Structured data types

A more elegant way to deal with the *rgb* image is to employ *structured data types*. A *structure* is a aggregation of basic and structured data elements.

A C structure is declared as follows:

```
struct pix_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};
```

It is important to be aware that *struct pix_type is the name of a user defined structure type.* It is *not* the name of a variable. The use of _type as a suffix is not required but can help you remember what name is a type and what name is a variable.

To declare an *instance* (*variable*) of type *struct pix_type* use:

```
struct pix_type pixel;
```

`struct pix_type`	is the name of the *type*
`pixel`	is the name of a *variable* of type *struct pix_type*

To set or reference components of the *pixel* use:

```
pixel.r = 250;  // make Mr. Pixel yellow
pixel.g = 250;
pixel.b = 0;
```

A bad (but syntactically legal) idea.

The C language is sensitive to the context in which a name is used and so the following will not cause compile errors, *but its a very bad practice.*

```
struct pix
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pix pix;
```

Pointers to structures:

To declare a *pointer* to a *struct pix_type* use:

```
struct pix_type *pixptr;
```

Before using the pointer we must always make it point to something:

```
pixptr = (struct pix_type *)malloc(sizeof(struct pix_type));
```

To set or reference components of the *pix_type* to which it points use:

```
(*pixptr).r = 250;  // make Mr. *pixptr magenta
(*pixptr).g = 0;
(*pixptr).b = 250;
```

Warning: the C compiler is very picky about the location of the parentheses here.

Perhaps because of the painful nature of the above syntax, an alteranative "short hand" notation has evolved for accessing elements of structures through a pointer:

```
pixptr->r = 0;      // make Mr. pixptr-> cyan
pixptr->g = 250;
pixptr->b = 250;
```

This shorthand form is almost universally used.

Revisiting the color image

Space for a color image in binary *rgb* format can be allocated by:

```
struct pix_type *pixptr;
     :
pixptr = (struct pixptr *)malloc(sizeof(struct pix_type)  *
                          numrows * numcols);
```

To read the image data

```
pixcount = fread(imageloc, sizeof(struct pix_type), numrows *
                    numcols, stdin);

if (pixcount != numrows * numcols)
{
   fprintf(stderr, “pix count err - wanted %d got %d \n”,
               numrows * numcols, pixcount);
   exit(1);
}
```

To access a specific pixel

```
red   = (*(pixptr + row * numcols + col)).r;
green = (*(pixptr + row * numcols + col)).b;
blue  = (*(pixptr + row * numcols + col)).g;

or

red   = (pixptr + row * numcols + col)->r;
green = (pixptr + row * numcols + col)->g;
blue  = (pixptr + row * numcols + col)->b;

or even

red   = pixptr[row * numcols + col].r;
```

Use of *sizeof()* with structures

The *sizeof()* facility should *always* be used in dynamically allocation storage for structured data types and in reading and writing them. However, it is somewhat easy to do this incorrectly.

```
struct pix_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pix_type pix;
struct pix_type *pixptr;
```

The following all have the value 3

```
sizeof(pix)
sizeof(struct pix_type)
sizeof(*pixptr)
```

The following has the value 4

```
sizeof(struct pix_type *)
sizeof(pixptr)
```

Confusing these is a common source of some nasty program bugs.

Arrays within structures

An element of a structure may be an *array:*

```
struct gs_image_type
{
   int numrows;
   int numcols;
   unsigned char pixmap[600 * 800];
};

struct gs_image_type image;
```

Elements of the array are accessed in the usual way:

```
image.pixmap[row * numcols * col] = 125;
```

A rule that always works is:

Subscripts are required in code

- immediately following any name declared as an array
- and are allowed no where else!

Arrays of structures

We can also create an array of stucture types .

```
struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pixel_type pixmap[600 * 800];
```

To access an individual element of the array place the subscript next to the name it indexes

```
pixmap[15].r = 250;
```

Arrays of structures containing arrays

It is also possible to create an array of stuctures containing arrays

```
struct gs_image_type images[20];
```

Elements of the array are accessed in the usual way:

```
images[4].pixmap[5] = 125;
```

Structures containing structures

It is common for structures to contain elements which are themselves structures or arrays of structures. In these cases *the structure definitions should appear in "inside-out" order.*

This is done to comply with the usual rule of not referencing a name before it is defined.

```
struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct img_type
{
   int numrows;
   int numcols;
   struct pixel_type pixdata[800 * 600];
};


struct img_type image;

image.pixdata[140].r = 222;
```

Structures containing pointers to structures

Hard coding of image dimensions is an undesirable because we want to create programs that can create images of any size.

```
struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct rgb_image_type
{
   int numrows;
   int numcols;
   struct pixel_type *pixdata;
};
```

The follow examples shows how to dynamically allocate a *rgb_image_type* and access pixels.

```
   struct rgb_image_type *rgb_ptr;
   struct pixel_type     *pixptr;

   int rows = atoi(argv[1]);
   int cols = atoi(argv[2]);

   rgb_ptr = malloc(sizeof *rgb_ptr);
   rgb_ptr->pixdata = malloc(sizeof(struct pixel_type) *
                             rows, cols);
   rgb_ptr->numrows = rows;
   rgb_ptr->numcols = cols;

// set the red component of row 11 col 22 to 250

   pixptr = rgb_ptr->pixdata;
   pixptr =  pixptr + 11 * numcols + 22;
   pixptr->r = 250;
```

How to read a .ppm header

```
int read_color()
{
   char id[3] = {0, 0, 0};
   int  vals[6];
   int  count = 0;
   char buf[256];
   int  howmany;

   fgets(id, 3, stdin);

   while (count < 3)
   {
      howmany = fscanf(stdin, "%d", &vals[count]);
      if (howmany == 0)
         fgets(buf, 256, stdin);
      else
         count = count + 1;
   }
   fgets(buf,   256, stdin);
   outrows = inrows = vals[1];
   outcols = incols = vals[0];

   in_image = malloc(sizeof(struct pix_type) * vals[0] * vals[1]);

   howmany = fread(in_image, sizeof(struct pix_type),
                                        inrows * incols, stdin);
```

Two dimensional arrays in C

Two dimensional arrays are declared as follows:

```
double  x[4][5];
```

The declaration above creates 4 x 5 = 20 double precision values. Specific elements are accessed as follows:

```
x[2][3] = sqrt(10.0);
```

As with single dimension arrays the legal values of the first subscript are 0, 1, 2, 3 and the legal values of the second are 0, 1, 2, 3, 4

The values are stored in the following order:

```
x[0][0],  x[0][1], ... x[0][4],  x[1][0], .., x[3][4]
```

This is consistent with subscripting techniques commonly used in mathematics in which the first subscript commonly represents a *row* and the second represents a *column.*

The name *x[i]* represents a pointer to the *ith* row of the array.

Thus the previous technique *(row * numcols + col)* can be used to access specific elements

```
double *p = x[0];
 p = p + 2 * 5  + 3
*p = sqrt(10.0);
```

Passing array rows as parameters

Suppose we have a two dimensional array called *filter*.

```
float filter[3][3] =
{

   {  0.110, 0.140, 0.110 },
   {  0.140, 0.000, 0.140 },
   {  0.110, 0.140, 0.110 },
};
```

Applying one row of the filter to 3 adjacent pixels is what is called and "inner product" operation

```
unsigned char inner_prod(
struct pix_type *pix,
float *filt)
{
   float sum = 0.0;
   int   j;

   for (j = 0; j < 3; j++)
   {
      sum = sum + pix->g * (*filt);
      pix = pix + 1;
      filt = filt + 1;
   }
   return ((unsigned char)sum);
}
```

Thus the filter may be applied in a standards compliant manner as follows:

```
void make_gspix(
int row,
int col)
{
   int i;
   struct pix_type *pixloc;
   unsigned char sum = 0;

/* If (row, col) is on the border
   copy input pixel to output pixel and return */


/* Apply our filter one row of the filter at a time */

   for (i = 0; i < 3; i++)
   {
    /* set pixloc to point to (row + i – 1, col – 1)
       in the current INPUT IMAGE                    */
    /* now compute next component of sum             */

       sum = sum + inner_prod(pixloc, filter[i]);
   }

/* Save sum at (row, col) in CURRENT OUTPUT image */

   return;
}
```

Passing two dimensional arrays as parameters

```
/**/
/* Add two three by three matrices together */
void matadd(
double inleft[][],
double inright[][],
double out[][])
{
   double x;

   x = inleft[2][1];

}

 ==> gcc -c twod.c
twod.c: In function `vl_matmult3':
twod.c:11: invalid use of array with unspecified bounds
```

The problem here is that the compiler has now way to know *how many columns there are in each row of the matrix.* Recall the way that we implicitly handled two dimensional pixmaps

```
*(imageloc + row * numcols + col)
```

To obtain the offset of the start of a particular row we had to know how many columns were in the pixmap.

Defining arrays passed as parameters

The correct way is to specify the length of the columns (or both rows and columns)

```
void matadd(
double inleft[][3],
double inright[][3],
double out[3][3])
{
   double x;
   int i, j;
   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
         out[i][j] = inleft[i][j] + inright[i][j];


}
 ==> gcc -c -Wall twod3.c
 ==>
```

Structures as function parameters and return values

Structures as parameters

The example program below illustrates two aspects of passing structures

- A *struct*, like an *int* may be passed as a parameter to a function.

In fact the process works just like passing an in that:

- The complete structure is *copied* to the stack
- The function is *unable* to modify the caller's copy of the variable

```
/* p41.c */

struct work_type
{
   int w;
};

void fun(struct work_type q)
{
   printf("q.w = %d \n", q.w);
   q.w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(z);
   printf("z.w = %d\n", z.w);
}

acad/cs101/examples/notes ==> gcc -o p41 p41.c
acad/cs101/examples/notes ==> p41
q.w = 99
z.w = 99
acad/cs101/examples/notes ==> cat p41.c
```

The disadvantages of passing structures by value are that copying large structures onto the stack

- is very inefficient and
- may even cause program failure due to stack overflow.

```
   struct work_type
   {
      int w[1024 * 1024];
   };

/* This fellow will cause 4 Terabytes to be copied */
/* onto the stack.                                 */

   struct work_type fourMB;
   for (i = 0; i < 1000000; i++)
      slow_call(fourMB);
```

Passing pointers to structures

Passing a pointer requires that only a single word by pushed on the stack *regardless of how large the structure is.*

Furthermore, the called function can now modify the structure!

```
/* p42.c */

struct work_type
{
   int w;
};

void fun(struct work_type *q)
{
   printf("q->w = %d \n", q->w);
   q->w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(&z);
   printf("z.w = %d\n", z.w);
}

acad/cs101/examples/notes ==> gcc -o p42 p42.c
acad/cs101/examples/notes ==> p42
q->w = 99
z.w = 1000
```

But what if you *don't want* the recepient to be able to modify the structure? Then the *const* qualifier can be prefixed to the parameter.

```
/* p43.c */

struct work_type
{
   int w;
};

void fun(const struct work_type *q)
{
   printf("q->w = %d \n", q->w);
   q->w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(&z);
   printf("z.w = %d\n", z.w);
}
```

```
acad/cs101/examples/notes ==> gcc -g p43.c
p43.c: In function `fun':
p43.c:11: error: assignment of read-only member `w'
```

This form of protection is not especially strong and is easily bypassed

```
/* p44.c */

struct work_type
{
   int w;
};

void fun(const struct work_type *q)
{
   int *bypass = (int *)q;
   printf("q->w = %d \n", q->w);
   *bypass = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(&z);
   printf("z.w = %d\n", z.w);
}
```

```
acad/cs101/examples/notes ==> gcc -g p44.c -o p44
acad/cs101/examples/notes ==> p44
q->w = 99
z.w = 1000
```

Structures as return values from functions

Scalar values (*ints, floats, etc)* are efficiently returned in CPU registers
Historically, the structure assignments and the return of structures was not supported in C.
But the return of *pointers* including *pointers* to structures has always been supported.

```
/* p31.c */

struct work_type
{
   int w;
};

struct work_type *fun(void)
{
   struct work_type work;

   work.w = 99;
   return(&work);
}

int main()
{
   struct work_type *q;

   q = fun();
   printf("%d \n", q->w);
}
```

```
acad/cs101/examples/notes ==> gcc -o p31 p31.c
p31.c: In function `fun':
p31.c:13: warning: function returns address of local
variable
acad/cs101/examples/notes ==> p31
99
```

The reason for the warning is that the function is returning a pointer to a variable that was allocated on the stack during execution of the function. Such variables are subject to being wiped out by subsequent function calls.

This example shows this can indeed occur.

```
/* p32.c */

struct work_type
{
   int w;
};
struct work_type *fun(void)
{
   struct work_type work;

   work.w = 99;
   return(&work);
}

int fun2(int v)
{
  int w;
  int x;
  w = v;
  x = w * v;
  return(x);
}

int main()
{
   struct work_type *q;
   int z = 12;
   q = fun();
   z = fun2(z);
   printf("%d \n", q->w);
}
```

```
acad/cs101/examples/notes ==> gcc -o p32 p32.c
p32.c: In function `fun':
p32.c:13: warning: function returns address of local variable
acad/cs101/examples/notes ==> p32
-1077017368
```

Return of structures

It is possible for a function to return a structure. This facility depends upon the structure assignment mechanisms which copies one complete structure to another.

- This avoids the unsafe condition associated with returning a pointer but
- incurs the possibly extreme penalty of copying a very large structure

```
/* p33.c */

struct work_type
{
   int w;
};

struct work_type fun(void)
{
   struct work_type work;

   work.w = 99;
   return(work);
}

int main()
{
   struct work_type q;
   struct work_type z;
   q = fun();
   z = q;
   printf("%d %d \n", q.w, z.w);
}
```

```
acad/cs101/examples/notes ==> gcc -g -o p33 p33.c
acad/cs101/examples/notes ==> p33
99 99
acad/cs101/examples/notes ==>
```

Summary

Passing/returning instances of structures potentially incurs big overhead
Passing/returning pointers incurs almost no overhead
Accidental modifications can be prevented with *const*

- Therefore, it is recommended that you *never pass nor return* an instance of a structure unless you have a very good reason for doing so.

This problem doesn't arise with arrays.

- The only way to pass an array by value in the C language is to embed it in a structure!!
- The only way to return an array is to embed it in a structure

The *typedef* facility

Can be used to create new names for existing types.

Syntax is:

typedef existing-type new-type;

```
typedef unsigned char uc8;

typedef
struct pix_type
{
   uc8 r;
   uc8 g;
   uc8 b;
}  pix_t;

pix_t pixel;
pix_t *pixptr;
```

Useful for creating

shorthand notation

writing code that is (somewhat) insensitive to variations in the *sizeof short/long/int*

use

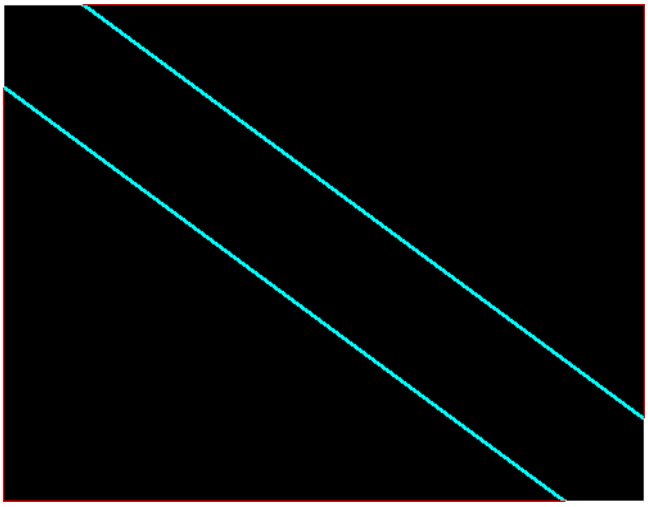
i32 x;

in declarations and then use *typedef* to equate *i32* to a native 32 bit integer type.

An edge detecting filter

In the previous assignment we constructed a smoothing filter. We will now address the problem of identifying *edges* within an image. An edge maybe informally defined as a region of a region in which there is a rapid change of gray level or color.

For example in the dive flag there are two diagonal edges: where the image changes from red to white and where it changes from white back to red. The filter that we will build will show bright colors only where edges occur. Application of the filter we will build yields the following:



Notes on the resulting image:

- The edge detection filter replaces each pixel with a function of the *difference* between neigboring pixels.
- As with the smoothing filter the value of the pixel being replaced is *irrelevant*
- The image is black (as it should be) in areas of constant color.
- The border of the image is preserved just as with the smoothing filter.
- The blue-green color of the edges arises as follows

    - The white pixels have the value (250, 250, 250)
    - The red pixels have the value (255, 0, 0)
    - Thus at the edge there is a *big* change in *blue* and *green* components
    - But there is not a big change in the *red* component.

The Sobel filter

This filter is designed to replace a pixel with a value that represents a weighted average of the differences in its neighbor pixels

Suppose three adjacent pixels in the same row of a grayscale image have the value

(64, 128, 192)

There is clearly a big jump between the left and right pixels.
The magnitude of the jump is 192-64 = 128

Now suppose three adjacent pixels in the same row have the value

(120, 128, 136)

Here the difference is only 16 which might just be caused by an area of gradually increasing brightness.

Complicating the problem is that the edge may be

horizontal
vertical
diagonal (as with the dive flag)
or have orientation somewhere in between

Sobel_x and Sobel_y

Because of the orientation issues the filter we will use is a cleverly weighted average of two filters.

The X-direction filter is:

-1	0	1
-2	0	2
-1	0	1

The filter is applied to a 3 x 3 region centered at each pixel of the image just like the smoothing filter was. For a grayscale image the value of *soble_x* is the sum of the inner products of the 3 rows of the filters with the three rows of the image. For a color image *soble_x* produces *(r, g, b)* values, where each component (r, g, b) is the sum of the inner products of the 3 rows of the filters with each component of the three rows of the image.

The Y-direction filter is:

-1	-2	-1
0	0	0
1	2	1

Its mission is generally the same as the X-direction but its objective is to identify differences in the Y-direction.

Application of the X-direction filter

In this example we see a situation in which there is an edge between the center and rightmost pixels of the 3 x 3 block. Since the edge is decreasing in brightness, the filter produces a value of -512.

This value is both

- negative and
- exceeds 255 in absolute value
- so we will have to deal with that later

Filter			3x3			Pairwise Products			Sum
-1	0	1	128	128	0	-128	0	0	-512
-2	0	2	128	128	0	-256	0	0	
-1	0	1	128	128	0	-128	0	0	

Here we see the same sobel_x filter applied to a different 3 x 3 area of the image. This shows that for a single pixel edge in the input image we should expect a “double wide” line in the output.

-1	0	1	128	0	0	-128	0	0	-512
-2	0	2	128	0	0	-256	0	0	
-1	0	1	128	0	0	-128	0	0	

Application of the y-direction filter

The y-direction filter is designed to detect differences in the *y* direction. When it is applied to the 3x3 pixel array from the previous page the result is -128 + 0 + 128 = 0 as it should be.

Filter			3x3			Pairwise Products			Sum
-1	-2	-1	128	0	0	-128	0	0	0
0	0	0	128	0	0	0	0	0	
1	2	1	128	0	0	128	0	0	

But when the y direction filter is applied to data which varies in the *y* direction the variation is evident.

-1	-2	-1	128	128	128	-128	-256	-128	-512
0	0	0	0	0	0	0	0	0	
1	2	1	0	0	0	0	0	0	

When the data varies diagonally both the X and Y filters return non-0 values

-1	0	1	128	128	128	-128	0	128	384
-2	0	2	0	128	128	0	0	256	
-1	0	1	0	0	128	0	0	128	
-1	-2	-1	128	128	128	-128	-256	-128	-384
0	0	0	0	128	128	0	0	0	
1	2	1	0	0	128	0	0	128	

Combining the results of the X and Y filters

It should be clear from the above example that we can't just add them up. (If we add 384 – 384) we get 0). Instead we treat the (x, y) values computed by the Soble_x and Soble_y filters as a vector and compute the length of the vector in the usual way by taking the square root of the sum of the squares.

|| (x, y) || = sqrt(x * x + y * y)

For a grayscale image this is (almost) the value of the output pixel.

The final complication is that in the presence of a very well defined edge the filter my compute a value greater than 255 (which is the maximum pixel value).

For example:

|| (384, -384) || = 543

Therefore the last step in the operation must be to see if the value is > than 255 and if so set the the value to 255. This operation is known as *clamping*.

Application to color images

Because of the fact that the filter can produce values larger than 255 it is necessary that the sums be accumulated in an *int* or *float* entity. Furthermore since we are now dealing with a color image we must compute *separate* sums for the *r, g,* and *b* components.

```
void make_pix(
int row,
int col)
{
   int i, j;

   float sum_dx[3] = {0, 0, 0};  // rgb sums for soble_x
   float sum_dy[3] = {0, 0, 0};  // rgb sums for soble_y
   float sum[3]    = {0, 0, 0};  // final rgb sums that
                    // will be assigned to output pixel
```

The inner product function is then modified to compute the vector of sums.

```
void inner_prod(
struct pix_type *pix,
float *filt,
float *sum)
{
   int   i;

   for (i = 0; i < 3; i++)
   {
      sum[0] += pix->r * (*filt);
      sum[1] += pix->g * (*filt);
      sum[2] += pix->b * (*filt);
      pix = pix + 1;
      filt = filt + 1;
   }
}
```

When the filter is applied to an image containing more information than the dive flag...

Interesting effects are produced.