MPI DERIVED DATATYPES

- In virtually all distributed-memory systems, communication can be much more expensive than local computation. For example, sending a ***double*** from one node to another will take far longer than adding two doubles stored in the local memory of a node.

- The cost of sending a fixed amount of data in multiple messages
  is usually much greater than the cost of sending a single message with the same
  amount of data. For example, we would expect the following pair of for loops to be much slower than the single send/receive pair:

```
double x[1000];
. . .
if (my_rank == 0)
        for (i = 0; i < 1000; i++)
```

```
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);
```

```
if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

- In fact, on one of our systems, the code with the loops of sends and receives takes nearly 50 times longer. On another system, the code with the loops takes more than 100 times longer. Thus, if we can reduce the total number of messages we send, we're likely to improve the performance of our programs.

- In MPI, a ***derived datatype*** can be used to represent any collection of data items
  in memory by storing both the types of the items and their relative locations in memory.
- If a function that sends data knows the types and the relative locations in memory of a collection of data items, it can collect the items from memory before they are sent.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.

- Formally, a ***derived datatype*** consists of a sequence of basic MPI datatypes together with a displacement for each of the datatypes. In our trapezoidal rule example, suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses.

Variable	Address
a	24
b	40
n	48

- Then the following derived datatype could represent these data items: $\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$.
- The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the beginning of the type. We've assumed that the type begins with ***a***, so it has displacement 0, and the other elements have displacements measured, in bytes, from a: b is 40-24= 16 bytes beyond the start of a, and n is 48-24= 24 bytes beyond the start of ***a***.
- We can use MPI_Type_ create_ struct to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
      int             count                          /* in   */,
      int             array_of_blocklengths[]        /* in   */,
      MPI_Aint        array_of_displacements[]       /* in   */,
      MPI_Datatype    array_of_types[]               /* in   */,
      MPI_Datatype*   new_type_p                     /* out  */);
```

- The argument ***count*** is the number of elements in the datatype, so for our example, it should be three. Each of the array arguments should have ***count*** elements.
- The first array, ***array_ of_ block_lengths***, allows for the possibility that the individual data items might be arrays or subarrays. If, for example, the first element were an array containing five elements, we would have

  array_ of_ block_lengths[0] =5

  However, in our case, none of the elements is an array, so we can simply define

```
int array_of_blocklengths[3] = {1, 1, 1};
```

- The third argument to ***MPI_Type_create_ struct***, ***array_of_displacements***, specifies the displacements, in bytes, from the start of the message. So we want

```
array_of_displacements[] = {0, 16, 24};
```

  To find these values, we can use the function ***MPI_Get_ address***:

```
int MPI_Get_address(
      void*      location_p  /* in  */,
      MPI_Aint*  address_p   /* out */);
```

- It returns the address of the memory location referenced by ***location_ p***. The special type ***MPI _Aint*** is an integer type that is big enough to store an address on the system. Thus, in order to get the values in array of displacements, we can use the following code:

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

- The ***array_ of _datatypes*** should store the MPI datatypes of the elements, so we can just define

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

`

With these initializations, we can build the new datatype with the call

```
MPI_Datatype input_mpi_t;
. . .
MPI_Type_create_struct(3, array_of_blocklengths,
      array_of_displacements, array_of_types,
      &input_mpi_t);
```

Before we can use `input_mpi_t` in a communication function, we must first **commit** it with a call to

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

This allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

Now, in order to use `new_mpi_t`, we make the following call to `MPI_Bcast` on each process:

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

So we can use `input_mpi_t` just as we would use one of the basic MPI datatypes.

In constructing the new datatype, it's likely that the MPI implementation had to allocate additional storage internally. Therefore, when we're through using the new type, we can free any additional storage used with a call to

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```