CHAPTER 19

Struct and Union

Struct

A struct in C++ is equivalent to a class, except that members of a struct default to public access, instead of private access as in classes. By convention, structs are used instead of classes to represent simple data structures that mainly contain public fields.

```
struct Point
{
  int x, y; // public
};

class Point
{
  int x, y; // private
};
```

Declarator list

To declare objects of a struct the normal declaration syntax can be used.

```
Point p, q; // object declarations
```

Another alternative syntax commonly used with structs is to declare the objects when the struct is defined by placing the object names before the final semicolon. This position is known as the *declarator list* and can contain a comma-separated sequence of declarators.

```
struct Point
{
  int x, y;
} r, s; // object declarations
```

Aggregate initialization

There is a syntactical shortcut available when initializing an object called *aggregate initialization*. This allows programmers to set fields by using a brace-enclosed comma-separated list of initializer-clauses. Aggregate initialization can only be used when the class type does not include any constructors, virtual functions or base classes. The fields must also be public, unless they are declared as static. This is the reason why it is more common to use aggregate initialization with structs rather than with classes.

```
struct Point {
  int x, y;
} r = { 2, 3 }; // set values of x and y

int main() {
  Point p = { 2, 3 };
}
```

Union

Although similar to struct, the union type is different in that all fields share the same memory position. Therefore, the size of a union is the size of the largest field it contains. For example, in the case below this is the integer field which is 4 bytes large.

```
union Mix
{
  char c;  // 1 byte
  short s; // 2 bytes
  int i;   // 4 bytes
} m;
```

This means that the union type can only be used to store one value at a time, because changing one field will overwrite the value of the others.

```
int main()
{
  m.c = 0xFF; // set first 8 bits
  m.s = 0;    // reset first 16 bits
}
```

The benefit of a union, in addition to efficient memory usage, is that it provides multiple ways of viewing the same memory location. For example, the union below has three data members that allow access to the same group of 4 bytes in multiple ways.

```
union Mix
{
  char c[4];                  // 4 bytes
  struct { short hi, lo; } s; // 4 bytes
  int i;                      // 4 bytes
} m;
```

The integer field will access all 4 bytes at once. With the struct 2 bytes can be viewed at a time, and by using the char array each byte can be referenced individually.

```
int main()
{
  m.i=0xFF00F00F; // 11111111 00000000 11110000 00001111
  m.s.lo;         // 11111111 00000000
  m.s.hi;         //                  11110000 00001111
  m.c[3];         // 11111111
  m.c[2];         //          00000000
  m.c[1];         //                   11110000
  m.c[0];         //                            00001111}
```

Anonymous union

A union type can be declared without a name. This is called an *anonymous union* and defines an unnamed object whose members can be accessed directly from the scope where it is declared. An anonymous union cannot contain methods or non-public members.

```
int main()
{
  union { short s; }; // defines an unnamed union object
  s = 15;
}
```

An anonymous union that is declared globally must be made static.

```
static union {};
```