Claw);
    Atom = record
        Kind: AtomKind;
        Height: real;      {also used for Thickness of a Joint}
        Width: real;       {also used for Length of a Joint}
        Angle: real;       {also used in an AnimalTrunk to store the number of atoms in the animal}
                           {also used in SectionTrunk to store the Overlap of segments}
                           {also used in SegmentTrunk to store the rank number of the segment}
        NextLikeMe: Integer;     {where to look in the BoneYard for the next atom. 0 means end of chain}
{Also used in AnimalTrunk to store Gradient gene, slightly more or less than 100.  Treat as Percentage}
        FirstBelowMe: Integer;       {where to look in the BoneYard for the next atom. 0 means end of chain}
      end;
    AtomPtr = ^Atom;
    AtomHdl = ^AtomPtr;
    AtomArray = array[1..Yardsize] of AtomHdl;       {for the real thing, use 2500}
    SmallAtomArray = array[1..miniSize] of AtomHdl;     {Just holds one animal, compactly}
    AnimalStarts = array[0..MaxBoxes] of integer;

    LevelLocs = array[1..10] of integer;     {stores indexes of where we are when travelling through an animal}
        {to copy it.  1 spare, 2 AnimalTrunk, 3 AnimalJoint, 4  SectionTrunk, 5 SectionJoint, 6 SegmentTrunk, }
            {7 SegmentJoint, 8 Joint, 9 Claw, 10 spare}
    KindsData = array[AtomKind] of integer;     {a number for each kind of Atom}
    CumParams = array[1..9] of real;        {where the AnimalTrunk.Width is multiplied by SegmentTrunk.Width}
  var
    BoneYard: AtomArray;      {all atoms live here.  We index it to look at atoms}
    MiniYard: SmallAtomArray;
    RecordTop, RecordBottom, CurrentGenome: integer;      {index of first atom on an Animal}
    BreedersChoice: AnimalStarts;     {indexes of starts of all the Animals on the screen}
    NorthPole, SouthPole, EastPole, WestPole, FreePointer, MiniFree: integer;           {start searching from here for free bloc
    ParamOffset: KindsData;     {Tells where Height, Width, Angle go in a CumParams.  see Draw}
    AnimalPicture: array[0..MaxBoxes] of PicHandle;
    Midriff, SegmentCounter, SecondSegmentAtomNo: integer;
    f: file of  Atom;
    naive: boolean;
    GradientFactor: real;

  function CountAtoms (which: integer): integer;
  procedure NewMinimal;
  procedure InitBoneYard;
  procedure Breed;
  procedure evolve (MLoc: point);
{***call this as Evolve(MyPt) from Do_Breeding_Window immediately after defining MyPt}
  procedure UpDateAnimals;
  procedure SaveArthromorph;
  procedure LoadArthromorph;
  procedure StartDocument;
  procedure flipWantColor;
  procedure QuitGracefully; {Call right at end of whole program}
  procedure Draw (which: integer; params: CumParams; x, y, xCenter: integer; var ySeg: integer);
  procedure DrawInBox (BoxNo: integer);
  procedure TellError (what: string);
  procedure Tandem (target: integer);
```

```
implementation
  procedure TellError (what: string);
  begin
    ParamText(what, '', '', '');
    A_Error_Alert;
  end;

  function randint (Max: Integer): Integer;
    var
      r: integer;
  begin
{delivers integer between 1 and Max;}
    repeat
      r := ABS(Random) mod (Max + 1)
    until r > 0;
    randint := r;
  end;

{Basic handling of Atoms}
  procedure InitBoneYard;     {Call just once at the beginning}
    var
      this: Atom;
      which: integer;
  begin
    for which := 1 to YardSize do
      BoneYard[which] := AtomHdl(NewHandle(SizeOf(Atom)));
    for which := 1 to MiniSize do
      begin
        MiniYard[which] := AtomHdl(NewHandle(SizeOf(Atom)));
        MiniYard[which]^^.kind := free;
      end;
    FreePointer := 1;
    for which := 1 to YardSize do
      begin
        BoneYard[which]^^.Kind := Free;
        BoneYard[which]^^.NextLikeMe := 0;    {Don't count on this}
      end;
    ParamOffset[AnimalTrunk] := 1;        {where in a CumParams the Width of an AnimalTrunk gets multiplied in}
    ParamOffset[AnimalJoint] := 4;
    ParamOffset[AnimalClaw] := 7;
    ParamOffset[SectionTrunk] := 1;
    ParamOffset[SectionJoint] := 4;
    ParamOffset[SectionClaw] := 7;
    ParamOffset[SegmentTrunk] := 1;
    ParamOffset[SegmentJoint] := 4;
    ParamOffset[SegmentClaw] := 7;
    ParamOffset[Joint] := 4;
    ParamOffset[Claw] := 7;
  end;

  function Allocate: Integer;
    var
      this: Atom;
      oldFreePtr, which: integer;
  begin
    oldFreePtr := FreePointer;
    which := FreePointer;
    repeat
      this := BoneYard[which]^^;
      which := which + 1;        {remember its one bigger}
    until (this.Kind = Free) or (which > YardSize);
    if which > YardSize then
```

```
      begin
        which := 1;
        repeat
          this := BoneYard[which]^^;
          which := which + 1;
        until (this.Kind = Free) or (which > oldFreePtr);
        if which = oldFreePtr + 1 then
          TellError('Morphs are too complex');
      end;
    FreePointer := which;
    if which <= 1 then
      TellError('Allocate tried to put out less than 1');
    if which > Yardsize then
      TellError('Allocate tried to put out >Yardsize');
    Allocate := which - 1;       {undo the +1 above}
  end;

  procedure Deallocate (which: integer);
  begin
    BoneYard[which]^^.Kind := Free;        {toss it back}
  end;

{Creating and destroying Animals}
  procedure Kill (which: integer);
    {Destroy this animal.   Mark all of its Atoms as Free again.}
    {Recursively step through the animal}
    var
      this: Atom;
  begin
    this := BoneYard[which]^^;
    if this.FirstBelowMe <> 0 then
      Kill(this.FirstBelowMe);
    if (this.NextLikeMe <> 0) and (this.kind <> AnimalTrunk) then
      Kill(this.NextLikeMe);
    Deallocate(which);       {Free this Atom}
  end; {Kill}

  function Copy (which: integer): integer;
    var
      newPlace: integer;
  begin
    {Duplicate this entire animal.   Return the index of the start of the new animal.}
    {It is a very good idea to Kill the old animal first.  That way, we can reuse its atoms.}
    newPlace := Allocate;        {Grab a new atom}
    BoneYard[NewPlace]^^ := BoneYard[which]^^;
    if BoneYard[which]^^.FirstBelowMe <> 0 then
      BoneYard[NewPlace]^^.FirstBelowMe := Copy(BoneYard[which]^^.FirstBelowMe);
    if (BoneYard[which]^^.NextLikeMe <> 0) and (BoneYard[which]^^.kind <> AnimalTrunk) then
      BoneYard[NewPlace]^^.NextLikeMe := Copy(BoneYard[which]^^.NextLikeMe);
    Copy := newPlace;            {Return the index of the new one}
  end;

  function CopyExceptNext (which: integer): integer;
    var
      newPlace: integer;
  begin
    {Duplicate Subtree starting at the atom which, but don't copy NextLikeMe.  Leave old value there}
    {Copy the things I own, but not the things after me}
    newPlace := Allocate;        {Grab a new atom}
    BoneYard[NewPlace]^^ := BoneYard[which]^^;
    if BoneYard[which]^^.FirstBelowMe <> 0 then
      BoneYard[NewPlace]^^.FirstBelowMe := Copy(BoneYard[which]^^.FirstBelowMe);        {Normal COPY from here on}
    CopyExceptNext := newPlace;        {Return the index of the new one}
```

```
end;


function FindNth (which, pick: integer; var count: integer): integer;
  {travel over the Animal, counting Atoms and return the Nth}
begin
  count := count + 1;
  if BoneYard[which]^^.kind = SegmentTrunk then
    SegmentCounter := Segmentcounter + 1;
  if segmentCounter = 2 then
    SecondSegmentAtomNo := count;
  if count >= pick then
    FindNth := which        {We are done!}
  else
    with BoneYard[which]^^ do
      begin
        if FirstBelowMe <> 0 then
          FindNth := FindNth(FirstBelowMe, pick, count);
        if not (count >= pick) then
          if (NextLikeMe <> 0) then
            FindNth := FindNth(NextLikeMe, pick, count);
        if not (count >= pick) then
          FindNth := 0;     {not there yet}
      end;
end;

procedure CountSeg (which: integer);
  var
    this: Atom;
begin
  this := BoneYard[which]^^;
  with this do
    begin
      if kind = SegmentTrunk then
        begin
          SegmentCounter := SegmentCounter + 1;
          BoneYard[which]^^.angle := SegmentCounter;
        end;
      if FirstBelowMe <> 0 then
        CountSeg(FirstBelowMe);
      if (NextLikeMe <> 0) and (kind <> AnimalTrunk) then
        CountSeg(NextLikeMe);
    end
end;

function CountAtoms (which: integer): integer;
  {travel over the Animal, counting Atoms}
  var
    count: integer;
begin
  count := 1;   {count me}
  with BoneYard[which]^^ do
    begin
      if FirstBelowMe <> 0 then
        count := count + CountAtoms(FirstBelowMe);
      if (NextLikeMe <> 0) and (kind <> AnimalTrunk) then
        count := count + CountAtoms(NextLikeMe);
    end;
  CountAtoms := count;   {Me and all below me}
end;

function GetFactor: real;        {How much to grow or shrink a Length or Height or Angle}
  var
```

```
      choose: integer;
  begin
    case MutationPressure of
      positive:
        choose := 2 + randint(2);
      zero:
        choose := randint(4);
      negative:
        choose := randint(2);
    end; {cases}
    case choose of
      1:            {Richard, you can play with these factors}
        GetFactor := 0.50;
      2:
        GetFactor := 0.9;
      3:
        GetFactor := 1.1;
      4:
        GetFactor := 1.5;
    end; {cases}
  end;


  function DoDelete (which: integer): boolean;
    {Delete a section of the animal somewhere near the atom which.}
    {Caller must correct the AtomCount of the whole animal.  Return false if failed}
    var
      parent, chain: integer;
    {Must have a hold on the atom above what we delete.  If chosen atom is:  }
    {AnimalTrunk    delete first Sec}
    {   AnimalJoint    delete first Sec}
    {   AnimalClaw   delete first Sec}
    {       SectionTrunk delete next Sec}
    {           SectionJoint       delete first Seg}
    {           SectionClaw        delete first Seg}
    {               SegmentTrunk        delete next Seg}
    {                   SegmentJoint     delete first Joint}
    {                   SegmentClaw      delete first Joint}
    {                       Joint               delete next Joint}
    {                       Joint               delete next Joint}
    {                       Joint               delete Claw}
    {                           Claw               fail}
    {Also fail if trying to delete last example of a Kind}
  begin
    parent := which;
    DoDelete := false;    {unless we actually succeed in killing one}
    if (BoneYard[Parent]^^.Kind = AnimalTrunk) then
      begin
        parent := BoneYard[Parent]^^.FirstBelowMe;      {AinmalJoint}
      end;
    if (BoneYard[Parent]^^.Kind = AnimalJoint) or (BoneYard[Parent]^^.Kind = SectionJoint) or (BoneYard[Parent]^^.Kind =
SegmentJoint) then
      begin
        parent := BoneYard[Parent]^^.FirstBelowMe;      {AinmalClaw is parent}
      end;
    if parent <> 0 then
      with BoneYard[Parent]^^ do
        if (Kind = SectionTrunk) or (Kind = SegmentTrunk) or (Kind = Joint) then
          begin       {Delete NextLikeMe of parent}
            if (NextLikeMe <> 0) then
              begin
                chain := BoneYard[NextLikeMe]^^.NextLikeMe;         {May be 0}
                BoneYard[NextLikeMe]^^.NextLikeMe := 0;     {So Kill won't get the rest of chain}
                Kill(NextLikeMe);    {won't be killing last one, since parent qualifies as one}
                NextLikeMe := chain;
                DoDelete := true;
              end;
          end
        else     {Try to delete FirstBelow}
          if (FirstBelowMe <> 0) then         {we know FirstBelow exists}
            begin
              chain := BoneYard[FirstBelowMe]^^.NextLikeMe;        {Atom after one we will delete}
              BoneYard[FirstBelowMe]^^.NextLikeMe := 0;
              if (chain <> 0) then            {FirstBelow is not only one }
                begin
                  Kill(FirstBelowMe);
                  FirstBelowMe := chain;
                  DoDelete := true;
                end;
            end;
  end; {DoDelete}

  procedure Tandem (target: integer);
    var
      extraclaw: integer;
      targetAtom: Atom;
        {If Dup and target is second or third part of an Animal, Section, or Segment,}
        {Then jump down to the next part of the animal}
  begin
    targetAtom := BoneYard[target]^^;
    if (targetAtom.Kind = AnimalJoint) or (targetAtom.Kind = SectionJoint) or (targetAtom.Kind = SegmentJoint) then
      begin
        target := BoneYard[target]^^.NextLikeMe;    {AinmalClaw}
        targetAtom := BoneYard[target]^^;        {fetch new atom}
      end;
    if (targetAtom.Kind = AnimalClaw) or (targetAtom.Kind = SectionClaw) or (targetAtom.Kind = SegmentClaw) then
      target := BoneYard[target]^^.FirstBelowMe;
{SectionTrunk .. where we want to be }
    with BoneYard[target]^^ do
      begin
        NextLikeMe := CopyExceptNext(target);  {Insert copy of me after me}
                    {CopyExceptNext makes sure NextLikeMe of copy now points to old NextLikeMe of target}
                    {So brothers are kept, and new subtree is inserted}
        if (Kind = Joint) and (FirstBelowMe <> 0) then      {last joint has claw.  When duplicate, get rid of extra claw}
          begin
            extraClaw := FirstBelowMe;
            FirstBelowMe := 0;
            Kill(extraClaw);
          end;
      end;
    BoneYard[BreedersChoice[MidBox]]^^.Angle := CountAtoms(BreedersChoice[MidBox]);        {A little wasteful to count entire
 again}
  end; {Tandem}

  function Mutate (which: integer): boolean;
    {Mutate first picks an atom randomly from the Animal.}
    {    From num of atoms, picks one and step down to it}
    {        Flip a coin for what to do: change Height, Width, Angle, Dup part, Delete part, Flip angle}
    {            Test if legal to do it and do it (else return false)}
    {                Delete does not delete the first-and-only of its Kind}
    {Forbid: Angle mod if none, delete last Section, or Seg }
    {        Delete Animal, Dup Animal,   Delete Claw, Dup Claw}
    {Range limits on some modifications??  Only angles can be negative.}
    var
      size, pick, count, target, change, extraclaw, thisSegment, lastSegment, AtomNumber: integer;
      this, targetAtom: Atom;
      OK, MutOK, CouldBe: boolean;
      factor:  real;
  begin
    this := BoneYard[which]^^;
    if this.Kind <> AnimalTrunk then
      TellError('Not an animal');
    SecondSegmentAtomNo := 0;
    AtomNumber := CountAtoms(which);
    LastSegment := SegmentCounter;
    size := trunc(this.Angle);        {As a convention, we keep the number of Atoms in this animal in AnimalTrunk's Angle field}
    pick := Randint(size);        {a number from 1 to size.  Index of the atom we will modify}
    count := 0;
    target := FindNth(which, pick, count);       {find the Nth atom}
    if target = 0 then
      begin
        TellError('Atom count is wrong.  Fatal.  Quitting');        {Aren't pick atoms in this Animal}
        exitToShell
      end;
    targetAtom := BoneYard[target]^^;

    {Decide what to do}
    change := randint(7);        {seven basic operations}
            { 1 twiddle Height, 2 twiddle Width, 3 twiddle Angle, 4 Duplicate entire subtree, 5 Delete subtree}
            { 6 reverse an angle , 7 reverse sign of Gradient}
    if (change = 7) and (targetAtom.kind = AnimalTrunk) then
      BoneYard[target]^^.NextLikeMe := -BoneYard[target]^^.NextLikeMe;
    if (change = 4) then
        {If Dup and target is second or third part of an Animal, Section, or Segment,}
        {Then jump down to the next part of the animal}
      begin
        if (targetAtom.Kind = AnimalJoint) or (targetAtom.Kind = SectionJoint) or (targetAtom.Kind = SegmentJoint) then
          begin
            target := BoneYard[target]^^.NextLikeMe;    {AinmalClaw}
            targetAtom := BoneYard[target]^^;        {fetch new atom}
          end;
        if (targetAtom.Kind = AnimalClaw) or (targetAtom.Kind = SectionClaw) or (targetAtom.Kind = SegmentClaw) then
          target := BoneYard[target]^^.FirstBelowMe;
{SectionTrunk .. where we want to be }
      end;

    MutOK := false;
    with BoneYard[target]^^ do
      case kind of
        AnimalTrunk:
          if AnimalTrunkMut then
            MutOK := true;
        AnimalJoint:
          if AnimalLegsMut then
            MutOK := true;
        AnimalClaw:
          if AnimalClawsMut then
            MutOK := true;
        SectionTrunk:
          if SectionTrunkMut then
            MutOK := true;
        SectionJoint:
          if SectionLegsMut then
            MutOK := true;
        SectionClaw:
          if SectionClawsMut then
            MutOK := true;
        SegmentTrunk:
          if SegmentTrunkMut then
```

```
            MutOK := true;
        SegmentJoint:
          if SegmentLegsMut then
            MutOK := true;
        SegmentClaw:
          if SegmentClawsMut then
            MutOK := true;
        Joint:
          if LegsMut then
            MutOK := true;
        Claw:
          if ClawsMut then
            MutOK := true;
        otherwise
          MutOK := false;
      end; {cases }

    case FocusOfAttention of
      FirstSegmentOnly:
        if SecondSegmentAtomNo > 0 then
          begin
            if count < SecondSegmentAtomNo then
              begin
                with BoneYard[target]^^ do
                  CouldBe := (kind = SegmentTrunk) or (kind = SegmentJoint) or (kind = SegmentClaw) or (kind = joint) or (kind =
 claw);
                if not CouldBe then
                  MutOK := false;
              end
          end
        else
          MutOK := false;
      LastSegmentOnly:
        if SegmentCounter <> lastSegment then
          MutOk := false;
      AnySegment:
        ;
{No need for action.  MutOK retains its present value}
    end; {cases}


    if MutOK then
      with BoneYard[target]^^ do
        begin
          OK := true;
          if ((change = 4) or (change = 5)) and ((Kind = Claw)) then{(Kind = AnimalTrunk) or}
            OK := false; {Forbid delete or dup of claw}
          if ((change = 3) or (change = 6)) and ((Kind = AnimalTrunk) or (Kind = SegmentTrunk)) then
            OK := false;      {These atoms have no Angle part. SectionTrunk does, because 'angle' is overlap, by convention}
          if OK then
            begin
              if (change = 4) then
                begin
                  if DuplicationMut then
                    begin
                      if kind = AnimalTrunk then
                        NextLikeMe := NextLikeMe + 1
                      else{Special case, means GradientFactor}
                        NextLikeMe := CopyExceptNext(target);  {Insert copy of me after me}
                    {CopyExceptNext makes sure NextLikeMe of copy now points to old NextLikeMe of target}
                    {So brothers are kept, and new subtree is inserted}
                      if (Kind = Joint) and (FirstBelowMe <> 0) then        {last joint has claw.  When duplicate, get rid of extra c
                        begin
```

```
                          extraClaw  :=  FirstBelowMe;
                          FirstBelowMe  :=  0;
                          Kill(extraClaw);
                        end;
                      BoneYard[which]^^.Angle  :=  CountAtoms(which);        {A little wasteful to count entire animal again}
                    end
                  else
                    OK := false;
                end; {change=4}
              if (change < 4) then
                begin
                  factor  :=  GetFactor;        {See table above}
                  case change of
                    1:
                      begin
                        if HeightMut then
                          Height  :=  Height  *  factor
                        else
                          OK  :=  false;
                      end;
                    2:
                      begin
                        if WidthMut then
                          Width  :=  Width  *  factor
                        else
                          OK  :=  false;
                      end;
                    3:
                      begin
                        if AngleMut then
                          begin
                            Angle  :=  Angle  *  factor;
                            if (kind = SectionTrunk) then
                              begin
                                Angle  :=  abs(angle); {forbid backward overlaps}
                                if Angle > 1 then
                                  Angle  :=  1; {Otherwise disembodied segments}
                              end;
                          end
                        else
                          OK  :=  false;
                      end;
                  end; {cases}
                end;
              if (change = 5) then
                begin
                  if DeletionMut then
                    begin
                      if kind = AnimalTrunk then
                        NextLikeMe := NextLikeMe - 1; {special case. by convention means GradientFactor}
{Delete.  Complex because we need to talk to the atom above where we delete}
                      OK := DoDelete(target);  {there is a problem here}
                      if OK then
                        BoneYard[which]^^.Angle  :=  CountAtoms(which);
        {A little wasteful to count entire animal again}
                    end
                  else
                    OK := false;
                end;
              if (change = 6) and (kind <> SectionTrunk) then
                begin
                  if AngleMut then
                    Angle  :=  -1.0  *  Angle {reverse an angle}
                  else
                    OK := false;
                end
            end;
        end;
    Mutate := OK and MutOK;
  end;

  function Reproduce (which: integer): integer;
    {Reproduce copies an animal and calls Mutate}
    {Please kill the old animal before calling this.  We may need to use his atoms.}
    var
      counter, new: integer;
      done: boolean;
  begin
    counter := 0;
    new := Copy(which);
    repeat
      counter := counter + 1;
{if counter = 100 then}
{SetCursor(GetCursor(watchCursor)^^);}
      done := Mutate(new);     {If it fails, just try again until we succeed at changing something}
    until done or (counter > 1000);
    if counter > 1000 then
      begin
        TellError('Timed out, perhaps attempting impossible duplication or deletion');
        Reproduce := which;
      end
    else
      Reproduce := new;     {Return it}
{SetCursor(GetCursor(-16000)^^);}
 {Arrow cursor}
  end;

  procedure DrawLine (x, y, endx, endy, thick: integer);
    procedure Dline (x, y, endx, endy, thick: integer);
    begin
{thick := round(thick div thickscale);}
{if thick < 1 then thick := 1;}
      if endy < NorthPole then
        NorthPole := endy;
      if endy > SouthPole then
        SouthPole := endy;
      if endx < WestPole then
        WestPole := endx;
      if endx > EastPole then
        EastPole := endx;
      PenSize(thick, thick);
      MoveTo(x - thick div 2, y - thick div 2);
      LineTo(endX - thick div 2, endY - thick div 2);
      PenSize(1, 1);
    end;
  begin
    if sideways then
      Dline(y, x, endy, endx, thick)
    else
      Dline(x, y, endx, endy, thick);
  end; {Drawline}

  procedure DrawOval (x, y, width, height: integer);
    procedure DOval (x, y, width, height: integer);
      var
        r: rect;
  begin
    with r do
      begin
        left := x;
        top := y;
        right := left + width;
        bottom := top + height;
        if top < NorthPole then
          NorthPole := top;
        if bottom > SouthPole then
          SouthPole := bottom;
        if left < WestPole then
          WestPole := left;
        if right > EastPole then
          EastPole := right;
      end;
    if WantColor then
      begin
        backcolor(GreenColor);
        eraseOval(r)
      end
    else
      fillOval(r, ltgray);
    pensize(2, 2);
    frameOval(r);
    pensize(1, 1);
    backColor(whiteColor);
  end;
begin
  if sideways then
    DOval(y, x, height, width)
  else
    DOval(x, y, width, height);
end; {DrawOval}

procedure DrawSeg (x, y: integer; width, height: real);
  {We must adjust the position before drawing the oval}
  var
    halfW: integer;
begin
  width := width;
  height := height;
  halfW := round(width / 2);
  DrawOval(x - halfW, y, round(width), round(height));
  forecolor(BlackColor);
  {convert from center of oval to left corner}
end;{DrawSeg}


procedure DrawClaw (which: integer; params: CumParams; x, y, xCenter: integer);
  {Draw a claw, note that we don't use which at all.  Param info is already folded in}
  var
    oldX, oldY, leftOldX, leftX, thick: integer;
    ang: real;
begin
  foreColor(RedColor);
  oldX := x;
  oldY := y;
  ang := params[9] / 2.0;
      {half claw opening, in radians}
  x := round(x + params[8] * sin(ang));     {line end point    width*sine(angle)}
  y := round(y + params[8] * cos(ang));     {line end point}
  thick := 1 + trunc(params[7]);     {1 is minimum thickness}
```

```
    DrawLine(oldX, oldY, x, y, thick);      {right side, top of claw}

    leftX := xCenter - (x - xCenter);       {do the left side, top of claw}
    leftOldX := xCenter - (oldX - xCenter);
    DrawLine(leftOldX, oldY, leftX, y, thick);

    {Bottom of the claw}
    y := round(y - 2.0 * params[8] * cos(ang));
    DrawLine(oldX, oldY, x, y, thick);              {right side}
    DrawLine(leftOldX, oldY, leftX, y, thick); {left side}
    ForeColor(BlackColor);
  end;

  procedure Draw (which: integer; params: CumParams; x, y, xCenter: integer; var ySeg: integer);
    {Starting at the atom 'which', multiply its numbers into the array of params.}
    {At the bottom, draw the part starting at x,y}
    {params accumulates the final Joint width, Claw angle, etc.}
    {params: 1 Seg height, 2 Seg width, 3 (not used), 4 Joint thickness, 5 Joint length, 6 Joint angle,}
    {   7 Claw thickness, 8 Claw length, 9 Claw angle between pincers}
    {x,y are current local point, xCenter is the centerline of the animal (left and right Joints need this)}
    var
      myPars: CumParams;
      j, oldX, oldY, leftOldX, leftX, offset, thick: integer;
      ang, jointscale, theFactor: real;
      rankstring:  str255;

  begin
    jointscale := 0.5;
    myPars := params;
    {local copy so next segment builds on section above, not this segment}
    with BoneYard[which]^^ do
      begin
        if kind = AnimalTrunk then
          begin
            GradientFactor := NextLikeMe;
            if gradientFactor > 1000 then
              sysbeep(1);
          end;
        offset := ParamOffset[Kind];        {where in params to begin, see InitBoneYard}
        params[offset] := params[offset] * Height;       {fold in this atom's params}
        params[offset + 1] := params[offset + 1] * Width;
        params[offset + 2] := params[offset + 2] * Angle;  {Must be a real number, even if not used in this Atom}
        if kind = SectionTrunk then
          overlap := angle;{by convention}
        if Kind = SegmentTrunk then
          begin
            if GradientFactor > 1000 then
              sysbeep(1);
            params[2] := params[2] + GradientFactor * angle;
            Params[1] := Params[1] + GradientFactor * angle;
            DrawSeg(x, ySeg, params[2], params[1]);
{Draw the oval in the right place. 2 = Width , 1 = Height }
            oldY := ySeg;  {Save for next segment}
            x := x + round(params[2] / 2.0);{joint starts at the side of the segment}
            y := ySeg + round(params[1] / 2.0);
{joint starts half way down the segment }
          end;
        if Kind = Joint then
          begin
            {both next joint (NextLikeMe) and claw (FirstBelowMe) want x,y at end of this joint}
            oldX := x;
            oldY := y;
            ang := params[6];
            x := round(x + jointscale * params[5] * cos(ang));       {line end point    width*sine(angle)}
            y := round(y + jointscale * params[5] * sin(ang));       {line end point}
            thick := 1 + trunc(params[4]);      {1 is minimum thickness}
            DrawLine(oldX, oldY, x, y, thick);      {right side leg}
            leftX := xCenter - (x - xCenter);       {do the left side leg}
            leftOldX := xCenter - (oldX - xCenter);
            DrawLine(leftOldX, oldY, leftX, y, thick);
            foreColor(blackColor);
          end;
        if kind = Claw then
          DrawClaw(which, params, x, y, xCenter)       {all work is done in here}
        else
 {TED:  why else?  Presumably because claw is the end of the line?}
          begin
            if FirstBelowMe <> 0 then
              Draw(FirstBelowMe, params, x, y, xCenter, ySeg);       {build on my cumulative numbers}
            if Kind = SegmentTrunk then
              begin
                x := xCenter;
                ySeg := round(oldY + overlap * params[1]);{Seg}
        {Jump down by height of this segment to the next segment}
              end;
            if NextLikeMe <> 0 then
              begin
                if (Kind = AnimalJoint) or (Kind = SectionJoint) or (Kind = SegmentJoint) then
                  Draw(NextLikeMe, params, x, y, xCenter, ySeg)       {build on me}
                else if kind <> AnimalTrunk then
                  Draw(NextLikeMe, myPars, x, y, xCenter, ySeg);     {build on my parent's numbers}
              end;
                {Note that each Joint builds on the length of the SegJoint, not the joint just before it.}
                {This is consistant with the way Sections and Segments work.}
          end;
      end;
  end; {Draw}


  procedure DrawAnimal (BoxNo, x, y: integer);
    {An example of how to call Draw for an animal}
    var
      params: CumParams;
      ii, j, ySeg: integer;
  begin
    for ii := 1 to 9 do
      params[ii] := 1.0;      {clear it all out}
    ySeg := y;
    Draw(BreedersChoice[BoxNo], params, x, y, x, ySeg);
        {x = xCenter when we start}
  end;

  procedure DrawInBox (BoxNo: integer);
    var
      where: rect;
      centre, start, boxwidth, boxheight: integer;
  begin
    where := Box[BoxNo];
    boxwidth := where.right - where.left;
    boxheight := where.bottom - where.top;
    if sideways then
      begin
        centre := where.top + boxheight div 2;
        start := where.left + boxwidth div 2;
        WestPole := Prect.right;
        EastPole := Prect.left;
        if centring or (BoxNo = MidBox) then
          begin
            hidePen;
            DrawAnimal(BoxNo, centre, start); {return with NorthPole and SouthPole updated}
            ShowPen;
            Midriff := WestPole + (EastPole - WestPole) div 2;
            verticalOffset := Start - Midriff;
          end;
      end
    else
      begin
        start := where.top + boxheight div 2;
        centre := where.left + boxwidth div 2;
        NorthPole := Prect.bottom;
        SouthPole := Prect.top;
        if centring or (BoxNo = MidBox) then
          begin
            hidePen; {Preliminary dummy draw to measure North & South extent of animal}
            DrawAnimal(BoxNo, centre, start); {return with NorthPole and SouthPole updated}
            ShowPen;
            Midriff := NorthPole + (SouthPole - NorthPole) div 2;
            verticalOffset := Start - Midriff;
          end;
      end;
    if AnimalPicture[BoxNo] <> nil then
      KillPicture(AnimalPicture[BoxNo]); {redraw Picture in correct position}
    AnimalPicture[BoxNo] := OpenPicture(Box[BoxNo]);
    showpen;
    ClipRect(Box[BoxNo]);
    DrawAnimal(BoxNo, centre, start + VerticalOffset);
{Midriff := NorthPole - VerticalOffset + (SouthPole - NorthPole) div 2;}
{VerticalOffset := Start - Midriff;}
    hidepen;
    ClipRect(Prect);
    ClosePicture;
  end; {DrawInBox}

  procedure Clear (box: rect);
    var
      r: rect;
  begin
    with box do
      begin
        r.top := top + 1;
        r.bottom := bottom - 1;
        r.left := left + 1;
        r.right := right - 1;
      end;
    eraserect(r);
  end;{clear }

  procedure evolve (MLoc: point);
    var
      j, Margcentre: INTEGER;
      BoxesChanged: BOOLEAN;
      SlideRect: rect;

    procedure GrowChild (j: INTEGER);
      var
        k: LONGINT;
    begin
      Cliprect(Prect);
      PenMode(PatXor);
      MoveTo(Centre[Midbox].h, Centre[Midbox].v);
      LineTo(Centre[j].h, Centre[j].v);
      k := TickCount;
      repeat
      until TickCount >= k + 2;
      MoveTo(Centre[Midbox].h, Centre[Midbox].v);
      LineTo(Centre[j].h, Centre[j].v);
      PenMode(PatCopy);
      if (BoneYard[BreedersChoice[j]]^^.kind <> AnimalTrunk) then
        TellError('Breeders Choise is not an animal');
      if j <> MidBox then
        kill(BreedersChoice[j]);
      BreedersChoice[j] := reproduce(BreedersChoice[MidBox]);
      SegmentCounter := 0;
      CountSeg(BreedersChoice[j]);
{ClipRect(Box[j]);}
{if not AbortFlag then}
      DrawInBox(j);
    end;

  begin
    j := 0;
    special := 0;
    repeat
      j := j + 1;
      if (PtInRect(Mloc, box[j])) then
        special := j;
    until (j = NBoxes);
    if special > 0 then
      begin
        ObscureCursor;
        for j := 1 to NBoxes do
          if j <> special then
            if not resizing then
              EraseRect(box[j]);
        PenPat(white);
        Framerect(box[special]);
        PenPat(black);
        Slide(box[special], box[MidBox]);
        if special <> MidBox then
          begin
            kill(BreedersChoice[MidBox]);
            BreedersChoice[MidBox] := Allocate;
          end;
        BreedersChoice[MidBox] := Copy(BreedersChoice[special]);
        if not resizing then
          SetUpBoxes;
        ClipRect(Box[MidBox]);
        DrawInBox(MidBox);
        for j := 1 to MidBox - 1 do
          Growchild(j);
        for j := MidBox + 1 to NBoxes do
          Growchild(j);
        ClipRect(Prect);
        special := MidBox;
      end;
  end; {evolve}

  procedure UpDateAnimals;
    var
      j, offset: integer;
      frameBox: rect;
  begin
```

```
    if resizing then
      begin
        setupboxes;
        evolve(centre[MidBox]);
        resizing := false;
      end
    else
      begin
        if startingUp then
          SetUpBoxes
        else
          Drawboxes;
        startingUp := false;
        for j := 1 to NRows * NCols do
          DrawPicture(AnimalPicture[j],  box[j]);
      end;
  end; {UpDateAnimal}


  function NewAtom: integer;
    {Create a new atom with generic values in it}
    {NewAtom has 1.0 in factors and 0 in pointers as a nice default}
    var
      new: integer;
  begin
    new := Allocate;
    with BoneYard[new]^^ do
      begin
        Height := 1.0;
        Width := 1.0;
        Angle := 1.0;
        NextLikeMe := 0;
        FirstBelowMe := 0;
      end;
    NewAtom := new;
  end;
{I still vote for AnimalJoint . Width = 20 and AnimalJoint . Angle = 0.25 in the default animal .}

  function MinimalAnimal: integer;
    var
      aa, bb: integer;
  begin
    aa := NewAtom;
    BoneYard[aa]^^.Kind := Claw;

    bb := NewAtom;
    BoneYard[bb]^^.Kind := Joint;
    BoneYard[bb]^^.width := 5;
    BoneYard[bb]^^.angle := 2;
    BoneYard[bb]^^.FirstBelowMe := aa;

    aa := NewAtom;
    BoneYard[aa]^^.Kind := SegmentClaw;
    BoneYard[aa]^^.FirstBelowMe := bb;

    bb := NewAtom;
    BoneYard[bb]^^.Kind := SegmentJoint;
    BoneYard[bb]^^.NextLikeMe := aa;
    BoneYard[bb]^^.angle := 2;

    aa := NewAtom;
    BoneYard[aa]^^.Kind := SegmentTrunk;
    BoneYard[aa]^^.FirstBelowMe := bb;
    bb := NewAtom;
    BoneYard[bb]^^.Kind := SectionClaw;
    BoneYard[bb]^^.FirstBelowMe := aa;

    aa := NewAtom;
    BoneYard[aa]^^.Kind := SectionJoint;
    BoneYard[aa]^^.NextLikeMe := bb;

    bb := NewAtom;
    BoneYard[bb]^^.Kind := SectionTrunk;
    BoneYard[bb]^^.Angle := 0.5; {Segment overlap, by convention}
    BoneYard[bb]^^.FirstBelowMe := aa;

    aa := NewAtom;
    BoneYard[aa]^^.Kind := AnimalClaw;
    BoneYard[aa]^^.FirstBelowMe := bb;

    bb := NewAtom;
    BoneYard[bb]^^.Kind := AnimalJoint;
    BoneYard[bb]^^.NextLikeMe := aa;
    BoneYard[bb]^^.Width := 5;      {make it visible}
    BoneYard[bb]^^.angle := 5;

    aa := NewAtom;
    BoneYard[aa]^^.Kind := AnimalTrunk;
    BoneYard[aa]^^.FirstBelowMe := bb;
    BoneYard[aa]^^.NextLikeMe := -2; {Gradient, by convention}
    BoneYard[aa]^^.Angle := CountAtoms(aa);
    BoneYard[aa]^^.Height := 20;
    BoneYard[aa]^^.Width := 20;

    MinimalAnimal := aa;
  end;

  procedure FirstGeneration;
    var
      ii: integer;
  begin
    for ii := 1 to MidBox - 1 do
      begin
        BreedersChoice[ii] := Reproduce(BreedersChoice[MidBox]);
      end;
    for ii := MidBox + 1 to NRows * NCols do
      begin
        BreedersChoice[ii] := Reproduce(BreedersChoice[MidBox]);
      end;
{PenNormal;}
    Evolve(Centre[MidBox]);
  end; {FirstGeneration}

  procedure Breed;
    var
      ii: integer;
      NeedAnimal: Boolean;
  begin
    NeedAnimal := false;
    ii := BreedersChoice[MidBox];
    if (ii < 1) or (ii > YardSize) then
      NeedAnimal := true
    else if Boneyard[BreedersChoice[MidBox]]^^.kind = free then
      NeedAnimal := true;
    if needAnimal then
```

```
      begin
        BreedersChoice[MidBox] := Allocate;
        BreedersChoice[MidBox] := MinimalAnimal;
        FirstGeneration;
        BreedersChoice[MidBox] := MinimalAnimal;
      end; {else the Open Breed_Window in HandleMenus is sufficient to replace the old Arthromorphs}
  end;{Breed}

  procedure NewMinimal;
  begin
    BreedersChoice[MidBox] := 0; {Force Breed to recreate new MinimalAnimal}
    Breed
  end;

  procedure flipWantColor;
  begin
    wantColor := not wantColor;
    DrawinBox(MidBox);
  end; {flipWantColor}

  function Extract (which: integer): integer;
    {Copy this animal from the BoneYard to the MiniYard.}
    {Return index of copy in MiniYard}
{Afterwards: Since Animal is compact in the front part of MiniYard, just copy atoms}
{   from 1 to MiniFree-1 into the file}
    var
      newPlace, ii: integer;
  begin
    if BoneYard[which]^^.Kind = AnimalTrunk then
      begin  {Once at the start of the copy.  Erase the MiniYard}
        MiniFree := 1;
        for ii := 1 to miniSize do
          begin
            MiniYard[ii]^^.Kind := Free;
          end;
      end;

    {Duplicate this entire animal in the other yard.   }
    {Return the index of the start of the new animal.}
    newPlace := miniFree;       {Grab a new atom}
    miniFree := miniFree + 1;       {our Allocate since all are free}
    MiniYard[newPlace]^^ := BoneYard[which]^^;
    if BoneYard[which]^^.FirstBelowMe <> 0 then
      MiniYard[newPlace]^^.FirstBelowMe := Extract(BoneYard[which]^^.FirstBelowMe);
    if (BoneYard[which]^^.NextLikeMe <> 0) and (BoneYard[which]^^.kind <> AnimalTrunk) then
      MiniYard[newPlace]^^.NextLikeMe := Extract(BoneYard[which]^^.NextLikeMe);
    Extract := newPlace;            {Return the index of the new one}
  end;
{Example of use:-}
{Extract(BreedersChoice[ii]);  }
    {Copy this animal out to the MiniYard}
    {Now write MiniYard from 1 to MiniFree-1 out into a file}

  function Deposit (which: integer): integer;
    {Caller must copy Animal from a file directly into the MiniYard, then call Deposit(1)}
    {Here we copy the animal from the MiniYard into the BoneYard.}
    {Return the index of the start of the new animal in the BoneYard.}
    var
      newPlace: integer;
  begin
    newPlace := Allocate;       {Grab a new atom in the BoneYard}
    BoneYard[NewPlace]^^ := MiniYard[which]^^;
    if MiniYard[which]^^.FirstBelowMe <> 0 then
```

```
        BoneYard[NewPlace]^^.FirstBelowMe  :=  Deposit(MiniYard[which]^^.FirstBelowMe);
      if (MiniYard[which]^^.NextLikeMe <> 0) and (BoneYard[NewPlace]^^.kind <> AnimalTrunk) then
        BoneYard[NewPlace]^^.NextLikeMe  :=  Deposit(MiniYard[which]^^.NextLikeMe);
      Deposit  :=  newPlace;          {Return the index of the new one}
    end;
  {Example of use:-}
      {Read file into the MiniYard, then call this to move it to the BoneYard}
  {BreedersChoice[ii]  :=  Deposit(1);}
          {Install the animal in MiniYard in the BoneYard and return its start}

    procedure  SaveArthromorph;
      var
        where:  point;
        theReply: SFReply;
        theRefNum:  integer;
        Error:  OSErr;
        i:  integer;
    begin
      with where do
        begin
          h  :=  100;
          v  :=  100;
        end;
      i  :=  extract(BreedersChoice[MidBox]);
      SFPutFile(where, 'Save this Arthromorph as:', '', nil, theReply);
      if theReply.good then
        begin {not cancel}
          Error  :=  SetVol(nil,  theReply.vRefNum);
          if Error = NoErr then
            ReWrite(f,  theReply.fName);
          for i := 1 to MiniFree - 1 do
            write(f,   MiniYard[i]^^);
          Close(f);
        end; {not Cancel}
    end;  {SaveArthromorph}

    function  MyFilter  (param:  ParmBlkPtr):  BOOLEAN;
      var
        Wanted: BOOLEAN;
    begin
      Wanted  :=  (param^.ioFlFndrInfo.fdCreator  =  'JOHN')  and  (param^.ioFlFndrInfo.fdType  =  'DATA');
      MyFilter  :=  not  wanted;
    end;


    procedure  LoadArthromorph;
      var
        where:  point;
        theReply:  SFReply;
        theTypeList:  SFTypeList;
        theRefNum:  integer;
        Error:  OSErr;
        i:  integer;
        a:  atom;
        Exists:  Boolean;

    begin
      with where do
        begin
          h  :=  100;
          v  :=  100;
        end;
      theTypeList[0]  :=  'DATA';
      SFGetFile(where, 'Load which Arthromorph?', @MyFilter, -1, theTypeList, nil, theReply);
      if theReply.good then {else Cancel }
        begin
          i := BreedersChoice[MidBox];
          Exists := (i > 0) and (i < YardSize);
          if Exists then
            Kill(i);
          Error := SetVol(nil, theReply.vRefNum);
          if Error = NoErr then
            ReSet(f, theReply.fname);
          i := 0;
          while (i <= MiniSize) and (not eof(f)) do
            begin
              i := i + 1;
              read(f, MiniYard[i]^^);
            end;
          Close(f);
          BreedersChoice[MidBox] := Deposit(1);
          FirstGeneration;
          ValidRect(Prect);
        end; {not Cancel}
    end; {LoadArthromorph}

    procedure StartDocument;
      var
        j, i, NB, vRefNum: INTEGER;
        theFile: AppFile;
        ErrorCode: OSErr;
    begin
      j := 0;
      GetAppFiles(1, theFile);
      with theFile do
        if fType = 'APPL' then
          SysBeep(1)
        else
          begin
            ErrorCode := SetVol(nil, vRefNum);
            if ErrorCode <> noErr then
              SysBeep(1)
            else
              begin
                Reset(f, fName);
                i := 0;
                while (i <= MiniSize) and (not eof(f)) do
                  begin
                    i := i + 1;
                    read(f, MiniYard[i]^^);
                  end;
                Close(f);
                BreedersChoice[MidBox] := Deposit(1);
                FirstGeneration;
                ValidRect(Prect);
              end
          end;
    end; {StartDocument}

    procedure QuitGracefully;
      var
        j: integer;
    begin
      for j := 1 to YardSize do
        DisposHandle(Handle(BoneYard[j]));
      for j := 1 to MiniSize do
```

```
      DisposHandle(Handle(MiniYard[j]));
    for j := 1 to NRows * NCols do
      KillPicture(AnimalPicture[j]);
  end; {QuitGracefully}
end.
```

```
unit Richard;
interface
  uses
    MyGlobals, Ted;
  procedure MakeAllBodyMutations (State: boolean);
  procedure MakeAllAtomMutations (State: boolean);
  procedure PrintMiddle;

implementation

  procedure MakeAllBodyMutations (State: boolean);
  begin
    TrunkMut := State;
    LegsMut := State;
    ClawsMut := State;
    AnimalTrunkMut := State;
    AnimalLegsMut := State;
    AnimalClawsMut := State;
    SectionTrunkMut := State;
    SectionLegsMut := State;
    SectionClawsMut := State;
    SegmentTrunkMut := State;
    SegmentLegsMut := State;
    SegmentClawsMut := State;
  end;

  procedure MakeAllAtomMutations (State: boolean);
  begin
    WidthMut := State;
    HeightMut := State;
    AngleMut := State;
    DuplicationMut := State;
    DeletionMut := State;
  end;

  procedure PrintAt (this: Atom);
  begin
    with this do
      begin
        write(Height : 10 : 2, Width : 10 : 2, Angle : 10 : 2, '        ');
        case kind of
          AnimalTrunk:
            write('AnimalTrunk');
          AnimalJoint:
            write('        AnimalJoint');
          AnimalClaw:
            write('        AnimalClaw');
          SectionTrunk:
            write('            SectionTrunk');
          SectionJoint:
            write('                SectionJoint');
          SectionClaw:
            write('                SectionClaw');
          SegmentTrunk:
            begin
              SegmentCounter := SegmentCounter + 1;
              write('                        SegmentTrunk', SegmentCounter);
            end;
          SegmentJoint:
            write('                        SegmentJoint');
          SegmentClaw:
            write('                        SegmentClaw');
          Joint:
            write('                    Joint');
          Claw:
            write('                    Claw');
        end; {cases}
        writeln;
      end
  end; {PrintAt}

  procedure Print (which: integer);
   {Print this animal}
   {Recursively step through the animal}
    var
      this: Atom;
  begin
    this := BoneYard[which]^^;
    with this do
      begin
        if kind <> free then
          PrintAt(this);
        if FirstBelowMe <> 0 then
          Print(FirstBelowMe);
        if (NextLikeMe <> 0) and (kind <> AnimalTrunk) then
          Print(NextLikeMe);
      end
  end;

  procedure PrintMiddle;
    var
      sub: integer;
      r: rect;
  begin
    r := Prect;
    r.top := 60;
    SetTextRect(r);
    showtext;
    rewrite(output);
    writeln('Height ' : 10, 'Width' : 10, 'Angle' : 10);
    sub := BreedersChoice[MidBox];
    SegmentCounter := 0;
    if sub > 0 then
      if BoneYard[sub]^^.kind = AnimalTrunk then
        Print(BreedersChoice[MidBox]);
  end;

end.
```

```
unit InitTheMenus;

{File name:  InitTheMenus.Pas}
{Function: Pull in menu lists from a resource file.}
{          This procedure is called once at program start.}
{         AppleMenu is the handle to the Apple menu, it is also}
{         used in the procedure that handles menu events.}
{History: 12/12/90 Original by Prototyper.    }
{                }
interface

  procedure Init_My_Menus;           {Initialize the menus}

  var
    AppleMenu: MenuHandle;          {Menu handle}
    M_File: MenuHandle;             {Menu handle}
    M_Edit: MenuHandle;             {Menu handle}
    M_Operation: MenuHandle;        {Menu handle}
    M_View: MenuHandle;             {Menu handle}

implementation

  procedure Init_My_Menus;           {Initialize the menus}
    const
      Menu1 = 1001;                 {Menu resource ID}
      Menu2 = 1002;                 {Menu resource ID}
      Menu3 = 1003;                 {Menu resource ID}
      Menu4 = 1004;                 {Menu resource ID}
      Menu5 = 1005;                 {Menu resource ID}

  begin                             {Start of Init_My_Menus}
    ClearMenuBar;                 {Clear any old menu bars}

        { This menu is the APPLE menu, used for About and desk accessories.}
    AppleMenu := GetMenu(Menu1);{Get the menu from the resource file}
    InsertMenu(AppleMenu, 0);       {Insert this menu into the menu bar}
    AddResMenu(AppleMenu, 'DRVR');{Add in DAs}

    M_File := GetMenu(Menu2);       {Get the menu from the resource file}
    InsertMenu(M_File, 0);        {Insert this menu into the menu bar}

    M_Edit := GetMenu(Menu3);       {Get the menu from the resource file}
    InsertMenu(M_Edit, 0);          {Insert this menu into the menu bar}

    M_Operation := GetMenu(Menu4);{Get the menu from the resource file}
    InsertMenu(M_Operation, 0);{Insert this menu into the menu bar}

    M_View := GetMenu(Menu5);       {Get the menu from the resource file}
    InsertMenu(M_View, 0);          {Insert this menu into the menu bar}

    DrawMenuBar;                  {Draw the menu bar}

  end;                              {End of procedure Init_My_Menus}

end.                                {End of this unit}
```

```
unit Engineering_Window;

{File name:  Engineering_Window.Pas  }
{Function: Handle a dialog}
{History: 1/4/91 Original by Prototyper.    }
{                   }


interface

  uses
    MyGlobals, Ted, Richard, Error_Alert;

  procedure D_Engineering_Window;

implementation

  const                               {These are the item numbers for controls in the Dialog}
    I_OK = 1;
    I_All = 2;
    I_None = 3;
    I_All4 = 4;
    I_None6 = 5;
    I_Cancel = 6;
    I_Animal_Trunk = 7;
    I_Animal_Legs = 8;
    I_Animal_Claws = 9;
    I_Section_Trunk = 10;
    I_Section_Legs = 11;
    I_Section_Claws = 12;
    I_Segment_Trunk = 13;
    I_Segment_Legs = 14;
    I_Segment_Claws = 15;
    I_Length = 16;
    I_Height = 17;
    I_Angle = 18;
    I_Duplication = 19;
    I_Deletion = 20;
    I_Legs = 21;
    I_Claws = 22;
    I = 23;
    I_0 = 24;
    I27 = 25;
    I_Focus_on_1st_seg = 26;
    I_Focus_on_last_seg = 27;
    I_No_focus = 28;
    I_x = 29;
    I_x33 = 30;
    I_Rectangle1 = 31;
    I_Rectangle2 = 32;
    I_Rectangle4 = 33;
    I_Rectangle138 = 34;
  var
    ExitDialog, Accept: boolean;          {Flag used to exit the Dialog}
    DoubleClick: boolean;          {Flag to say that a double click on a list happened}
    MyPt: Point;                   {Current list selection point}
    MyErr: OSErr;                  {OS error returned}
    DearthOfAtomMuts, DearthOfBodyMuts, AnimalOrClawsOnly, DupDeleteOnly: boolean;

  procedure D_Engineering_Window;
    var
      GetSelection: DialogPtr;{Pointer to this dialog}
      tempRect: Rect;         {Temporary rectangle}
      DType: Integer;          {Type of dialog item}
      Index: Integer;          {For looping}
      DItem: Handle;           {Handle to the dialog item}
      CItem, CTempItem: controlhandle;{Control handle}
      sTemp: Str255;               {Get text entered, temp holding}
      itemHit: Integer;        {Get selection}
      temp: Integer;           {Get selection, temp holding}
      dataBounds: Rect;        {Rect to setup the list}
      cSize: Point;            {Pointer to a cell in a list}
      Icon_Handle: Handle;     {Temp handle to read an Icon into}
      NewMouse: Point;             {Mouse location during tracking Icon presses}
      InIcon: boolean;         {Flag to say pressed in an Icon}
      ThisEditText: TEHandle; {Handle to get the Dialogs TE record}
      TheDialogPtr: DialogPeek;{Pointer to Dialogs definition record, contains the TE record}

    {This is an update routine for non-controls in the dialog}
    {This is executed after the dialog is uncovered by an alert}
    procedure Refresh_Dialog;          {Refresh the dialogs non-controls}
      var
        rTempRect: Rect;           {Temp rectangle used for drawing}

    begin
      SetPort(GetSelection);       {Point to our dialog window}
      rTempRect := tempRect;   {Save the current contents of  tempRect}
      GetDItem(GetSelection, I_OK, DType, DItem, tempRect);{Get the item handle}
      PenSize(3, 3);               {Change pen to draw thick default outline}
      InsetRect(tempRect, -4, -4);{Draw outside the button by 1 pixel}
      FrameRoundRect(tempRect, 16, 16); {Draw the outline}
      PenSize(1, 1);               {Restore the pen size to the default value}

          {Draw a rectangle, Rectangle1  }
      SetRect(TempRect, 18, 35, 170, 286);{left,top,right,bottom}
      FrameRect(TempRect);         {Frame this rectangle area}

          {Draw a rectangle, Rectangle2  }
      SetRect(TempRect, 191, 36, 326, 196);{left,top,right,bottom}
      FrameRect(TempRect);         {Frame this rectangle area}

          {Draw a rectangle, Rectangle4  }
      SetRect(TempRect, 192, 215, 327, 273);{left,top,right,bottom}
      FrameRect(TempRect);         {Frame this rectangle area}

          {Draw a rectangle, Rectangle1  }
      SetRect(TempRect, 16, 292, 170, 361);{left,top,right,bottom}
      FrameRect(TempRect);         {Frame this rectangle area}

      tempRect := rTempRect;   {Restore the current contents of  tempRect}
    end;


    procedure AdjustCheckBoxes;
    begin
          {Setup initial conditions}
      GetDItem(GetSelection, I_Animal_Trunk, DType, DItem, tempRect);{Get the item handle}
      CItem := Pointer(DItem);{Change dialog handle to control handle}
      SetCtlValue(CItem, integer(AnimalTrunkMut));       {Check the checkbox}

      GetDItem(GetSelection, I_Animal_Legs, DType, DItem, tempRect);{Get the item handle}
      CItem := Pointer(DItem);{Change dialog handle to control handle}
      SetCtlValue(CItem, integer(AnimalLegsMut));    {Check the checkbox}

      GetDItem(GetSelection, I_Animal_Claws, DType, DItem, tempRect);{Get the item handle}
      CItem := Pointer(DItem);{Change dialog handle to control handle}
```

```
        SetCtlValue(CItem, integer(AnimalClawsMut));        {Check the checkbox}

        GetDItem(GetSelection, I_Section_Trunk, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(SectionTrunkMut));       {Check the checkbox}

        GetDItem(GetSelection, I_Section_Legs, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(SectionLegsMut));    {Check the checkbox}

        GetDItem(GetSelection, I_Section_Claws, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(SectionClawsMut));       {Check the checkbox}

        GetDItem(GetSelection, I_Segment_Trunk, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(SegmentTrunkMut));      {Check the checkbox}

        GetDItem(GetSelection, I_Segment_Legs, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(SegmentLegsMut));        {Check the checkbox}

        GetDItem(GetSelection, I_Segment_Claws, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(SegmentClawsMut));      {Check the checkbox}

        GetDItem(GetSelection, I_Legs, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(LegsMut));    {Check the checkbox}

        GetDItem(GetSelection, I_Claws, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(ClawsMut));        {Check the checkbox}

        GetDItem(GetSelection, I_Length, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(WidthMut));        {Check the checkbox}

        GetDItem(GetSelection, I_Height, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(HeightMut));       {Check the checkbox}

        GetDItem(GetSelection, I_Angle, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(AngleMut));        {Check the checkbox}

        GetDItem(GetSelection, I_Duplication, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(DuplicationMut));      {Check the checkbox}

        GetDItem(GetSelection, I_Deletion, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(DeletionMut));    {Check the checkbox}

    {And  now the radio buttons}

        GetDItem(GetSelection, 23, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(MutationPressure = positive));

        GetDItem(GetSelection, 24, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(MutationPressure = zero));
        GetDItem(GetSelection, 25, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(MutationPressure = negative));

        GetDItem(GetSelection, 26, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(FocusOfAttention = FirstSegmentOnly));

        GetDItem(GetSelection, 27, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(FocusOfAttention = LastSegmentOnly));

        GetDItem(GetSelection, 28, DType, DItem, tempRect);{Get the item handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        SetCtlValue(CItem, integer(FocusOfAttention = AnySegment));

      end; {AdjustCheckBoxes}

    begin                          {Start of dialog handler}
      GetSelection := GetNewDialog(4, nil, Pointer(-1));{Bring in the dialog resource}
      ShowWindow(GetSelection);{Open a dialog box}
      SelectWindow(GetSelection);{Lets see it}
      SetPort(GetSelection);    {Prepare to add conditional text}

      TheDialogPtr := DialogPeek(GetSelection);{Get to the inner record}
      ThisEditText := TheDialogPtr^.textH;{Get to the TE record}
      HLock(Handle(ThisEditText));{Lock it for safety}
      ThisEditText^^.txSize := 12;{TE Point size}
      TextSize(12);              {Window Point size}
      ThisEditText^^.txFont := systemFont;{TE Font ID}
      TextFont(systemFont);          {Window Font ID}
      ThisEditText^^.txFont := 0;{TE Font ID}
      ThisEditText^^.fontAscent := 12;{Font ascent}
      ThisEditText^^.lineHeight := 12 + 3 + 1;{Font ascent + descent + leading}
      HUnLock(Handle(ThisEditText));{UnLock the handle when done}

      AdjustCheckBoxes;

      Refresh_Dialog;            {Draw any Lists, popups, lines, or rectangles}

      ExitDialog := FALSE;           {Do not exit dialog handle loop yet}

      repeat                     {Start of dialog handle loop}
        ModalDialog(nil, itemHit);{Wait until an item is hit}
        GetDItem(GetSelection, itemHit, DType, DItem, tempRect);{Get item information}
        CItem := Pointer(DItem);{Get the control handle}

                  {Handle it real time}
        if (ItemHit = I_OK) then{Handle the Button being pressed}
          begin
            Accept := true;
                    {?? Code to handle this button goes here}
            ExitDialog := TRUE;{Exit the dialog when this selection is made}
            Refresh_Dialog;
          end;                   {End for this item selected}

        if (ItemHit = I_All) then{Handle the Button being pressed}
          begin
            MakeAllBodyMutations(true);
            AdjustCheckBoxes;
                    {?? Code to handle this button goes here}
            Refresh_Dialog;
          end;                   {End for this item selected}

        if (ItemHit = I_None) then{Handle the Button being pressed}
          begin
            MakeAllBodyMutations(false);
            AdjustCheckBoxes;
                    {?? Code to handle this button goes here}
            Refresh_Dialog;
          end;                   {End for this item selected}

        if (ItemHit = I_All4) then{Handle the Button being pressed}
          begin
            MakeAllAtomMutations(true);
            AdjustCheckBoxes;
                    {?? Code to handle this button goes here}
            Refresh_Dialog;
          end;                   {End for this item selected}

        if (ItemHit = I_None6) then{Handle the Button being pressed}
          begin
            MakeAllAtomMutations(false);
            AdjustCheckBoxes;
                    {?? Code to handle this button goes here}
            Refresh_Dialog;
          end;                   {End for this item selected}

        if (ItemHit = I_Cancel) then{Handle the Button being pressed}
          begin
            Accept := false;
                    {?? Code to handle this button goes here}
            ExitDialog := TRUE;{Exit the dialog when this selection is made}
            Refresh_Dialog;
          end;                   {End for this item selected}


        if (ItemHit = I_Animal_Trunk) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            AnimalTrunkMut := not boolean(temp);
          end;                   {End for this item selected}


        if (ItemHit = I_Animal_Legs) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            AnimalLegsMut := not boolean(temp);
          end;                   {End for this item selected}


        if (ItemHit = I_Animal_Claws) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            AnimalClawsMut := not boolean(temp);             {End for this item checked}
          end;                   {End for this item selected}


        if (ItemHit = I_Section_Trunk) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            SectionTrunkMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Section_Legs) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            SectionLegsMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Section_Claws) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            SectionClawsMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Segment_Trunk) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            SegmentTrunkMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Segment_Legs) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            SegmentLegsMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Segment_Claws) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            SegmentClawsMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Length) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            WidthMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Height) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
            SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
            HeightMut := not boolean(temp);
          end;                  {End for this item selected}


        if (ItemHit = I_Angle) then{Handle the checkbox being pressed}
          begin
            temp := GetCtlValue(CItem);{Get the current Checkbox value}
```

```
          SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
          AngleMut := not boolean(temp);
        end;                      {End for this item selected}


      if (ItemHit = I_Duplication) then{Handle the checkbox being pressed}
        begin
          temp := GetCtlValue(CItem);{Get the current Checkbox value}
          SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
          DuplicationMut := not boolean(temp);
        end;                      {End for this item selected}


      if (ItemHit = I_Deletion) then{Handle the checkbox being pressed}
        begin
          temp := GetCtlValue(CItem);{Get the current Checkbox value}
          SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
          DeletionMut := not boolean(temp);
        end;                      {End for this item selected}


      if (ItemHit = I_Legs) then{Handle the checkbox being pressed}
        begin
          temp := GetCtlValue(CItem);{Get the current Checkbox value}
          SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
          LegsMut := not boolean(temp);
        end;                      {End for this item selected}


      if (ItemHit = I_Claws) then{Handle the checkbox being pressed}
        begin
          temp := GetCtlValue(CItem);{Get the current Checkbox value}
          SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
          LegsMut := not boolean(temp);                 {End for this item checked}
        end;                      {End for this item selected}

      if (ItemHit >= I) and (ItemHit <= I27) then{Handle the Radio selection}
        begin
          for Index := I to I27 do{Clear all other radios}
            begin
              GetDItem(GetSelection, Index, DType, DItem, tempRect);{Get the Radio handle}
              CTempItem := Pointer(DItem);{Convert to a control handle}
              SetCtlValue(CTempItem, 0);{Turn the radio selection OFF}
            end;                  {End of clear the radio selections loop}
          SetCtlValue(CItem, 1);{Turn the one radio selection ON}
        end;                      {End for this item selected}

      if (ItemHit >= I_Focus_on_1st_seg) and (ItemHit <= I_No_focus) then{Handle the Radio selection}
        begin
          for Index := I_Focus_on_1st_seg to I_No_focus do{Clear all other radios}
            begin
              GetDItem(GetSelection, Index, DType, DItem, tempRect);{Get the Radio handle}
              CTempItem := Pointer(DItem);{Convert to a control handle}
              SetCtlValue(CTempItem, 0);{Turn the radio selection OFF}
            end;                  {End of clear the radio selections loop}
          SetCtlValue(CItem, 1);{Turn the one radio selection ON}
        end;                      {End for this item selected}


    until ExitDialog;             {Handle dialog items until exit selected}
  {Get results after dialog}
    if Accept then
      begin
        DearthOfAtomMuts := true;
        DearthOfBodyMuts := true;
        AnimalOrClawsOnly := true;
        DupDeleteOnly := true;

        GetDItem(GetSelection, I_Deletion, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        DeletionMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if DeletionMut then
          DearthOfAtomMuts := false;
            {??? HANDLE THE CHECKBOX RESULT FOR  Deletion HERE}

        GetDItem(GetSelection, I_Duplication, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        DuplicationMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if DuplicationMut then
          DearthOfAtomMuts := false;
            {??? HANDLE THE CHECKBOX RESULT FOR  Duplication HERE}

        GetDItem(GetSelection, I_Angle, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        AngleMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if AngleMut then
          begin
            DearthOfAtomMuts := false;
            DupDeleteOnly := false;
          end;
            {??? HANDLE THE CHECKBOX RESULT FOR  Angle HERE}

        GetDItem(GetSelection, I_Height, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        HeightMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if HeightMut then
          begin
            DearthOfAtomMuts := false;
            DupDeleteOnly := false;
          end;
            {??? HANDLE THE CHECKBOX RESULT FOR  Height HERE}

        GetDItem(GetSelection, I_Length, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        WidthMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if WidthMut then
          begin
            DearthOfAtomMuts := false;
            DupDeleteOnly := false;
          end;
            {??? HANDLE THE CHECKBOX RESULT FOR  Length HERE}

        GetDItem(GetSelection, I_Animal_Trunk, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        AnimalTrunkMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if AnimalTrunkMut then
          DearthOfBodyMuts := false;
            {??? HANDLE THE CHECKBOX RESULT FOR  Animal Trunk HERE}

        GetDItem(GetSelection, I_Animal_Legs, DType, DItem, tempRect);{Get the Checkbox handle}
        CItem := Pointer(DItem);{Change dialog handle to control handle}
        AnimalLegsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
        if AnimalLegsMut then
```

```
  DearthOfBodyMuts := false;
    {??? HANDLE THE CHECKBOX RESULT FOR  Animal Legs HERE}

GetDItem(GetSelection, I_Animal_Claws, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
AnimalClawsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if AnimalClawsMut then
  DearthOfBodyMuts := false;
    {??? HANDLE THE CHECKBOX RESULT FOR  Animal Claws HERE}

GetDItem(GetSelection, I_Section_Trunk, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
SectionTrunkMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if SectionTrunkMut then
  begin
    DearthOfBodyMuts := false;
    AnimalOrClawsOnly := false;
  end;
    {??? HANDLE THE CHECKBOX RESULT FOR  Section Trunk HERE}

GetDItem(GetSelection, I_Section_Legs, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
SectionLegsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if SectionLegsMut then
  begin
    DearthOfBodyMuts := false;
    AnimalOrClawsOnly := false;
  end;
    {??? HANDLE THE CHECKBOX RESULT FOR  Section Legs HERE}

GetDItem(GetSelection, I_Section_Claws, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
SectionClawsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if SectionClawsMut then
  DearthOfBodyMuts := false;
    {??? HANDLE THE CHECKBOX RESULT FOR  Section Claws HERE}

GetDItem(GetSelection, I_Segment_Trunk, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
SegmentTrunkMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if SegmentTrunkMut then
  begin
    DearthOfBodyMuts := false;
    AnimalOrClawsOnly := false;
  end;
    {??? HANDLE THE CHECKBOX RESULT FOR  Segment Trunk HERE}

GetDItem(GetSelection, I_Segment_Legs, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
SegmentLegsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if SegmentLegsMut then
  begin
    DearthOfBodyMuts := false;
    AnimalOrClawsOnly := false;
  end;
    {??? HANDLE THE CHECKBOX RESULT FOR  Segment Legs HERE}

GetDItem(GetSelection, I_Segment_Claws, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
SegmentClawsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if SegmentClawsMut then
  DearthOfBodyMuts := false;
    {??? HANDLE THE CHECKBOX RESULT FOR  Segment Claws HERE}
```

```
GetDItem(GetSelection, I_Legs, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
LegsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if LegsMut then
  begin
    DearthOfBodyMuts := false;
    AnimalOrClawsOnly := false;
  end;
    {??? HANDLE THE CHECKBOX RESULT FOR Legs HERE}

GetDItem(GetSelection, I_Claws, DType, DItem, tempRect);{Get the Checkbox handle}
CItem := Pointer(DItem);{Change dialog handle to control handle}
ClawsMut := boolean(GetCtlValue(CItem));{Get the checkbox value}
if ClawsMut then
  DearthOfBodyMuts := false;
    {??? HANDLE THE CHECKBOX RESULT FOR Claws HERE}

Index := I;                 {Start at the first radio in this group}
repeat                      {Look until we have found the radio selected}
  GetDItem(GetSelection, Index, DType, DItem, tempRect);{Get the radio handle}
  CItem := Pointer(DItem);{Change dialog handle to control handle}
  temp := GetCtlValue(CItem);{Get the radio value}
  Index := Index + 1;{Go to next radio}
until (temp <> 0) or (Index > I27);{Go till we find it}
temp := Index - I + 1;          {The indexed radio selection}
case temp of
  2:
    MutationPressure := positive;
  3:
    mutationPressure := zero;
  4:
    MutationPressure := negative;
end; {cases}
    {??? HANDLE THE RADIO RESULT FOR I TO I27 HERE}

Index := I_Focus_on_1st_seg;{Start at the first radio in this group}
repeat                      {Look until we have found the radio selected}
  GetDItem(GetSelection, Index, DType, DItem, tempRect);{Get the radio handle}
  CItem := Pointer(DItem);{Change dialog handle to control handle}
  temp := GetCtlValue(CItem);{Get the radio value}
  Index := Index + 1;{Go to next radio}
until (temp <> 0) or (Index > I_No_focus);{Go till we find it}
temp := Index - I_Focus_on_1st_seg + 1;{The indexed radio selection}
case temp of
  2:
    FocusOfAttention := FirstSegmentOnly;
  3:
    FocusOfAttention := LastSegmentOnly;
  4:
    FocusOfAttention := AnySegment;
end; {cases}
    {??? HANDLE THE RADIO RESULT FOR I_Focus_on_1st_seg TO I_No_focus HERE}

AgreeToExit := True;
if DearthOfAtomMuts then
  begin
    AgreeToExit := false;
    TellError('You must allow at least one class of mutation');
  end;
if DearthOfBodyMuts then
  begin
    AgreeToExit := false;
```

```
            TellError('You must allow at least one body part to mutate');
          end;
        if AnimalOrClawsOnly and DupDeleteOnly then
          begin
            AgreeToExit := false;
            TellError('You cannot duplicate or delete claws or whole animal');
          end;

      end {OK button pressed}
    else
      AgreeToExit := true; {Cancel button pressed}

    DisposDialog(GetSelection);{Flush the dialog out of memory}

  end;                        {End of procedure}

end.                          {End of unit}
```

```
unit Genome_Window;

{File name: Genome_Window.Pas}
{Function: Handle a Window}
{History: 12/12/90 Original by Prototyper.   }

interface



    {Initialize us so all our routines can be activated}
  procedure Init_Genome_Window;

    {Close our window}
  procedure Close_Genome_Window (whichWindow: WindowPtr; var theInput: TEHandle);

    {Open our window and draw everything}
  procedure Open_Genome_Window (var theInput: TEHandle);

    {Update our window, someone uncovered a part of us}
  procedure Update_Genome_Window (whichWindow: WindowPtr);

    {Handle action to our window, like controls}
  procedure Do_Genome_Window (myEvent: EventRecord; var theInput: TEHandle);

implementation

  var
    MyWindow: WindowPtr;            {Window pointer}
    tempRect, temp2Rect: Rect;      {Temporary rectangle}
    Index: Integer;             {For looping}
    CtrlHandle: ControlHandle;{Control handle}
    sTemp: Str255;                {Get text entered, temp holding}

{===============================}

    {Initialize us so all our routines can be activated}
  procedure Init_Genome_Window;

  begin                             {Start of Window initialize routine}
    MyWindow := nil;                {Make sure other routines know we are not valid yet}
  end;                              {End of procedure}

{===============================}

    {Close our window}
  procedure Close_Genome_Window;

  begin                             {Start of Window close routine}
    if (MyWindow <> nil) and ((MyWindow = whichWindow) or (ord4(whichWindow) = -1)) then{See if we should close this win
      begin
        DisposeWindow(MyWindow);{Clear window and controls}
        MyWindow := nil;          {Make sure other routines know we are open}
      end;                        {End for if (MyWindow<>nil)}
  end;                              {End of procedure}

{===============================}

    {Update our window, someone uncovered a part of us}
  procedure UpDate_Genome_Window;
    var
      SavePort: WindowPtr;          {Place to save the last port}

  begin                             {Start of Window update routine}
```

```
    if (MyWindow <> nil) and (MyWindow = whichWindow) then{Handle an open when already opened}
      begin
        GetPort(SavePort);          {Save the current port}
        SetPort(MyWindow);              {Set the port to my window}
        DrawControls(MyWindow);{Draw all the controls}
        SetPort(SavePort);          {Restore the old port}
      end;                          {End for if (MyWindow<>nil)}
  end;                              {End of procedure}

{================================}

    {Open our window and draw everything}
  procedure Open_Genome_Window;
    var
      Index: Integer;               {For looping}
      dataBounds: Rect;             {For making lists}
      cSize: Point;                 {For making lists}

  begin                                 {Start of Window open routine}

    if (MyWindow = nil) then            {Handle an open when already opened}
      begin
        MyWindow := GetNewWindow(1, nil, Pointer(-1));{Get the window from the resource file}
        SetPort(MyWindow);              {Prepare to write into our window}

        ShowWindow(MyWindow);           {Show the window now}
        SelectWindow(MyWindow);{Bring our window to the front}

      end                           {End for if (MyWindow<>nil)}
    else
      SelectWindow(MyWindow);{Already open, so show it}

  end;                                  {End of procedure}

{================================}

    {Handle action to our window, like controls}
  procedure Do_Genome_Window;
    var
      RefCon: longint;                  {RefCon for controls}
      code: integer;                    {Location of event in window or controls}
      theValue: integer;                {Current value of a control}
      whichWindow: WindowPtr;               {Window pointer where event happened}
      myPt: Point;                      {Point where event happened}
      theControl: ControlHandle;        {Handle for a control}
      MyErr: OSErr;                     {OS error returned}

  begin                                 {Start of Window handler}
    if (MyWindow <> nil) then           {Handle only when the window is valid}
      begin
        code := FindWindow(myEvent.where, whichWindow);{Get where in window and which window}

        if (myEvent.what = MouseDown) and (MyWindow = whichWindow) then{}
          begin                     {}
            myPt := myEvent.where;{Get mouse position}
            with MyWindow^.portBits.bounds do{Make it relative}
              begin
                myPt.h := myPt.h + left;
                myPt.v := myPt.v + top;
              end;

          end;
        if (MyWindow = whichWindow) and (code = inContent) then{for our window}
         begin

          code := FindControl(myPt, whichWindow, theControl);{Get type of control}

          if (code <> 0) then{Check type of control}
           code := TrackControl(theControl, myPt, nil);{Track the control}

         end;                  {End for if (MyWindow=whichWindow)}
       end;                    {End for if (MyWindow<>nil)}
     end;                        {End of procedure}

    {================================}


    end.                         {End of unit}
```

```
unit Breeding_Window;

{File name: Breeding_Window.Pas}
{Function: Handle a Window}
{History: 12/15/90 Original by Prototyper.    }

interface
 uses
  MyGlobals, Ted;

  {Initialize us so all our routines can be activated}
 procedure Init_Breeding_Window;

  {Close our window}
 procedure Close_Breeding_Window (whichWindow: WindowPtr; var theInput: TEHandle);

  {Open our window and draw everything}
 procedure Open_Breeding_Window (var theInput: TEHandle);

  {Update our window, someone uncovered a part of us}
 procedure Update_Breeding_Window (whichWindow: WindowPtr);

  {Handle action to our window, like controls}
 procedure Do_Breeding_Window (myEvent: EventRecord; var theInput: TEHandle);


  {Handle resizing scrollbars}
 procedure Resized_Breeding_Window (OldRect: Rect; whichWindow: WindowPtr);

implementation

 var
  MyWindow: WindowPtr;          {Window pointer}
  tempRect, temp2Rect: Rect;    {Temporary rectangle}
  Index: Integer;           {For looping}
  ScrollHHandle, ScrollVHandle: controlhandle;{Scrolling Control handles}
  CtrlHandle: ControlHandle;{Control handle}
  sTemp: Str255;              {Get text entered, temp holding}

{=================================}

  {Initialize us so all our routines can be activated}
 procedure Init_Breeding_Window;

 begin                          {Start of Window initialize routine}
  MyWindow := nil;              {Make sure other routines know we are not valid yet}
  ScrollHHandle := nil;         {Make sure other routines know we are not valid yet}
  ScrollVHandle := nil;         {Make sure other routines know we are not valid yet}
 end;                           {End of procedure}

{=================================}

  {Close our window}
 procedure Close_Breeding_Window;

 begin                          {Start of Window close routine}
  if (MyWindow <> nil) and ((MyWindow = whichWindow) or (ord4(whichWindow) = -1)) then{See if we should close this win
   begin
    DisposeWindow(MyWindow);{Clear window and controls}
    MyWindow := nil;        {Make sure other routines know we are open}
   end;                     {End for if (MyWindow<>nil)}
 end;                           {End of procedure}
```

```
    {================================}

      {We were resized or zoomed, update the scrolling scrollbars}
  procedure Resized_Breeding_Window;      {Resized this window}
    var
      SavePort: WindowPtr;          {Place to save the last port}
      temp2Rect: Rect;              {temp rectangle}
      Index: integer;             {temp integer}

  begin                             {Start of Window resize routine}
    if (MyWindow = whichWindow) then{Only do if the window is us}
      begin
        GetPort(SavePort);        {Save the current port}
        SetPort(MyWindow);            {Set the port to my window}

        if (ScrollHHandle <> nil) then{Only do if the control is valid}
          begin
            HLock(Handle(ScrollHHandle));{Lock the handle while we use it}
            tempRect := ScrollHHandle^^.contrlRect;{Get the last control position}
            tempRect.Top := tempRect.Top - 4;{Widen the area to update}
            tempRect.Right := tempRect.Right + 16;{Widen the area to update}
            InvalRect(tempRect);{Flag old position for update routine}
            tempRect := ScrollHHandle^^.contrlRect;{Get the last control position}
            temp2Rect := MyWindow^.PortRect;{Get the window rectangle}
            Index := temp2Rect.Right - temp2Rect.Left - 13;{Get the scroll area width}
            tempRect.Left := 0;  {Pin at left edge}
            HideControl(ScrollHHandle);{Hide it during size and move}
            SizeControl(ScrollHHandle, Index, 16);{Make it 16 pixels high, std width}
            MoveControl(ScrollHHandle, tempRect.Left - 1, temp2Rect.bottom - temp2Rect.top - 15);{Size it correctly}
            ShowControl(ScrollHHandle);{Safe to show it now}
            HUnLock(Handle(ScrollHHandle));{Let it float again}
          end;                    {End for scroll handle not nil)}

        if (ScrollVHandle <> nil) then{Only do if the control is valid}
          begin
            HLock(Handle(ScrollVHandle));{Lock the handle while we use it}
            tempRect := ScrollVHandle^^.contrlRect;{Get the last control position}
            tempRect.Left := tempRect.Left - 4;{Widen the area to update}
            tempRect.Bottom := tempRect.Bottom + 16;{Widen the area to update}
            InvalRect(tempRect);{Flag old position for update routine}
            tempRect := ScrollVHandle^^.contrlRect;{Get the last control position}
            temp2Rect := MyWindow^.PortRect;{Get the window rectangle}
            Index := temp2Rect.bottom - temp2Rect.top - 13;{Get the scroll area height}
            tempRect.Top := 0;      {Pin at top edge}
            HideControl(ScrollVHandle);{Hide it during size and move}
            SizeControl(ScrollVHandle, 16, Index);{Make it 16 pixels wide, std height}
            MoveControl(ScrollVHandle, temp2Rect.right - temp2Rect.Left - 15, tempRect.Top - 1);{Size it correctly}
            ShowControl(ScrollVHandle);{Safe to show it now}
            HUnLock(Handle(ScrollVHandle));{Let it float again}
          end;                    {End for scroll handle not nil)}

        SetPort(SavePort);        {Restore the old port}
      end;                        {End for window is us}
  end;                          {End of procedure}

        {================================}

          {Update our window, someone uncovered a part of us}
  procedure UpDate_Breeding_Window;
    var
      SavePort: WindowPtr;{Place to save the last port}

  begin                         {Start of Window update routine}
    if (MyWindow <> nil) and (MyWindow = whichWindow) then{Handle an open when already opened}
      begin
        GetPort(SavePort);{Save the current port}
        SetPort(MyWindow);{Set the port to my window}
        if resizing then
          begin
            cliprect(screenbits.bounds);
            EraseRect(myWindow^.portrect);
          end;
        SelectWindow(myWindow);
        DrawControls(MyWindow);{Draw all the controls}
        DrawGrowIcon(MyWindow);{Draw the Grow box}
        UpDateAnimals;
        SetPort(SavePort);{Restore the old port}
      end;                  {End for if (MyWindow<>nil)}
  end;                          {End of procedure}

        {=================================}

            {Open our window and draw everything}
  procedure Open_Breeding_Window;
    var
      Index: Integer;       {For looping}
      dataBounds: Rect;  {For making lists}
      cSize: Point;         {For making lists}

  begin                         {Start of Window open routine}

    if (MyWindow = nil) then{Handle an open when already opened}
      begin
        MyWindow := GetNewWindow(2, nil, Pointer(-1));{Get the window from the resource file}
        SetPort(MyWindow);{Prepare to write into our window}

        ShowWindow(MyWindow);{Show the window now}
        SelectWindow(MyWindow);{Bring our window to the front}

      end                   {End for if (MyWindow<>nil)}
    else
      SelectWindow(MyWindow);{Already open, so show it}
    BreedingWindow := MyWindow;
  end;                          {End of procedure}

        {=================================}

            {Handle action to our window, like controls}
  procedure Do_Breeding_Window;
    var
      RefCon: longint;          {RefCon for controls}
      code: integer;            {Location of event in window or controls}
      theValue: integer;        {Current value of a control}
      whichWindow: WindowPtr;{Window pointer where event happened}
      myPt: Point;              {Point where event happened}
      theControl: ControlHandle;{Handle for a control}
      MyErr: OSErr;             {OS error returned}

  begin                         {Start of Window handler}
    if (MyWindow <> nil) then{Handle only when the window is valid}
      begin
        code := FindWindow(myEvent.where, whichWindow);{Get where in window and which window}

        if (myEvent.what = MouseDown) and (MyWindow = whichWindow) then{}
          begin        {}
            myPt := myEvent.where;{Get mouse position}
```

```
            with MyWindow^.portBits.bounds do{Make it relative}
              begin
                myPt.h := myPt.h + left;
                myPt.v := myPt.v + top;
              end;
            evolve(myPt)
          end;

        if (MyWindow = whichWindow) and (code = inContent) then{for our window}
          begin

            code := FindControl(myPt, whichWindow, theControl);{Get type of control}

            if (code <> 0) then{Check type of control}
              code := TrackControl(theControl, myPt, nil);{Track the control}

          end;                      {End for if (MyWindow=whichWindow)}
      end;                          {End for if (MyWindow<>nil)}
  end;                              {End of procedure}

        {=================================}


end.                                {End of unit}
```

```
unit Preferences;

{File name:  Preferences.Pas  }
{Function: Handle a dialog}
{History: 12/12/90 Original by Prototyper.    }
{                    }


interface

  uses
    MyGlobals, Boxes, Ted, Breeding_Window;

  procedure D_Preferences;

implementation

  const                              {These are the item numbers for controls in the Dialog}
    I_OK = 1;
    I_Colour = 2;
    I_Sideways = 3;
    I_Centring = 9;
    I_x = 4;
    I_x5 = 5;
    I_x7 = 6;
    I_x9 = 7;
    I_x11 = 8;
  var
    theInput: TEHandle;
    ExitDialog: boolean;         {Flag used to exit the Dialog}
    DoubleClick: boolean;        {Flag to say that a double click on a list happened}
    MyPt: Point;                 {Current list selection point}
    MyErr: OSErr;                {OS error returned}

  procedure D_Preferences;
    var
      GetSelection: DialogPtr;{Pointer to this dialog}
      tempRect: Rect;         {Temporary rectangle}
      DType: Integer;         {Type of dialog item}
      Index: Integer;         {For looping}
      DItem: Handle;          {Handle to the dialog item}
      CItem, CTempItem: controlhandle;{Control handle}
      sTemp: Str255;              {Get text entered, temp holding}
      itemHit: Integer;       {Get selection}
      temp: Integer;          {Get selection, temp holding}
      dataBounds: Rect;       {Rect to setup the list}
      cSize: Point;           {Pointer to a cell in a list}
      Icon_Handle: Handle;    {Temp handle to read an Icon into}
      NewMouse: Point;            {Mouse location during tracking Icon presses}
      InIcon: boolean;        {Flag to say pressed in an Icon}
      ThisEditText: TEHandle; {Handle to get the Dialogs TE record}
      TheDialogPtr: DialogPeek;{Pointer to Dialogs definition record, contains the TE record}

    {This is an update routine for non-controls in the dialog}
    {This is executed after the dialog is uncovered by an alert}
    procedure Refresh_Dialog;          {Refresh the dialogs non-controls}
      var
        rTempRect: Rect;          {Temp rectangle used for drawing}

    begin
      SetPort(GetSelection);      {Point to our dialog window}
      GetDItem(GetSelection, I_OK, DType, DItem, tempRect);{Get the item handle}
      PenSize(3, 3);              {Change pen to draw thick default outline}
```

```
          InsetRect(tempRect, -4, -4);{Draw outside the button by 1 pixel}
          FrameRoundRect(tempRect, 16, 16); {Draw the outline}
          PenSize(1, 1);              {Restore the pen size to the default value}

        end;


      begin                           {Start of dialog handler}
        GetSelection := GetNewDialog(8, nil, Pointer(-1));{Bring in the dialog resource}
        ShowWindow(GetSelection);{Open a dialog box}
        SelectWindow(GetSelection);{Lets see it}
        SetPort(GetSelection);    {Prepare to add conditional text}

        TheDialogPtr := DialogPeek(GetSelection);{Get to the inner record}
        ThisEditText := TheDialogPtr^.textH;{Get to the TE record}
        HLock(Handle(ThisEditText));{Lock it for safety}
        ThisEditText^^.txSize := 12;{TE Point size}
        TextSize(12);             {Window Point size}
        ThisEditText^^.txFont := systemFont;{TE Font ID}
        TextFont(systemFont);         {Window Font ID}
        ThisEditText^^.txFont := 0;{TE Font ID}
        ThisEditText^^.fontAscent := 12;{Font ascent}
        ThisEditText^^.lineHeight := 12 + 3 + 1;{Font ascent + descent + leading}
        HUnLock(Handle(ThisEditText));{UnLock the handle when done}



                {Setup initial conditions}
        GetDItem(GetSelection, I_x9, DType, DItem, tempRect);{Get the item handle}
        NumToString(NRows, sTemp);
        SetIText(DItem, sTemp);       {Set the current value of NRows into dialog}

        GetDItem(GetSelection, I_x11, DType, DItem, tempRect);{Get the item handle}
        NumToString(NCols, sTemp);
        SetIText(DItem, sTemp);       {Set the current value of NCols into dialog}



        Refresh_Dialog;           {Draw any Lists, popups, lines, or rectangles}

        ExitDialog := FALSE;          {Do not exit dialog handle loop yet}

        GetDItem(GetSelection, I_Colour, DType, DItem, tempRect);{Get item information}
        CItem := Pointer(DItem);
        if WantColor then {Set check box to register present state of WantColor}
          temp := 1
        else
          temp := 0;
        SetCtlValue(CItem, temp);

        GetDItem(GetSelection, I_Centring, DType, DItem, tempRect);{Get item information}
        CItem := Pointer(DItem);
        if Centring then {Set check box to register present state of WantColor}
          temp := 1
        else
          temp := 0;
        SetCtlValue(CItem, temp);

        GetDItem(GetSelection, I_SideWays, DType, DItem, tempRect);{Get item information}
        CItem := Pointer(DItem);
        if Sideways then  {Set check box to register present state of Sideways}
          temp := 1
        else
          temp := 0;
        SetCtlValue(CItem, temp);
```

```
repeat                      {Start of dialog handle loop}
  ModalDialog(nil, itemHit);{Wait until an item is hit}
  GetDItem(GetSelection, itemHit, DType, DItem, tempRect);{Get item information}
  CItem := Pointer(DItem);{Get the control handle}

            {Handle it real time}
  if (ItemHit = I_OK) then{Handle the Button being pressed}
    begin
                {?? Code to handle this button goes here}
      ExitDialog := TRUE;{Exit the dialog when this selection is made}
    end;                    {End for this item selected}


  if (ItemHit = I_Colour) then{Handle the checkbox being pressed}
    begin
      temp := GetCtlValue(CItem);{Get the current Checkbox value}
      SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
      if (temp = 0) then{Do all CHECKED linkages}
        begin
        end                 {End for this item checked}
      else              {Do all UNCHECKED linkages}
        begin
        end;                {End for this item unchecked}

    end;                    {End for this item selected}

  if (ItemHit = I_Centring) then{Handle the checkbox being pressed}
    begin
      temp := GetCtlValue(CItem);{Get the current Checkbox value}
      SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}
      if (temp = 0) then{Do all CHECKED linkages}
        begin
        end                 {End for this item checked}
      else              {Do all UNCHECKED linkages}
        begin
        end;                {End for this item unchecked}

    end;                    {End for this item selected}


  if (ItemHit = I_Sideways) then{Handle the checkbox being pressed}
    begin
      if sideways then
        temp := 1
      else
        temp := 0;
      SetCtlValue(CItem, temp);
      temp := GetCtlValue(CItem);{Get the current Checkbox value}
      SetCtlValue(CItem, (temp + 1) mod 2);{Toggle the value to the opposite}

      if (temp = 0) then{Do all CHECKED linkages}
        begin
        end                 {End for this item checked}
      else              {Do all UNCHECKED linkages}
        begin
        end;                {End for this item unchecked}

    end;                    {End for this item selected}


until ExitDialog;           {Handle dialog items until exit selected}
```

```
            {Get results after dialog}
    GetDItem(GetSelection, I_Colour, DType, DItem, tempRect);{Get the Checkbox handle}
    CItem := Pointer(DItem);{Change dialog handle to control handle}
    temp := GetCtlValue(CItem);{Get the checkbox value}
    GetDItem(GetSelection, I_Colour, DType, DItem, tempRect);{Get item information}
    CItem := Pointer(DItem);
    if temp = 1 then
      wantColor := true
    else
      wantColor := false;
        {??? HANDLE THE CHECKBOX RESULT FOR  Colour HERE}

    GetDItem(GetSelection, I_Centring, DType, DItem, tempRect);{Get the Checkbox handle}
    CItem := Pointer(DItem);{Change dialog handle to control handle}
    temp := GetCtlValue(CItem);{Get the checkbox value}
    GetDItem(GetSelection, I_Centring, DType, DItem, tempRect);{Get item information}
    CItem := Pointer(DItem);
    if temp = 1 then
      Centring := true
    else
      Centring := false;
        {??? HANDLE THE CHECKBOX RESULT FOR  Centring HERE}

    GetDItem(GetSelection, I_Sideways, DType, DItem, tempRect);{Get the Checkbox handle}
    CItem := Pointer(DItem);{Change dialog handle to control handle}
    temp := GetCtlValue(CItem);{Get the checkbox value}
            {??? HANDLE THE CHECKBOX RESULT FOR  Sideways HERE}
    if temp = 1 then
      sideways := true
    else
      sideways := false;

    GetDItem(GetSelection, I_x9, DType, DItem, tempRect);{Get the item handle}
    GetIText(DItem, sTemp);{Get the text entered}
            {??? HANDLE THE STRING ENTERED FOR 3  HERE}
    StringToNum(sTemp, NRows);

    GetDItem(GetSelection, I_x11, DType, DItem, tempRect);{Get the item handle}
    GetIText(DItem, sTemp);{Get the text entered}
            {??? HANDLE THE STRING ENTERED FOR 5  HERE}
    StringToNum(sTemp, NCols);
    MidBox := 1 + (NRows * NCols) div 2;
    DisposDialog(GetSelection);{Flush the dialog out of memory}
    Open_Breeding_Window(theInput);
  end;                                {End of procedure}

end.                                  {End of unit}
```

```
unit  About_Arthromorphs;

{File  name:  About_Arthromorphs.Pas}
{Function: Handle a Window}
{History:  12/12/90  Original  by  Prototyper.     }

interface



    {Initialize us so all our routines can be activated}
  procedure  Init_About_Arthromorphs;

    {Close  our  window}
  procedure Close_About_Arthromorphs (whichWindow: WindowPtr; var theInput: TEHandle);

    {Open our window and draw everything}
  procedure Open_About_Arthromorphs (var theInput: TEHandle);

    {Update our window, someone uncovered a part of us}
  procedure Update_About_Arthromorphs (whichWindow: WindowPtr);

    {Handle action to our window, like controls}
  procedure Do_About_Arthromorphs (myEvent: EventRecord; var theInput: TEHandle);

implementation

  var
    MyWindow: WindowPtr;             {Window  pointer}
    tempRect, temp2Rect: Rect;       {Temporary  rectangle}
    Index: Integer;              {For  looping}
    CtrlHandle: ControlHandle;{Control  handle}
    sTemp: Str255;                 {Get text entered, temp holding}

{=================================}

    {Initialize us so all our routines can be activated}
  procedure  Init_About_Arthromorphs;

  begin                              {Start  of  Window  initialize  routine}
    MyWindow := nil;                 {Make sure other routines know we are not valid yet}
  end;                               {End of procedure}

{=================================}

    {Close  our  window}
  procedure  Close_About_Arthromorphs;

  begin                              {Start  of  Window  close  routine}
    if (MyWindow <> nil) and ((MyWindow = whichWindow) or (ord4(whichWindow) = -1)) then{See if we should close this win
      begin
        DisposeWindow(MyWindow);{Clear  window  and  controls}
        MyWindow := nil;           {Make sure other routines know we are open}
      end;                         {End for if (MyWindow<>nil)}
  end;                               {End of procedure}

{=================================}

    {Update our window, someone uncovered a part of us}
  procedure  UpDate_About_Arthromorphs;
    var
      SavePort: WindowPtr;           {Place to save the last port}
      sTemp: Str255;                 {Temporary  string}
  begin                                 {Start of Window update routine}
    if (MyWindow <> nil) and (MyWindow = whichWindow) then{Handle an open when already opened}
      begin
        GetPort(SavePort);          {Save the current port}
        SetPort(MyWindow);              {Set the port to my window}
        TextFont(systemFont);           {Set the font to draw in}
            {Draw a string of text,  }
        SetRect(tempRect, 16, 45, 272, 69);
        sTemp := 'By Ted Kaehler and Richard Dawkins';
        TextBox(Pointer(ord(@sTemp) + 1), length(sTemp), tempRect, teJustLeft);
        TextFont(applFont);         {Set the default application font}

        DrawControls(MyWindow);{Draw all the controls}
        SetPort(SavePort);          {Restore the old port}
      end;                          {End for if (MyWindow<>nil)}
  end;                                  {End of procedure}

{================================}

    {Open our window and draw everything}
  procedure Open_About_Arthromorphs;
    var
      Index: Integer;               {For looping}
      dataBounds: Rect;             {For making lists}
      cSize: Point;                 {For making lists}

  begin                                 {Start of Window open routine}

    if (MyWindow = nil) then            {Handle an open when already opened}
      begin
        MyWindow := GetNewWindow(3, nil, Pointer(-1));{Get the window from the resource file}
        SetPort(MyWindow);              {Prepare to write into our window}

        ShowWindow(MyWindow);    {Show the window now}
        SelectWindow(MyWindow);{Bring our window to the front}

      end                           {End for if (MyWindow<>nil)}
    else
      SelectWindow(MyWindow);{Already open, so show it}

  end;                                  {End of procedure}

{================================}

    {Handle action to our window, like controls}
  procedure Do_About_Arthromorphs;
    var
      RefCon: longint;                  {RefCon for controls}
      code: integer;                    {Location of event in window or controls}
      theValue: integer;                {Current value of a control}
      whichWindow: WindowPtr;               {Window pointer where event happened}
      myPt: Point;                      {Point where event happened}
      theControl: ControlHandle;        {Handle for a control}
      MyErr: OSErr;                     {OS error returned}

  begin                                 {Start of Window handler}
    if (MyWindow <> nil) then           {Handle only when the window is valid}
      begin
        code := FindWindow(myEvent.where, whichWindow);{Get where in window and which window}

        if (myEvent.what = MouseDown) and (MyWindow = whichWindow) then{}
          begin                     {}
            myPt := myEvent.where;{Get mouse position}
            with MyWindow^.portBits.bounds do{Make it relative}
              begin
                myPt.h := myPt.h + left;
                myPt.v := myPt.v + top;
              end;

          end;

        if (MyWindow = whichWindow) and (code = inContent) then{for our window}
          begin

            code := FindControl(myPt, whichWindow, theControl);{Get type of control}

            if (code <> 0) then{Check type of control}
              code := TrackControl(theControl, myPt, nil);{Track the control}

          end;                    {End for if (MyWindow=whichWindow)}
      end;                      {End for if (MyWindow<>nil)}
  end;                          {End of procedure}

{================================}


end.                            {End of unit}
```

```
unit HandleTheMenus;

{File name :  HandleTheMenus.Pas  }
{Function: Handle all menu selections.}
{          This procedure is called when a menu item is selected.}
{          There is one CASE statement for all Lists.  There is}
{          another CASE for all the commands in each List.}
{History: 12/12/90  Original  by  Prototyper.     }
{                  }

interface

  uses
    MyGlobals, Ted, Richard, Error_Alert, Preferences, Engineering_Window, Genome_Window, Breeding_Window,
 About_Arthromorphs, InitTheMenus;

  procedure Handle_My_Menu (var doneFlag: boolean; theMenu, theItem: integer; var theInput: TEHandle);{Handle menu selec

implementation

  procedure Handle_My_Menu;          {Handle menu selections realtime}
    const
      L_Apple = 1001;             {Menu list}
      C_About_Arthromorphs = 1;
      L_File = 1002;              {Menu list}
      C_New = 1;
      C_Open = 2;
      C_Close = 4;
      C_Save = 5;
      C_Save_As = 6;
      C_Quit = 8;
      L_Edit = 1003;              {Menu list}
      C_Undo = 1;
      C_Cut = 3;
      C_Copy = 4;
      C_Paste = 5;
      C_Clear = 6;
      C_Select_All = 7;
      C_Show_Clipboard = 9;
      L_Operation = 1004;         {Menu list}
      C_Breed = 1;
      C_Show_as_Text = 2;
      C_Engineer = 3;
      L_View = 1005;              {Menu list}
      C_Preferences = 1;
    var
      DNA: integer;             {For opening DAs}
      BoolHolder: boolean;      {For SystemEdit result}
      DAName: Str255;             {For getting DA name}
      SavePort: GrafPtr;          {Save current port when opening DAs}

  begin                           {Start of procedure}

    case theMenu of               {Do selected menu list}

      L_Apple:
        begin
          case theItem of{Handle all commands in this menu list}
            C_About_Arthromorphs:
              begin
                Open_About_Arthromorphs(theInput);{Open a window for this menu selection}
              end;
            otherwise   {Handle the DAs}
              begin         {Start of Otherwise}
                GetPort(SavePort);{Save the current port}
                GetItem(AppleMenu, theItem, DAName);{Get the name of the DA selected}
                DNA := OpenDeskAcc(DAName);{Open the DA selected}
                SetPort(SavePort);{Restore to the saved port}
              end;          {End of Otherwise}

          end;              {End of item case}
        end;                    {End for this list}

      L_File:
        begin
          case theItem of{Handle all commands in this menu list}
            C_New:
              begin
                NewMinimal;
                Open_Breeding_Window(theInput);{Open a window for this menu selection}
              end;
            C_Open:
              begin
                Open_Breeding_Window(theInput);{Open a window for this menu selection}
                LoadArthromorph;
                            {Call the SFGetFile OS routine}
              end;
            C_Close:
              begin
                            {Call the SFPutFile OS routine}
              end;
            C_Save:
              begin
                SaveArthromorph;
                            {Call the SFPutFile OS routine}
              end;
            C_Save_As:
              begin
                SaveArthromorph;
                            {Call the SFPutFile OS routine}
              end;
            C_Quit:
              begin
                doneFlag := TRUE;
              end;
            otherwise
              begin         {Start of the Otherwise}
              end;          {End of Otherwise}

          end;              {End of item case}
        end;                    {End for this list}

      L_Edit:
        begin
          BoolHolder := SystemEdit(theItem - 1);{Let the DA do the edit to itself}
          if not (BoolHolder) then{If not a DA then we get it}
            begin               {Handle by using a Case statment}
              case theItem of{Handle all commands in this menu list}
                C_Undo:
                  begin
                    A_Error_Alert;{Call a alert for this menu selection}
                  end;
                C_Cut:
                  begin
                                {?? ADD IN HERE WHAT THIS COMMAND SHOULD DO}
                  end;
                C_Copy:
                  begin
                                {?? ADD IN HERE WHAT THIS COMMAND SHOULD DO}
                  end;
                C_Paste:
                  begin
                                {?? ADD IN HERE WHAT THIS COMMAND SHOULD DO}
                  end;
                C_Clear:
                  begin
                                {?? ADD IN HERE WHAT THIS COMMAND SHOULD DO}
                  end;
                C_Select_All:
                  begin
                                {?? ADD IN HERE WHAT THIS COMMAND SHOULD DO}
                  end;
                C_Show_Clipboard:
                  begin
                                {?? ADD IN HERE WHAT THIS COMMAND SHOULD DO}
                  end;
                otherwise{Send to a DA}
                  begin   {Start of the Otherwise}
                  end;    {End of Otherwise}

              end;            {End of not BoolHolder}
            end;              {End of item case}
        end;                {End for this list}

      L_Operation:
        begin
          case theItem of{Handle all commands in this menu list}
            C_Breed:
              begin
                Open_Breeding_Window(theInput);{Open a window for this menu selection}
                Breed;
              end;
            C_Show_as_Text:
              begin
                PrintMiddle;
                Close_Breeding_Window(WindowPtr(ord4(-1)),  theInput);
              end;
            C_Engineer:
              begin
                repeat
                  D_Engineering_Window;
                until  AgreeToExit;
                Close_Genome_Window(WindowPtr(ord4(-1)), theInput);{Close a window for this menu selection}
              end;
            otherwise
              begin      {Start of the Otherwise}
              end;       {End of Otherwise}

          end;           {End of item case}
        end;                {End for this list}

      L_View:
        begin
          case theItem of{Handle all commands in this menu list}
            C_Preferences:
              begin
                Close_Breeding_Window(WindowPtr(ord4(-1)),  theInput);
                D_Preferences;{Call a dialog for this menu selection}
              end;
          otherwise
           begin  {Start of the Otherwise}
           end;   {End of Otherwise}

         end;     {End of item case}
        end;        {End for this list}

       otherwise
        begin       {Start of the Otherwise}
        end;        {End of Otherwise}

      end;          {End for the Lists}

      HiliteMenu(0);  {Turn menu selection off}
     end;             {End of procedure Handle_My_Menu}

end.                {End of unit}
```

```
unit initialize;
interface
  uses
    MyGlobals, Ted, Richard, Breeding_Window;
  procedure MyInit;

implementation
  var
    DocumentMessage, DocumentCount: integer;

  procedure MyInit;
    var
      theInput: TEHandle;
  begin
    thickscale := 1;
    wantColor := false;
    sideways := false;
    resizing := false;
    centring := false;
    verticalOffset := 0;
    HorizontalOffset := 0;
    overlap := 1.0; {in case animal has no value}
    NRows := 3;
    NCols := 5; {Defaults}
    NBoxes := NRows * NCols;
    MidBox := 1 + (NRows * NCols) div 2;
    upregion := NewRgn;
    InitBoneyard;
    CountAppFiles(DocumentMessage, DocumentCount);
    if DocumentCount > 0 then {at least one biomorph double-clicked}
      begin
        StartDocument;
      end;
    startingUp := true;
    MakeAllBodyMutations(true);
    MakeAllAtomMutations(true);
    mutationPressure := zero;
    FocusOfAttention := AnySegment;
    Open_Breeding_Window(theInput);
    Breed;
  end;

end.
```

```
{The Project should have the following files in it:      }
{      µRunTime.lib      LSP This is for main Pascal runtime library}
{      Interface.lib       LSP This is the Mac trap interfaces}
{       PrintCalls.Lib     LSP This is the print routine library interface}
{      MacPrint.p          LSP This is the print equates for print calls}
{     InitTheMenus.Pas       This initializes the Menus.}
{      Error_Alert          Alert}
{     Preferences        Modal Dialog}
{     Engineering_Window        Modeless Dialog}
{    Genome_Window       Window}
{     Breeding_Window       Window}
{      About_Arthromorphs         Window}
{   HandleTheMenus          Handle the menu selections.}
{Set  RUN OPTIONS to use the  resource file Brand_New.RSRC  }
{ RMaker file to use is Brand_New.R  }
{   Brand_New.Pas       Main program  }
program  Brand_New;
{Program name:  Brand_New.Pas  }
{Function:  This is the main module for this program.  }
{History:  12/15/90  Original  by  Prototyper.     }
{                    }
  uses
    MyGlobals, Error_Alert, Preferences, Engineering_Window, Genome_Window, Breeding_Window, About_Arthromorphs,
 InitTheMenus, HandleTheMenus, Initialize;
  var                                    {Main  variables}
    myEvent: EventRecord;            {Event  record  for  all  events}
    doneFlag: boolean;              {Exit  program  flag}
    code: integer;                  {Determine  event  type}
    whichWindow: WindowPtr;          {See  which  window  for  event}
    tempRect, OldRect: Rect;       {Rect  for  dragging}
    mResult: longint;               {Menu  list  and  item  selected  values}
    theMenu, theItem: integer;{Menu  list  and  item  selected}
    chCode: integer;                {Key  code}
    ch: char;                     {Key  pressed  in  Ascii}
    theInput: TEHandle;             {Used  in  text  edit  selections}
    Is_A_Dialog: boolean;           {Flag  for  modless  dialogs}
    myPt:  Point;                   {Temp Point, used in Zoom}

begin                                  {Start  of  main  body}

  MoreMasters;                      {This  reserves  space  for  more  handles}
  InitGraf(@thePort);               {Quickdraw  Init}
  InitFonts;                    {Font  manager  init}
  InitWindows;                  {Window  manager  init}
  InitMenus;                    {Menu  manager  init}
  TEInit;                       {Text  edit  init}
  InitDialogs(nil);             {Dialog  manager}

  FlushEvents(everyEvent,  0);{Clear  out  all  events}
  InitCursor;                   {Make  an  arrow  cursor}

  doneFlag := FALSE;                {Do  not  exit  program  yet}

  Init_My_Menus;                    {Initialize  menu  bar}

  theInput  :=  nil;                {Init  to  no  text  edit  selection  active}
  Init_Genome_Window;           {Initialize  the  window  routines}
  Init_Breeding_Window;         {Initialize  the  window  routines}
  Init_About_Arthromorphs;      {Initialize  the  window  routines}
  MyInit;

  repeat                            {Start  of  main  event  loop}
    if  (theInput <> nil)  then{See  if  a  TE  is  active}
```

```
      TEIdle(theInput);     {Blink the cursor if everything is ok}
    SystemTask;                 {For support of desk accessories}

    if GetNextEvent(everyEvent, myEvent) then{If event then...}
      begin                     {Start handling the event}
        code := FindWindow(myEvent.where, whichWindow);{Get which window the event happened in}

        Is_A_Dialog := IsDialogEvent(myEvent);{See if a modeless dialog event}
        if Is_A_Dialog then{Handle a dialog event}
          begin                 {}
            if (myEvent.what = UpDateEvt) then{Handle the update of a Modeless Dialog}
              begin             {}
                whichWindow := WindowPtr(myEvent.message); {Get the window the update is for}
                BeginUpdate(whichWindow);{Set update clipping area}
                EndUpdate(whichWindow);{Return to normal clipping area}
              end               {}
          end                   {End of Is_A_Dialog}
        else                    {Otherwise handle a window}
          begin                 {}

            case myEvent.what of{Decide type of event}
              MouseDown:{Mouse button pressed}
                begin           {Handle the pressed button}
                  if (code = inMenuBar) then{See if a menu selection}
                    begin       {Get the menu selection and handle it}
                      mResult := MenuSelect(myEvent.Where);{Do menu selection}
                      theMenu := HiWord(mResult);{Get the menu list number}
                      theItem := LoWord(mResult);{Get the menu list item number}
                      Handle_My_Menu(doneFlag, theMenu, theItem, theInput);{Handle the menu}
                    end;        {End of inMenuBar}

                  if (code = InDrag) then{See if in a window drag area}
                    begin       {Do dragging the window}
                      tempRect := screenbits.bounds;{Get screen area,  l,t,r,b, drag area}
                      SetRect(tempRect, tempRect.Left + 10, tempRect.Top + 25, tempRect.Right - 10, tempRect.Bottom - 10);{}
                      DragWindow(whichWindow, myEvent.where, tempRect);{Drag the window}
                    end;        {End of InDrag}

                  if ((code = inGrow) and (whichWindow <> nil)) then{In a grow area of the window}
                    begin       {Handle the growing}
                      SetPort(whichWindow);{Get ready to draw in this window}


                      myPt := myEvent.where;{Get mouse position}
                      GlobalToLocal(myPt);{Make it relative}

                      OldRect := WhichWindow^.portRect;{Save the rect before resizing}

                      with screenbits.bounds do{use the screens size}
                        SetRect(tempRect, 15, 15, (right - left), (bottom - top) - 20);{l,t,r,b}

{EraseRect(Oldrect);}


                      mResult := GrowWindow(whichWindow, myEvent.where, tempRect);{Grow it}
                      SizeWindow(whichWindow, LoWord(mResult), HiWord(mResult), TRUE);{Resize to result}
                      Resizing := true;
                      InvalRect(WhichWindow^.portRect);


                      case (GetWRefCon(whichWindow)) of{Do the appropiate window}
                        2:
                          Resized_Breeding_Window(OldRect, whichWindow);{Resized this window}
                        otherwise{Handle others}
                          begin{Others}
                          end;{End of the otherwise}
                      end;{End of the case}

                      SetPort(whichWindow);{Get ready to draw in this window}

                      SetRect(tempRect, 0, myPt.v - 15, myPt.h + 15, myPt.v + 15); {Position for horz scrollbar area}
                      EraseRect(tempRect);{Erase old area}
                      InvalRect(tempRect);{Flag us to update it}
                      SetRect(tempRect, myPt.h - 15, 0, myPt.h + 15, myPt.v + 15);  {Position for vert scrollbar area}
                      EraseRect(tempRect);{Erase old area}
                      InvalRect(tempRect);{Flag us to update it}
                      DrawGrowIcon(whichWindow);{Draw the grow Icon again}


                    end;      {End of doing the growing}

                  if (code = inZoomIn) or (code = inZoomOut) then{Handle Zooming windows}
                    begin     {}
                      if (WhichWindow <> nil) then{See if we have a legal window}
                        begin{}
                          SetPort(whichWindow);{Get ready to draw in this window}

                          myPt := myEvent.where;{Get mouse position}
                          GlobalToLocal(myPt);{Make it relative}

                          OldRect := whichWindow^.portRect;{Save the rect before resizing}

                          if TrackBox(whichWindow, myPt, code) then{Zoom it}
                            begin{}
                              ZoomWindow(WhichWindow, code, TRUE);{Resize to result}
                              SetRect(tempRect, 0, 0, 32000, 32000);{l,t,r,b}
                              EraseRect(tempRect);{Make sure we update the whole window effectively}
                              InvalRect(tempRect);{Tell ourselves to update, redraw, the window contents}
                              case (GetWRefCon(whichWindow)) of{Do the appropiate window}
                                2:
                                  Resized_Breeding_Window(OldRect, whichWindow);{Resized this window}
                                otherwise{Handle others dialogs}
                                  begin{Others}
                                  end;{End of the otherwise}
                              end;{End of the case}
                            end;{}
                        end;{}
                    end;      {}

                  if (code = inGoAway) then{See if in a window goaway area}
                    begin     {Handle the goaway button}
                      if TrackGoAway(whichWindow, myEvent.where) then{See if mouse released in GoAway box}
                        begin{Handle the GoAway}
                          case (GetWRefCon(whichWindow)) of{Do the appropiate window}
                            1:
                              Close_Genome_Window(whichWindow, theInput);{Close this window}
                            2:
                              Close_Breeding_Window(whichWindow, theInput);{Close this window}
                            3:
                              Close_About_Arthromorphs(whichWindow, theInput);{Close this window}
                            otherwise{Handle others dialogs}
                              begin{Others}
                              end;{End of the otherwise}
                          end;{End of the case}
                        end;{End of TrackGoAway}
                    end;      {End of InGoAway}
```

```
    if (code = inContent) then{See if in a window}
      begin      {Handle the hit inside a window}
        if (whichWindow <> FrontWindow) then{See if already selected or not, in front if selected}
          SelectWindow(whichWindow){Select this window to make it active}
        else{If already in front the already selected}
          begin{Handle the button in the content}
            SetPort(whichWindow);{Get ready to draw in this window}
            case (GetWRefCon(whichWindow)) of{Do the appropiate window}
              1:
                Do_Genome_Window(myEvent, theInput);{Handle this window}
              2:
                Do_Breeding_Window(myEvent, theInput);{Handle this window}
              3:
                Do_About_Arthromorphs(myEvent, theInput);{Handle this window}
              otherwise{Handle others dialogs}
                begin{Others}
                end;{End of the otherwise}
            end;{End of the case}
          end;{End of else}
      end;       {End of inContent}

    if (code = inSysWindow) then{See if a DA selection}
      SystemClick(myEvent, whichWindow);{Let other programs in}

  end;           {End of MouseDown}

KeyDown, AutoKey:{Handle key inputs}
  begin          {Get the key and handle it}
    with myevent do{Check for menu command keys}
      begin      {}
        chCode := BitAnd(message, CharCodeMask);{Get character}
        ch := CHR(chCode);{Change to ASCII}
        if (Odd(modifiers div CmdKey)) then{See if Command key is down}
          begin{}
            mResult := MenuKey(ch);{See if menu selection}
            theMenu := HiWord(mResult);{Get the menu list number}
            theItem := LoWord(mResult);{Get the menu item number}
            if (theMenu <> 0) then{See if a list was selected}
              Handle_My_Menu(doneFlag, theMenu, theItem, theInput);{Do the menu selection}
            if ((ch = 'x') or (ch = 'X')) and (theInput <> nil) then{}
              TECut(theInput);{Handle a Cut in a TE area}
            if ((ch = 'c') or (ch = 'C')) and (theInput <> nil) then{}
              TECopy(theInput);{Handle a Copy in a TE area}
            if ((ch = 'v') or (ch = 'V')) and (theInput <> nil) then{}
              TEPaste(theInput);{Handle a Paste in a TE area}
          end{}
        else if (theInput <> nil) then{}
          TEKey(ch, theInput);{}
      end;       {}
  end;           {End for KeyDown,AutoKey}

UpDateEvt:{Update event for a window}
  begin          {Handle the update}
    whichWindow := WindowPtr(myEvent.message);{Get the window the update is for}
    BeginUpdate(whichWindow);{Set the clipping to the update area}
    case (GetWRefCon(whichWindow)) of{Do the appropiate window}
      1:
        Update_Genome_Window(whichWindow);{Update this window}
      2:
        Update_Breeding_Window(whichWindow);{Update this window}
      3:
        Update_About_Arthromorphs(whichWindow);{Update this window}
```

```
                    otherwise   {Handle others dialogs}
                      begin     {Others}
                      end;      {End of the otherwise}
                  end;          {End of the case}
                  EndUpdate(whichWindow);{Return to normal clipping area}
                end;          {End of UpDateEvt}

              DiskEvt:    {Disk inserted event}
                begin         {Handle a disk event}
                  if (HiWord(myevent.message) <> noErr) then{See if a diskette mount error}
                    begin     {due to unformatted diskette inserted}
                      myEvent.where.h := ((screenbits.bounds.Right - screenbits.bounds.Left) div 2) - (304 div 2);{Center horz}
                      myEvent.where.v := ((screenbits.bounds.Bottom - screenbits.bounds.Top) div 3) - (104 div 2);{Top 3ed
  vertically}
                      InitCursor;{Make sure it has an arrow cursor}
                      chCode := DIBadMount(myEvent.where, myevent.message);{Let the OS handle the diskette}
                    end;      {}
                end;      {End of DiskEvt}

              app1Evt:    {Check for events generated by this program}
                begin         {Start handling our events}
                  if (HiWord(myEvent.message) = 1) and (LoWord(myEvent.Message) = 1) then{See if OPEN event for this window
                    Open_Genome_Window(theInput);{Open the window}
                  if (HiWord(myEvent.message) = 2) and (LoWord(myEvent.Message) = 1) then{See if CLOSE event for this windo
                    Close_Genome_Window(WindowPtr(ord4(-1)), theInput);{Close the window}
                  if (HiWord(myEvent.message) = 1) and (LoWord(myEvent.Message) = 2) then{See if OPEN event for this window
                    Open_Breeding_Window(theInput);{Open the window}
                  if (HiWord(myEvent.message) = 2) and (LoWord(myEvent.Message) = 2) then{See if CLOSE event for this windo
                    Close_Breeding_Window(WindowPtr(ord4(-1)), theInput);{Close the window}
                  if (HiWord(myEvent.message) = 1) and (LoWord(myEvent.Message) = 3) then{See if OPEN event for this window
                    Open_About_Arthromorphs(theInput);{Open the window}
                  if (HiWord(myEvent.message) = 2) and (LoWord(myEvent.Message) = 3) then{See if CLOSE event for this windo
                    Close_About_Arthromorphs(WindowPtr(ord4(-1)), theInput);{Close the window}
                end;          {End handling our events}

              ActivateEvt:{Window activated event}
                begin         {Handle the activation}
                  whichWindow := WindowPtr(myevent.message);{Get the window to be activated}
                  if odd(myEvent.modifiers) then{Make sure it is Activate and not DeActivate}
                    begin{Handle the activate}
                      SelectWindow(whichWindow);{Activate the window by selecting it}
                      case (GetWRefCon(whichWindow)) of{Do the appropiate window}
                        2:
                          DrawGrowIcon(whichWindow);{Draw the Grow box}
                        otherwise{Handle others }
                          begin{Others}
                          end;{End of the otherwise}
                      end;{End of the case}
                    end;{End of Activate}
                end;          {End of ActivateEvt}

              otherwise       {Used for debugging, to see what other events are coming in}
                begin         {}
                            {?? ADDED FOR DEBUGGING, CATCHING OTHER EVENTS}
{}
                end;          {End of otherwise}

            end;              {End of case}

          end;                {End for not a modeless dialog event}
      end;                    {end of GetNextEvent}
  until doneFlag;                 {End of the event loop}
```

```
    end.                                {End of the program}
```