Topic 8 – Data Structures

STRUCTURING DATA

- Arrays are extremely powerful constructs, and allow you to solve problems that otherwise would be infeasible.
- However, they aren't appropriate in every circumstance.

- If you store many instances of a single value, arrays work well.
  - For example, if you want to store 100 exam marks in order to process them.

- However, if you want to store many instances of multiple values, arrays aren't well suited.
  - For example, if you want to store 100 exam marks along with the names of the students that scored each result.

- To solve these problems involving many different possible arrangements of data, you need to design your own data type.
- When you bundled together different values to design such a solution, these are called *data structures*.
- This essentially involves designing your own data type.

USER-DEFINED TYPES

- A first step towards being able to create your own data types is the ability that C provides to give user-defined names to any defined data type with the `typedef` statement.

- For example, say we want to define a counter variable type we might say:

```
typedef int ctr;
```

- Now we can define variables of type `ctr` as if it were a built-in type:

```
ctr i;

...

for(i = 0; i < MAX; i++)
...
```

- The advantage of this is again abstraction because it hides details like the exact type of the data while simultaneously telling the reader what the variable's purpose is.

- However, simply re-naming data types is a trivial use of the `typedef` statement and there are much more important uses.

Creating Data Structures

- One common way that new data types are created in C is through something called a `struct`.
- This is essentially an abbreviation of the term *data structure* and is created by combining simple data types.

- While an array allows you to bundle together multiple pieces of data into one variable, a data structure represents a new *data type* that you can define.
- From the new data type, you can create your own variables of this type.

- Say we are writing a program to keep track of the stock items held at a factory or warehouse.
- The program is to interoperate with a database that stores further details about the stock items (like where to order more from).
- However, the program's job is primarily to track stock numbers.
- If the number of items drops below a specific level, the program will flag a warning.
- Also, if the value of the stock of a particular item exceeds a fixed amount then the program will also flag a warning to prevent more stock being purchased.

- This program will probably need to store the following information:
  - A short name describing the stock item.
  - The number of each item currently in stock.
  - The low watermark for this stock item (after which more stock must be purchased because it's running out).
  - The value of the item.

- Here is a C struct to store all this data for each bin:

```
const int NAME_SIZE = 50;

struct item
{
   char name[NAME_SIZE];
   int stock;
   int lwm_stock;
   float value;
};
```

- This is a new complex data type called `struct item` and it is made up of the four different variables.

Things to note:

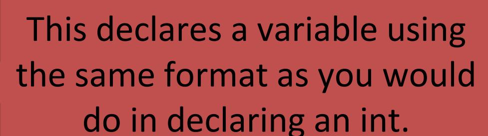
- The semi-colon after the right curly bracket! Everyone forgets this because it is the only time in C that it is done.
- This structure definition creates a new data type so you can declare variables of this type.

- However, the name of the data type is `struct item` and *not* simply `item`.
- So to declare variables of this type you need to write:

```
struct item myvar;
```

- Because this looks ugly, it is common to use typedef to define a custom type name:

```
typedef struct item item;
...
item myvar;
```



Using Structures

- Now we can create our own data structures, how do we go about using them?

- Essentially while the `struct` can be dealt with as a whole, many operations can only be performed on the fields that make it up.
  - This is because these fields are primitive data types, which can be processed by the language.

- To access each field we use the following syntax:

  variable.field

- So if we have a variable called `firstbin` of type `item` from before, for the fields that are simple data types we can use the following assignment statements to put values into the `struct`:

```
firstbin.stock = 10;
firstbin.lwm_stock = 2;
firstbin.value = 29.95;
```

For the field name that is a string, instead we could do something like:

```
if(NAME_SIZE >= (strlen(“iPad Air”) + 1))
   strcpy(firstbin.name, “iPad Air”);
else
   printf("Not enough room to copy!\n");
```

- Note that when using the *field selector notation* (`.`), the expression has the type of the field being referenced.
- For example, `firstbin.stock` behaves exactly like an ordinary integer.

Parameter Passing and Complex Data Types

- In general the same rules apply when passing complex data types into functions.
- One important principle though is that complex types should *always* be passed by reference.
  - **This applies even when they are not to be modified.**

- The reason for this is that complex data types take up a lot more memory space than primitive types (for obvious reasons).
  - Although we have used a simple example that is isn't huge, they can become *much* bigger.

- Making a copy of the entire complex data type (as would be done if passed by value) is therefore often inefficient, both in terms of the space and the computational work required.
- So as a general rule they should always be passed by reference.

- Another important point to remember is that functions cannot return complex data types – they can only be passed back as parameters.
  - Only simple data types can be the return values of functions.

- Finally remember that the individual fields that make up a `struct` can be passed into a function separately, just as variables of primitive data types can be normally.

Here is a program demonstrating the use of `struct` and parameter passing:

```
#include <stdio.h>
#include <string.h>

  /* This needs to be declared here so that
  the rest of the program knows about
 * this new type.
 */

const int NAME_SIZE = 50;

struct item
{
   char name[NAME_SIZE];
   int stock;
   int lwm_stock;
   float value;
};

typedef struct item item;


void FillBin(item &bin1)
{
   printf("Enter bin name: ");
   fgets(bin1.name, NAME_SIZE,stdin);
   bin1.name[strlen(bin1.name) - 1] = '\0';

   printf("Enter stock quantity: ");
   scanf("%d%*c", &bin1.stock);

   printf("Enter stock low water mark: ");
   scanf("%d%*c", &bin1.lwm_stock);

   printf("Enter value of stock: ");
   scanf("%f%*c", &bin1.value);
```

```
    return;
}

void PrintBin(item &bin)
{
    printf("Bin name: %s\n", bin.name);
    printf("Stock: %d\n", bin.stock);
    printf("Low Watermark Stock: %d\n",
                                bin.lwm_stock);
    printf("Value of item: $%.2f\n",
                                bin.value);

    return;
}


int main()
{
    item firstbin;

    FillBin(firstbin);

    printf("\n\n");

    PrintBin(firstbin);

    return(0);
}
```

NB: Pass by reference.

Combining Arrays and structs

- The fields in a struct can be made up of any data type that exists at that point.
- As with the last example, this can include other complex data types such as arrays/strings, and even other structs.
    - The important point to remember is that when accessed using the field selector operator (.), a field in a struct behaves exactly the same as an ordinary variable of that type.

- However, you can also combine arrays with structs the other way by having an array of structs.
- In the example before we declared only a single variable of type `item`, so we can only store information about one stock item.
- More realistically there would probably be many such items stored in the factory.
- Instead of defining individual struct variables we could instead just create an array.

- Creating an array of structs is not very difficult, once the struct type itself has been declared.
- For example, the following code creates an array of 20 items stored in the factory's bins:

```
const int MAX_ITEMS = 20;

…

item bins[MAX_ITEMS];
```

- When working with the array of structs, the same principle as always applies:

> *Reduce whatever structure you're working with down to a simple type and then process this as normal.*

- For example, the field `stock` within each element of the `bins` array created above is just an int.
- To access this integer, just use the `[]` to access the array element you want and then the field selector (`.`) operator to access the individual field.
- Therefore, to access the `stock` field in the first element of the `bins` array, you could write something like:

```
bins[0].stock = 100;
```

- When working with a field that is itself an array, the expression becomes more complex but the same principle applies.
- For example, to read in the name of whatever is in the first bin (first element of the `bins` array), you could write something like:

```
fgets(bins[0].name, NAME_SIZE,stdin);
bins[0].name[strlen(bins[0].name) - 1] = '\0';
```

- Work through this expression slowly until you can make sense of it!

- Of course, as always when working with arrays, you usually want to use a for loop to process all of the elements in the array.
- The following program demonstrates all of this.

```
#include <stdio.h>
#include <string.h>

const int NAME_SIZE = 50;
const int MAX_ITEMS = 3;

/* This needs to be declared here so that the rest of
the program knows about
 * this new type.
 */


struct item
{
     char name[NAME_SIZE];
     int stock;
     int lwm_stock;
     float value;
};

typedef struct item item;


void FillBin(item bins[], int numbins)
{
     int i;

     for(i = 0; i < numbins; i++)
     {
          printf("--- Enter details for bin #%d ---\n",
                                                  i+1);

          printf("Enter bin name: ");
          fgets(bins[i].name, NAME_SIZE, stdin);
          bins[i].name[strlen(bins[i].name) - 1] = '\0';

          printf("Enter stock quantity: ");
          scanf("%d%*c", &bins[i].stock);

          printf("Enter stock low water mark: ");
          scanf("%d%*c", &bins[i].lwm_stock);

          printf("Enter value of stock: ");
          scanf("%f%*c", &bins[i].value);
```

A small number is used here to make the program quick to demonstrate.

Arrays are always passed by reference automatically; no & required.

```
        printf("--------------------------------\n\n");
    }

    return;
}

void PrintBin(item bins[], int numbins)
{
    int i;

    for(i = 0; i < numbins; i++)
    {
        printf("--- Details for bin #%d ---\n", i+1);

        printf("Bin name: %s\n", bins[i].name);
        printf("Stock: %d\n", bins[i].stock);
        printf("Low Watermark Stock: %d\n",
                                  bins[i].lwm_stock);
        printf("Value of item: $%.2f\n",
                                     bins[i].value);
        printf("--------------------------------\n\n");
    }
    return;
}


int main()
{
    item bins[MAX_ITEMS];

    FillBin(bins, MAX_ITEMS);

    printf("\n\n");

    PrintBin(bins, MAX_ITEMS);

    return(0);
}
```