Notes on Structs

Most programming languages have a feature to create composite data types: data types that represent collections of other data types. Such data types are useful in situations where a single data type is not expressive enough to represent a piece of information needed in a programming language. For instance, a program may need to store information about points in the Cartesian plane. A single point in the plane has two components: an x-coordinate and a y-coordinate. Without composite types, storing each point needed in the program would require keeping track of two separate `int` variables (or longs/floats/doubles) for each point. Records allow the programmer to create a new point data type that is a combination of two int data types. This not only saves typing and makes passing and returning points to and from functions easier, but is good programming design because aggregate types serve as abstractions: they allow us to give meaningful names to concepts represented by collections of data types in our programs.

In C++, there are two similar mechanisms for creating records: structs and classes. Though in modern C++ both are equally powerful, traditionally structs (which come from the earlier language C) are used for simple combinations of data types, while classes are used for object-oriented programming.

Defining a struct

Structs are defined with the struct keyword. Structs, like functions, must be defined before they are used, so typically they are placed at the top of a program, usually immediately after the `#include` lines and the `using namespace std` line. Structs can also be defined in separate files and `#include`'d.

The syntax to define a struct is:

```
struct name_of_struct
{
  datatype field_name;
  datatype field_name;
  datatype field_name;
  (etc)
};
```

It is a common mistake to leave off the semicolon at the end of the struct definition block.

For instance, one could create a `point` struct like the one described earlier as follows:

```
struct point
{
  int x;
  int y;
};
```

If a struct uses multiple variables of the same type, they may be combined on one line:

```
struct point
{
  int x, y;
};
```

Both of the struct definitions above tell the C++ compiler to create a new data type called `point`. This data type is a combination of two other data types, an integer called `x` and an integer called `y`. The individual variables within a struct definition are called members or fields.

Using a struct

After a struct is defined, it can be used in a program just like any other data type (such as `int`, `double`, or `char`) would be used: the programmer can define a new variable with a struct data type, pass variables of a struct type to functions, and have functions return a variable of a struct type. A struct can even be defined to contain fields with another struct type. (For instance, a program that needed to store information about a mouse stuck in a maze might choose to define a struct that containing a field for the mouse's name (a `string`), and their location in the maze (a `point`):

```
struct mouse
{
  string name;
  point location;
};
```

Note that the word `struct` by itself is *not* a data type. In the paragraphs above, when we refer to a "struct data type," we mean a *specific* struct type, such as `point` or `mouse`. In other words, an appropriate response to the question "What data type is this variable?" is never "It's a struct" --- or at least that's not the complete answer. Instead, say "It's a point" or "It's a point struct."

Declaring a new variable of a struct type

To declare a new variable of a struct type, use the syntax

`name_of_struct variable_name;`

in the same way you would declare a variable of any other type. For instance, in `main()` or any other function, including the line of code

```
point pt;
```

tells C++ to declare a variable `pt` of type `point`, just as the line `int i;` declares a variable `i` of type `int`.

The dot operator

To access the individual fields of a struct, use the dot operator (a period). The left side of the dot operator must always a struct variable (or an expression that can be interpreted as a struct variable) and the right side must always be the name of a struct field.

For instance, once we have declared our point `pt`, we can set its fields as follows:

```
pt.x = 2;
pt.y = 4;
```

These two lines of code set the two fields separately; together we can interpret these lines of code as setting the point equal to the coordinate (2, 4) in the Cartesian plane.

Combining a struct variable name with the name of a field of that struct using the dot operator results in a variable that can be used the same as any other variable in the program. In other words, the variables `pt.x` and `pt.y` can be used any place in a program where a non-struct `int` variable would make sense (e.g., with `cin`, `cout`, function calls, returning values, math, etc). All of these lines are fine:

```
cout << pt.x << endl;
cin >> pt.y;
pt.x = (5 + pt.y) * 7;
pt.y += 2;
```

```
pt.x++;
if (pt.x < pt.y)
      cout << "the x coordinate is bigger!"
cout << square(pt.x) << endl;       // assuming you have a function square
                                    // that takes an int parameter
```

Recent C++ compilers allow the programmer to set all the fields of a struct at once, but only when the struct is first declared.

```
point otherpt = {5, -6};
```

Values must be given inside the curly braces in the order that they are defined in the struct definition.

After a struct variable is declared, there is no way to directly set all of the fields at once; they must be set individually:

```
// otherpt = {0, -1} is illegal!  Instead use these lines:
otherpt.x = 0;
otherpt.y = -1;
```

For structs that are defined to contain other struct variables as fields, the dot operator can be used multiple times. For instance, the code below manipulates a mouse struct variable using the mouse type defined earlier:

```
mouse fievel;
fievel.name = "Fievel Mousekewitz";  // see An American Tail
fievel.location.x = 7;
fievel.location.y = 4;
cout << "The mouse " << fievel.name << " is at position "
      << fievel.location.x << ", " << fievel.location.y << endl;
```

Struct assignment

Structs can be used on both sides of an assignment statement:

```
point p1 = {1, 2}, p2 = {3, 4};
p2 = p1;
```

The second line above *copies* all the fields of `p1` into the respective fields of `p2`. After this line of code, there is no further connection between `p1` and `p2`; they are still completely separate and independent variables.

Passing struct types to functions

Any function can take a parameter of a struct type like any other parameter. The function below defines a function that takes one argument of type point:

```
void print_point(const point & p)
{
  cout << "(" << p.x << ", " << p.y << ")" << endl;
}
```

Because structs can take up lots of memory, it is a good rule of thumb to always pass struct types to functions by reference or const reference, whichever is appropriate.

Returning struct types from functions

Any function can return struct type as a return value. The function below computes the point in the plane halfway between two other points (as close as can be represented with ints, anyway):

```
point midpoint(const point & p1, const point & p2)
{
  point middle;
  middle.x = (p1.x + p2.x) / 2;
  middle.y = (p1.y + p2.y) / 2;
  return middle;
}
```

This function could also have been written like this:

```
point midpoint(const point & p1, const point & p2)
{
  point middle = { (p1.x + p2.x) / 2, (p1.y + p2.y) / 2 };
  return middle;
}
```

Here is some code that uses the two functions (`print_point` and `midpoint`) that we wrote above:

```
point a, b;
cout << “Type in the x and y coordinates of point a: “;
cin >> a.x >> a.y;
cout << “Type in the x and y coordinates of point b: “;
cin >> b.x >> b.y;
point midpt = midpoint(a, b);
cout << “The midpoint is “;
print_point(midpt);
```

The last three lines of the code excerpt above could have been replaced with:

```
cout << “The midpoint is “;
print(midpoint(a, b));
```