struct **[C]**

- The C style `struct` is the precursor to C++ classes, and organizes different types of items into a single data item. `struct` is worth knowing as it's a simple and fast alternative to full blown C++ classes.
- Structures are most commonly used to define an object-a person, a place, a thing, or a record in a database or file, e.g.

```
struct StudentType
{
      string      name;
      char        letterGrade;
      float       testAverage;
      int         testScores[10];
      bool        ownsComputer;
};
```

- The "field identifiers" `name`, `letterGrade`, … are like variable names. They must be unique within a struct though may duplicate names of variables or field identifiers in other data structures.
- Initialize a `struct` much like you would a normal variable.

```
StudentType studentA;
```

- Field identifiers may be accessed and initialized using the "dot" operator.

```
studentA.name = "Brandy Alexander";
studentA.letterGrade = 'B';
studentA.testAverage = 7.3;
cout << "name: " << studentA.name << endl;
cout << "letterGrade: " << studentA.letterGrade << endl;
cout << "testAverage: " << studentA.testAverage << endl;
```

- You may also initialize a `struct` in the much the same way you would an array.

```
StudentType studentB = {"Andrew Beatnik", 'A', 9.1};
cout << "name: " << studentB.name << endl;
cout << "letterGrade: " << studentB.letterGrade << endl;
cout << "testAverage: " << studentB.testAverage << endl;
```

- A simpler though equally common example of a `struct` is a point. This `Point3d` holds three values `x`, `y`, and `z`.

```
struct Point3d
{
      int x;
      int y;
      int z;
};
```

- One might make a simple function that prints a point. Passing it by reference.

```
void printPoint(Point3d pt)
{
      cout << "x:" << pt.x << " y:" << pt.y
                        << " z:" << pt.z << endl;
}
...
      Point3d ptA = {3, 2, 7};
```

```
Point3d ptB = {7, 4, 5};

printPoint(ptA);
printPoint(ptB);
```

- Functions may have structures as arguments and return values.
- You may have an array of structures. One of the most common combinations of arrays and structures is an array whose elements are structures. An array of this kind can serve as a simple database.

Passing Structures by Reference

- Note that when passing a struct by value a copy of the entire struct is made for use by the function.
- As some structs may be very large, passing by value might well be undesirable as it takes time and memory to copy the `struct` between functions. Instead one should usually pass structs by reference.
- The `printPoint()` function done by reference might look as follows,

```
void printPointByRef(Point3d* pt)
{
      cout << "x:" << (*pt).x << " y:" << (*pt).y
                           << " z:" << (*pt).z << endl;
}
```

- But the syntax here is starting to get confusing with all the dereference of a pointer and then use the dot `(*pt).x`
- To simplify the syntax there is an “arrow operator” `->`
- Using the `->` operator in the `printPointByRef()` function allows it to be written like so,

```
// with "arrow syntax"
void printPointByRef(Point3d* pt)
{
      cout << "x:" << pt->x << " y:" << pt->y
                           << " z:" << pt->z << endl;
}
```

- Basically, the arrow operator should be used wherever you would use dot . syntax, but on a pointer.
- Since any element of an array or struct can itself be an array or struct it is possible to generate very complex data types, subject only to the rule that no data type may be used before its declaration is complete.

struct in C [C]

- `struct` originated in C, though their use in C is slightly more fussy than C++, requiring the use of the `struct` keyword when passed to a function and on initialization. In C, the Point3d code above would look like this,

```
struct Point3d
{
      int x, y, z;
};


void printPoint(struct Point3d pt)
{
      printf("x: %i y: %i z: %i \n", pt.x, pt.y, pt.z);
}

int main()
{
      struct Point3d ptA = {3, 2, 7};
      struct Point3d ptB = {7, 4, 5};

      printPoint(ptA);
      printPoint(ptB);

      return 0;
}
```

- Note the additional struct declarations shown in **bold**.
- C++ deduces it's a struct and doesn't need the struct keyword.

typedef [C]

- `typedef` may be used to give a data type a new name.
- `typedef` is often used to simplify the syntax of declaring complex data structures consisting of struct and union types, but is just as common in providing specific descriptive type names for integer data types.
- Here is an example to define a term `BYTE` for one-byte numbers,

```
typedef unsigned char byte;
```

- After this type definition, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example.

```
byte b1, b2;
```

- When working in C it's traditional to use the `typedef` keyword when declaring a `struct` as a short cut so not to have to include the various additional struct keywords shown in **bold** in the "struct in C" example above.
- Such a version of the `Point3d` code in C using `typedef` would look like this,

```
typedef struct
{
      int x, y, z;
} Point3d;

void printPoint(Point3d pt)
{
      printf("x: %i y: %i z: %i \n", pt.x, pt.y, pt.z);
}

int main()
{
      Point3d ptA = {3, 2, 7};
      Point3d ptB = {7, 4, 5};

      printPoint(ptA);
      printPoint(ptB);

      return 0;
}
```

- Some people are opposed to the extensive use of typedefs. Most arguments center on the idea that typedefs simply hide the actual data type of a variable and not only are unnecessarily but obfuscates code. They can also cause programmers to accidentally misuse large structures thinking them to be simple types.

union [C]

- Unions are moderately useful in C and largely useless in C++.
- Unions in C are almost exclusively used for type-safe polymorphism. Something C++ classes are much better at.
- C++'s substitute for unions is classes & inheritance.