CS 162C++
Data Structures (structs)

A data structure is a way to group multiple individual data elements into a single item. Once you define a structure, you can then create multiple instances of it using variable names in the same way you would create multiple integers or any other simple data type. You can also pass a structure to functions, create a pointer to a structure, and allocate them dynamically on the heap.

Why Bother

Sometimes you have multiple data items that are related, for example a move in a game that consists of a row and a column, or a card that has a suit and a rank, or a geometric shape that has a width, length, or diameter. By creating a structure, you can group these individual items together and treat them as a single item.

Defining a struct

You define a struct and give the definition a name before you use it in the program. For this reason, they are often defined at the beginning a program file or in a header file that is included in multiple program files.

The definition of a struct consists of the keyword **`struct`**, the name you are giving to this type, a body that defines what is included in the struct, and optionally instances of this struct.

For example:

```
struct t_move {
   int row;
   int col;
};
```

In this example the name of the structure is **`t_move`**. I normally use names starting with **`t_`** to indicate that I am defining a data type, not creating a variable.

The body is **`{ int row; int col; }.`** This tells the compiler what will be included in each instance of this structure. In this case, there will be space allocated for two integers.

In this example, there were no instances of **`t_move`** actually created. All we did was define what a **`t_move`** would look like if we created one.

To create a variable of type **`t_move`**, you do it exactly was you would create a variable of type **`integer`** or **`char`** or …

```
t_move playerMove;
```

You could have optionally defined **`playerMove`** in the same statement as you defined the structure:

```
struct t_move {
   int row;
   int col;
} playerMove;
```

Normally you would do this when you define the struct inside a program, but not when you define it outside of a program or in a header file; since that would be creating a global variable and you do not want to create or use global variables.

Using a struct

As shown above, you define an instance of a struct in the same way that you define an instance of any other variable type.

```
t_move playerMove;
```

This allocates memory for the data, says that the data in this location is of type **t_move** and gives it the name **playerMove**.

You then access the contents of this memory using a combination of the ***variable name*** **playerMove** and the two ***member names***, **row** and **col**, ***using dot notation***.

```
playerMove.row = 12;
playerMove.col = 22;
```

Passing and returning structs

You can pass a struct as a parameter and return it from a function in the same way you would do with any other data type:

```
t_move getMove();

void updateBoard(char board[], t_move aMove);
```

Or you can pass by reference:

```
void getMove(t_move &theMove);
```

Example

For a program that works with rectangles, passing their dimensions with a struct

```
#include <iostream>

using namespace std;

    // global definition of rectangle for later use
struct t_rect {
    int width;
    int length;
};

    // function declarations, definitions below
t_rect getRectangle();
void showArea(t_rect aRect);

    // main program, does example work
int main()
{
    t_rect aRectangle = getRectangle();

    showArea(aRectangle);

    return 0;
}

    // get the length and width of a rectangle
t_rect getRectangle()
{
    t_rect data;
    cout << "Enter width: ";
    cin >> data.width;
    cout << "Enter length: ";
    cin >> data.length;

    return data;
}

    // display the area of the passed rectangle
void showArea(t_rect aRect)
{
    int area = aRect.width * aRect.length;

    cout << "The area is " << area << endl;
}
```