Subject: Re: Passing Image Data :)
Posted by Logan Lindquist on Tue, 23 Oct 2001 19:31:10 GMT
View Forum Message <> Reply to Message

Dr(.) Fanning,
[I thought Dr. had an period after it because it is an abbrevation of
doctor? I do not know what Andrew Cool is talking about.]

> What you want in your info structure image field is a pointer
> to the image:
>
> info= { image:Ptr_New(myimage), ...}

I am wondering if you could clear up a couple of things about pointers in
IDL. How come myimage does not have to be defined during initalization? Does
the statement above create space in memory for a variable of indefinite
size? It seems to operate this way., where the data in memory is allocated
once the data has to be stored to the pointer array. Maybe I am
understanding pointers incorrectly.

1.. The Pointer is created - a variable that 'points' to space in RAM
reserved for a variable of indefinate size.
2.. The data is read into RAM during the read_image.pro.
3.. The Pointer then needs to store the image data for future reference.
This is done by '*info.image = newimage'. Where newimage is the image data
in RAM.
4.. Is the data then copied into the space originally allocated for it or
does it simply change it's reference so as to point to the location in RAM
where the image data was read into?

> *info.image = newimage
>
> IDL takes care of all the memory management for you. You don't
> have to worry about it.

I went back and reviewed how pointers are treated in C++. I was wondering if
I made my Struct a pointer, could I access memebers of Struct's using the
'->'?

Thank You,

Logan Lindquist

Below is what I found on pointers in C++.
****

Pointers to Objects

Pointers can point to objects as well as to simple data types and arrays. We've seen many examples of objects defined and given a name, in statements like

```
Distance dist;
```

where an object called dist is defined to be of the Distance class.
Sometimes, however, we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. As we've seen, new returns a pointer to an unnamed object. Let's look at a short example program, ENGLPTR, that compares the two approaches to creating objects.

```
// englptr.cpp
// accessing member functions by pointer
 #include <iostream>
 using namespace std;
  ////////////////////////////////////////////////////////////// ////
 class Distance              //English Distance class
   {
  private:
      int feet;
      float inches;
  public:
       void getdist()        //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
        }
       void showdist()       //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
  ////////////////////////////////////////////////////////////// ////

 int main()

    {
     Distance dist;           //define a named Distance object
    dist.getdist();          //access object members
    dist.showdist();         //   with dot operator
    Distance* distptr;       //pointer to Distance
    distptr = new Distance;  //points to new Distance object
     distptr->getdist();      //access object members
     distptr->showdist();     //   with -> operator
     cout << endl;
```

```
    return 0;

    }
```

This program uses a variation of the English Distance class seen in previous chapters. The main() function defines dist, uses the Distance member function getdist() to get a distance from the user, and then uses showdist() to display it.

Referring to Members

ENGLPTR then creates another object of type Distance using the new operator, and returns a pointer to it called distptr.

The question is, how do we refer to the member functions in the object pointed to by distptr? You might guess that we would use the dot (.) membership-access operator, as in

```
distptr.getdist();   // won't work; distptr is not a variable
```

but this won't work. The dot operator requires the identifier on its left to be a variable. Since distptr is a pointer to a variable, we need another syntax. One approach is to dereference (get the contents of the variable pointed to by) the pointer:

```
(*distptr).getdist();  // ok but inelegant
```

However, this is slightly cumbersome because of the parentheses. (The parentheses are necessary because the dot operator (.) has higher precedence than the indirection operator (*). An equivalent but more concise approach is furnished by the membership-access operator ->, which consists of a hyphen and a greater-than sign:

```
distptr->getdist();   // better approach
```

As you can see in ENGLPTR, the -> operator works with pointers to objects in just the same way that the . operator works with objects. Here's the output of the program:

```
Enter feet: 10   ;this object uses the dot operator
Enter inches: 6.25
10'-6.25"
Enter feet: 6    ; this object uses the -> operator
Enter inches: 4.75
6'-4.75"
```