QUICK IDL TUTORIAL NUMBER TWO: IDL DATATYPES AND ORGANIZATIONAL STRUCTURES
February 10, 2003

By the term *datatype* we mean, for example, integers, floating point variables, strings, complex numbers. By the term *organizational structures* we mean scalars, vectors, arrays, and structures. We cover these in the following two sections. All available datatypes can be arranged in all available organizational structures. For example, we can have arrays of strings, vectors of complex numbers.

1. DATATYPES

Here we cover only the basic IDL datatypes. There are others, including unsigned integers and complex numbers. In the last section here, we discuss structures; these are useful, almost essential, for databases such as you might generate from a series of observations.

1.1. DIGITS, BITS, BYTES, AND WORDS

We have gotten to the place where you need to know a little about the internal workings of computers. Specifically, how the computer stores numbers and characters.

Humans think of numbers expressed in powers-of-ten, or *decimal numbers*. This means that there are 10 digits ($0 \rightarrow 9$) and you begin counting with these digits. When you reach the highest number expressible by a single digit, you use two digits and generate the next series of numbers, $10 \rightarrow 99$. Let f and s be the *first* and *second* digits, respectively; then the number is $10^f + s$. And so on with more digits.

Fundamentally, all computer information is stored in the form of *binary numbers*, meaning powers-of-two. How many digits? Two! They are 0 and 1. The highest number expressible by a single digit is 1. The two-digit numbers range from 10 to 11; the number is $2^f + s$. And so on with more digits. But wait a minute! The word "digit" is a misnomer—it implies something about 10 fingers. Here it's the word **bit** that counts. Each binary "digit" is really a **bit**. So the binary number 1001 is a 4-bit number. What decimal number does the binary number 1001 equal?

For convenience, computers and their programmers group the bits into groups of eight. Each group of 8 bits is called a **byte**. Consider, then, the binary number 11111111; it's the maximum-sized number that can be stored in a byte. What is this number?

Finally, computers group the bytes into **words**. The oldest PC's dealt with 8-bit words—one byte. The Pentiums and Sparcs deal with 32-bit words—four bytes. What's the largest number you can store in a 4-byte word? And how about negative numbers?

Below we describe how IDL (and everybody else) gets around this apparent upper limit on numbers. They do this by defining different data types. Up to now, the details didn't matter much. But now...We don't cover all datatypes below—specifically, we omit Complex (yes, complex numbers!), Hexadecimal, Octal, and Structure datatypes, which you can look up if you are interested.

1.2. INTEGER DATATYPES IN IDL

Integer datatypes store the numbers just like you'd expect. IDL supports integers of four different lengths: 1, 2, 4, and 8 bytes. The shorter the word, the less memory required; the longer the word, the larger the numbers can be. Different requirements require different compromises.

1.2.1. 1 byte: The Byte Datatype

The Byte datatype is a single byte long and always positive. Therefore, its values run $0 \to 255$. Images are always represented in bytes. The *data* might not be in bytes, but the numbers that the computer sends to the video processor card are always bytes. Video screens require lots of memory and really quick processing speed, so bytes are ideal. You generate an array using **bindgen**; you can generate a single byte variable by saying **x=3b**. If, during a calculation, a byte number exceeds 255, then it will "wrap around"; for example, 256 wraps to 0, 257 to 1, etc.

1.2.2. 2 bytes: Integers and Unsigned Integers

With 2 bytes, numbers that are always positive are called **Unsigned Integers**. They can range from $0 \to 256^2 - 1$, or $0 \to 65535$. You generate an array using **uindgen**. How do you think unsigned integers wrap around?

Normally you want the possibility of negative numbers and you use **Integers**. The total number of integer values is $256^2 = 32768$. One possible value is, of course, zero. So the number of negative and positive values differ by one. The choice is to favor negative numbers, so Integers cover the range $-32768 \to 32767$. You generate an array using **indgen**. What happens with wraparound? What if **x=5**, **y=30000** and **z=x*y**? Check it out!

1.2.3. 4 bytes: Long Integers and Unsigned Long Integers

The discussion here is exactly like that for 2-byte integers, except that 256^2 becomes 256^4. What are the limits on these numbers? See IDL help under "Data Types" and "Integer Constants" for more information. You generate arrays using **ulindgen** and **lindgen**.

1.2.4.   8 bytes: 64-bit Long Integers and Unsigned 64-bit Long Integers

The discussion here is exactly like that for 2-byte integers, except that 256^2 becomes 256^8. What are the limits on these numbers? See IDL help under "Data Types" and "Integer Constants" for more information. You generate arrays using **ul64indgen** and **l64indgen**.

1.3.   FLOATING DATATYPES IN IDL

The problem with integer datatypes is that you can't represent anything other than integral numbers—no fractions! Moreover, if you divide two integer numbers and the result should fractional, but it won't be; instead, it will be rounded down (e.g. $\frac{5}{3}$ is calculated as 1). To get around this, the *floating* datatype uses some of the bits to store an *exponent*, which may be positive or negative. You throw away some of the precision of the integer representation in favor of being able to represent a much wider range of numbers.

1.3.1.   4 bytes: Floats

"Floating point" means floating decimal point—it can wash all around. With Floats, the exponent can range from about $-38 \rightarrow +38$ and there is about 6 digits of precision. You generate an array using **findgen** and a single variable by including a decimal point (**x=3.**) or using exponential notation (**x=3e5**).

1.3.2.   8 bytes: Double-Precision

Like Float, but the exponent can range from about $-307 \rightarrow +307$ and there is about 16 digits of precision. You generate an array using **dindgen** and a single variable by writing **x=3d** or **x=3d5**.

1.4.   STRINGS

Strings store characters—letters, symbols, and numbers (but numbers as *characters*—you can't calculate with strings! A string constant such as **hello** consists of five letters. It takes 5 bytes to store this constant—one byte for each character. There are 256 possible characters for each of the bytes; with 2*26 letters (smalls and caps) and 10 digits, this leaves 104 other possibilities, which are used for things like semicolons and periods. You can generate an array of strings with **strarr** and a single string with **x = 'Hi there!!!'**.

2. ORGANIZATIONAL STRUCTURES

2.1. SCALARS

A scalar is just a single number. For example, a string scalar is **joename= 'joe'**.

2.2. VECTORS

A vector is a one-dimensional array. For example, a three-element vector of names is **threenames = ['joe', 'ivan', 'mark']**.

2.3. ARRAYS

IDL handles arrays up to 8 dimensions, i.e. with 8 subscripts. Arrays with two subscripts can be mathematically treated a matrices using the # and ## operators, and various matrix manipulation routines; see IDL help under **matrices** and **matrix operators**. You create vectors and arrays using, for example, the **fltarr** or **findgen** commands (for floating point numbers; equivalent commands exists for all variable types). You populate them as appropriate, but try to avoid using for loops; instead, use **where**, appropriate use of the * operator, etc.

IDL provides a great deal of flexibility in using subscripts to address particular array elements, and this flexibility is what makes IDL so useful. For example, consider a two-dimensional array **a=findgen(100,100)**. Then:

```
b = a[ 23:25, 67:69]
```

makes **b** a 3×3 2-d array equal to a's array elements in the little box specified. The combination

```
indx = where( a gt 10.)
b = a[ indx]
```

makes **b** a 1-d array equal to the elements of **a** that are larger than 10. The combination

```
indx = where( a gt 10.)
jndx = where( a[ indx] le 100.)
b = a[ indx[ jndx]]
```

shows that you can subscript arrays with other arrays, and makes **b** equal to a 1-d array equal to the elements of **a** that are both larger than 10 and less than or equal to 100.

2.4.   STRUCTURES

Structures are immensely useful for any project in which data of different types are related. For example, if you have a catalog of stars with positions and reddenings, you can put the whole catalog in a structure array in which each element of the array contains many quantities such as the name and position. And you can have arrays of structures. Structures allow you to create and customize your own data base. Having done this, using the **where** command allows you flexible access to anything with a one-line command.

We refer you to Chapter 7 of *Building IDL Applications* for a complete discussion of structures. Here we provide a quick example. For our star catalog, define the structure

```
{A = {star, name: 'alpha ori', ra:5.3345, dec:-7.6568,reddening:fltarr(12)}
```

Now if you type **help,/struct, a** you will see on the screen

```
** Structure STAR, 4 tags, length=64:
   NAME            STRING    ''
   RA              FLOAT         0.00000
   DEC             FLOAT         0.00000
   REDDENING       FLOAT     Array[12]
```

This says that the structure A is a type defined as *star*, and it has four fields, the name the two positions, and 12 different measurements of reddening. You could populate the 12 reddening measurements by typing, for example,

```
a.reddening = [1.2, 1.4, 1.3, 1.6, 1.3, 1.4, 1.3, 1.3, 1.6, 1.3, 1.4, 1.3,]
```

and you could type

```
print, a.dec
```

to find the declination, and you could change things by typing

```
a.dec = 5.5
a.reddening[3] = 0.7
```

This example is a **named structure**, which means that all you cannot change the contents of this structure after having defined it. You can also create an **anonymous structure**, without a name:

```
b = {name: 'alpha ori', ra:5.3345, dec:-7.6568, reddening:fltarr(12)
```

Now if you type **help, b, /str** you see

```
** Structure <cb0d0>, 4 tags, length=64, refs=1:
   NAME            STRING    'alpha ori'
   RA              FLOAT          5.33450
   DEC             FLOAT         -7.65680
   REDDENING       FLOAT     Array[12]
```

and it has no given name. The contents of anonymous structures can be changed after they've been defined.

If this is all there were to it, then structures wouldn't be very useful because you have observed 585 stars, say, and you'd need a separate structure for each. But you can create arrays of structures, e.g.

```
cataloga = replicate( {star}, 585) [OR cataloga = replicate( a, 585) ]
catalogb = replicate( b, 585)
```

creates structure arrays of 565 elements for **a** and **b**. You can to them with a subscript, for example you could write

```
cataloga[ 3] = a
```

to set the third element of the structure array equal to **a**. Or you could do it element-by-element, for example

```
cataloga[ 3].name = 'alpha ori'
```

Now you can print the star name of the third element by typing

```
print, cataloga[ 3].name
```

or, less conventionally. . .

```
print, (cataloga.name)[3]
```

Try using structures when taking data for your experiments. You'll grow to love them!