**struct** is a group of data grouped under a single name. The data does not have to be of the same type.

```c
struct student {
    int sid;
    char* name;
    int year;
    string field;
    int age;
    double wage;
    double gpa;
};

struct prof {
    bool tenure;
    string name;
    string field;
    double wage;
};
```

```c
int main() {
    student brian;
    prof david;

    // set some things
    brian.sid = 123456789;
    david.name = "David";

    // see some things
    cout << brian.name << endl;
    cout << david.field << endl;
    if (david.tenure) {
        david.wage++;
    }
}
```

- access struct fields via “.”

bad call, segfaults since name isn’t set
### Classes

**class** is pretty much the same as a struct, but slightly more private.

```cpp
class student {
public:
    int sid;
    char* name;
    int year;
    string field;
    int age;
    double wage;
    double gpa;
};

class prof {
public:
    bool tenure;
    string name;
    string field;
    double wage;
};
```

```cpp
int main() {
    student brian;
    prof david;

    // set some things
    brian.sid = 123456789;
    david.name = "David";

    // see some things
    cout << brian.name << endl;
    cout << david.field << endl;

    if (david.tenure) {
        david.wage++;
    }
}
```

In a struct, all functions and variables default to public access. In a class, they default to private:

- **public**: anything with access to the student object can access the *public* variables
- **private**: only student objects can access the *private* variables

Restrictive access is good programming practice to ensure the integrity of your objects (i.e., prevent random objects from screwing up your object’s state or data (e.g., changing field to “the crow flies at midnight”)).
struct & object access practice

```
class student {
public:
    int sid;
    int age;
    double wage;
    double gpa;
    string name;
    string field;
    int year;
};

struct prof {
    string name;
    string field;
    double wage;
    bool tenure;
};

int main() {
    student brian;
    prof david;
    student* pbrian = &brian;
    prof* pdavid = &david;
}

pbrian->wage*3 + pdavid->wage;

OR

(student a; pbrian = &a;

pbrian->name = david.name;

(pBrian).wage*3 + (pDvaid).wage;

OR

pbrian->wage*3 + pdavid->wage;

(the two are equivalent)
```

**how do I?**

set brian’s name to “david”?

```
brian.name = “David”;
```

compare brian’s and david’s field of study

```
brian.field == david.field;
```

find total wages for a prof and his lab

```
brian.wage*3 + david.wage;
```

(assuming that all students make the same wages)

```
(*pbrian).wage*3 + (*pdavid).wage;
```

do the same using pointers

```
brian.name = david.name;
```

set pbrian to another student object you create

```
student a; pbrian = &a;
```

swap brian’s (via pbrian) & david’s (via david) names

```
string s = pbrian->name;
pbrian->name = david.name;
david.name = s;
```
**member functions** are functions specific to a class or struct definition (e.g. get and set functions)

```cpp
class prof {
private:
    bool tenure;
    string name;
    string field;
    double wage;

public:
    // some useful fxns
    void talk();
    void teach();

    // variable access fxns
    string get_name() {
        return name;
    }
    void set_name(string s) {
        name = s;
    }

    ...}

void prof::talk() {
    cout << "Did you read ";
    cout << "the spec?";
}
```

```cpp
int main() {
    prof david;

    // set some things
    david.set_name("David");
    david.set_tenure(true);

    // see some things
    cout << "Prof. " << david.get_name();
    cout << " says, "" << david.talk();
    cout << "/"" << endl;

    if (david.get_tenure()) {
        david.set_wage(david.get_wage() + 1);
    }

    cout << "Field: " << david.get_field();
}
```
struct & object access practice

```cpp
class student {
    int sid;
    int age;
    double wage;
    double gpa;
public:
    string name;
    string field;
    int year;
};

struct prof {
    string name;
    string field;
    double wage;
private:
    bool tenure;
};

int main() {
    student brian, jim;
    prof david, carey;
    student* ps = &jim;
    prof* pp = &david;
};
```

some statements later in main()

```cpp```
brian.name = "jim"; david.name = "Dave";
brian.sid = 123456789;
brian.set_sid(12345678);
david.wage += pp->tenure();
jim.wage += (jim.get_gpa() / 4.0);
jim.set_wage(jim.get_wage() + jim.get_gpa()/4.0);
assert(pp->wage > ps->get_wage());
carey.set_wage(pp.get_wage())
carey.set_wage(pp->get_wage())
if (carey.tenure) jim.gpa++;
if (carey.tenure()) jim.set_gpa(jim.get_gpa()+1);
```cpp```

what happens?

names are set for people

```cpp```
error, sid is private & not accessible here
sets sid using class member functions
increases david’s income if he is tenured
```cpp```
error! student wage is private. We need to use student member functions to access it (e.g. student.get_wage, student.set_wage)
```cpp```
ensures professor makes more than student
```cpp```
error! pp is a pointer, should use "->". Using set_wage is fine for public variables too, but not necessary as prof.wage is public
```cpp```
error! tenure and gpa are private member variables of student and prof, respectively. Use get/set functions
int main() {
    prof david;

    // set some things
    david.name(“David”);
    david.tenure(true);

    // see some things
    cout << “Prof. “ << david.name() << “ says, “ << david.talk() << “/” << endl;

    // combo
    if (david.tenure()) {
        david.wage(david.wage() + 1);
    }
    cout << “Field: “ << david.field() << “/” << endl;
}

- private variables preceded by “_”
- get/set functions overloaded with same name
  - using variable name with no arguments gets the value
  - using variable name with arguments sets it to input value
objects + functions

example function header with a pass by reference input for a dog object

```c
void wash(dog &d)
```

**objects as input arguments:** can be passed by value, by reference, and via pointer

1. which will allow the object to be changed? pass by reference and via pointer

2. to protect arguments from being modified, use const

**examples:**
- `const dog &d`
  - cannot change object d refers to
- `const dog *d`
  - cannot change object d points to

how many ways can we define a function to change a dog’s cleanliness state such that the dog stays clean after the function exits? (i.e. set dog.clean to true)

**definition**

```c
void wash(dog &d) { d._clean = true }
void wash(dog &d) { d.clean(true) }
void wash(dog *d) { d->_clean = true }
void wash(dog *d) { d->clean(true) }
dog* wash(dog *d) { d->_clean = true }
dog* wash(dog *d) { d->clean(true) }
```

**main**

```c
wash(d)
wash(pd)
pd = wash(pd)
```

**what’s happening?**

- pass d by reference
- pass d by pointer (assuming pd points to d)
- pass d by pointer and use return value

if clean is declared as private

```c
void wash(dog &d) { d.clean(true) }
void wash(dog *d) { d->clean(true) }
dog* wash(dog *d) { d->clean(true) }
```
**Constructors**

Constructors are used to ensure objects are valid instances of a class (i.e. that all private member variables are initialized to sensible starting values)

```cpp
class prof {
    private:
        bool _tenure;
        string _name;
        string _field;
        double _wage;
    
    public:
        prof() { // default
            _tenure = false;
            _name = "Prof";
            _field = "CS";
            _wage = 200000;
        }

        prof(bool t, string n, string f, double w) {
            _tenure = t;
            _name = n;
            _field = f;
            _wage = w;
        }

    // if not defined, a default constructor is automatically created for you
};
```

```cpp
int main() {
    prof bill;
    prof david(true, "David", "CS", 250000);
    // set some things
    bill.name("Bill");
    bill.tenure(true);

    // see some things
    cout << "Prof. " << bill.name();
    cout << "Prof. " << david.name();
    if (david.tenure()) {
        david.wage(david.wage() + 1);
    }
}
```
arrays of objects

the line below creates an array of 10 dog objects, using the default constructor to construct each one

\[
\text{dog} \ \text{puppies}[10];
\]

objects arrays as input arguments to functions: like any array, passed by reference

how many ways can we define a function to wash all the puppies? (i.e. set dog.clean to true)

```java
\[
\text{void} \ \text{wash}() \ { \ \text{name} \ \text{size} \\
\text{for} (\text{int} \ k = 0; \ k < \text{n}; \ k++) \ { \\
\text{\_C\_;} \\
\} \\
\text{\_D\_;} \\
}\]
```

<table>
<thead>
<tr>
<th><em>A</em></th>
<th><em>B</em></th>
<th><em>C</em></th>
<th><em>D</em></th>
<th>main</th>
<th>what’s happening</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>dog litter[]</td>
<td>litter[k].clean(true)</td>
<td>N/A</td>
<td>wash(puppies)</td>
<td>array is passed by reference</td>
</tr>
<tr>
<td>void</td>
<td>dog* litter</td>
<td>litter[k].clean(true)</td>
<td>N/A</td>
<td>wash(puppies)</td>
<td>passing array via pointer</td>
</tr>
<tr>
<td>dog*</td>
<td>dog* litter</td>
<td>litter[k].clean(true)</td>
<td>return litter</td>
<td>puppies = wash(puppies)</td>
<td>setting returned pointer</td>
</tr>
</tbody>
</table>

now what if puppies was an array of dog pointers o_O?! (e.g. dog* puppies[10] and all elements point to a unique dog object)

<table>
<thead>
<tr>
<th><em>A</em></th>
<th><em>B</em></th>
<th><em>C</em></th>
<th><em>D</em></th>
<th>main</th>
<th>what’s happening</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>dog* litter[]</td>
<td>litter[k]-&gt;clean(true)</td>
<td>N/A</td>
<td>wash(puppies)</td>
<td>array is passed by reference</td>
</tr>
<tr>
<td>void</td>
<td>dog** litter</td>
<td>litter[k]-&gt;clean(true)</td>
<td>N/A</td>
<td>wash(puppies)</td>
<td>passing array via pointer to dog pointer</td>
</tr>
<tr>
<td>dog**</td>
<td>dog** litter</td>
<td>litter[k]-&gt;clean(true)</td>
<td>return litter</td>
<td>puppies = wash(puppies)</td>
<td>setting returned pointer</td>
</tr>
</tbody>
</table>
const int MAXSTUDENTS = 1000;
class dis {
private:
    student _students[MAXSTUDENTS];
    char _name[64];
    int _nstudents;

public:
    dis(string name, int n);

    mean_gpa();
};

dis::dis(string name, int n) {
    // name
    strcpy(_name, name.c_str());

    int _nstudents = n;

    // generates students
    for (int i = 0; i < n; i++) {
        _students[i].gpa = 0.0;
    }
}

double dis::mean_gpa() {
    double sum_gpa = 0.0;
    for (int i = 0; i < _nstudents; i++) {
        sum_gpa += _students[i].gpa();
    }
    return sum_gpa / _nstudents;
}

int main() {
    dis 1H("1H", 13);
    dis 1G("1G", 15);

    ... // some projects/tests alter gpa

    if (1H.mean_gpa() < 1G.mean_gpa()) {
        cout << "Brian gets fired" << endl;
    } else {
        cout << "Costas gets fired" << endl;
    }
}
object classes: p2 - smart water bottle

design a water bottle that keeps track of how much the owner drinks each day for up to 7 days. The tiny LCD display shows the user how much water is being consumed on average over the past week. An example program of the user and bottle follows. Implement the necessary functions.

```cpp
int main() {
    SmartBottle bottle(16); // create an empty bottle with a maximum capacity (oz)
    bottle.add(2); // adds water (in oz)
    cout << bottle.left() << endl; // returns how many oz the bottle has
    bottle.drink(4); // how much water one drinks (no more than in the bottle)
    bottle.nextday(); // starts a new day of recording
    bottle.add(10);
    bottle.drink(4);
    bottle.nextday();
    bottle.add(12); // :( - spilled 2 oz
    bottle.drink(5);
    bottle.add(3);
    bottle.drink(4);
    bottle.drink(4);
    bottle.drank(0); // tells how much was drank today
    bottle.drank(1); // tells how much was drank: 1=yesterday, 2=day before, ...
    bottle.nextday();
    bottle.eject(); // empties the bottle
    bottle.add(2);
    bottle.drink(1);
    if (bottle.consumed_avg() < 10) {
        cout << "Drink more water!" << endl;
    }
}
```
First, what functions do we need to write? Second, figure out what variables you might need (likely an int for the number of oz in the bottle and an array of ints for how much was drank each day over the past 7 days). Finally, start writing the class.

```cpp
int main() {
    SmartBottle bottle(16); // create an empty bottle with a maximum capacity (oz)
    bottle.add(2); // adds water (in oz)
    cout << bottle.left() << endl; // returns how many oz the bottle has
    bottle.drink(4); // how much water one drinks (no more than in the bottle)
    bottle.nextday(); // starts a new day of recording
    bottle.add(10);
    bottle.drink(4);
    bottle.nextday();
    bottle.add(12); // :( - spilled 2 oz
    bottle.drink(5);
    bottle.add(3);
    bottle.drink(4);
    bottle.drink(4);
    bottle.drank(0); // tells how much was drank today
    bottle.drank(1); // tells how much was drank: 1=yesterday, 2=day before, ...
    bottle.nextday();
    bottle.eject(); // empties the bottle
    bottle.add(2);
    bottle.drink(1);
    if (bottle.consumed_avg() < 10) { // returns average oz consumed per day
        cout << “Drink more water!” << endl;
    }
}
```
object classes: p2 - smart water bottle (answer)

class declaration
const int NDAYS = 7;
class SmartBottle {
    public:
        SmartBottle(int max);
        void add(int n_oz);
        void drink(int n_oz);
        void nextday();
        void eject();
        int left() const;
        int drank(int day) const;
    double consumed_avg();

    private:
        int _amount;
        int _max_amount;
        int _drank[NDAYS];
        int _day;
};

start with ez fxns
void SmartBottle::eject() {
    _amount = 0;
}

int SmartBottle::left() const {
    return _amount;
}

complex fxns
int SmartBottle::drank(int n) {
    if ((n > 0) && (n < NDAYS)) {
        int old_day = _day - n;
        if (old_day < 0) {
            old_day += NDAYS;
        }
        return _drank[old_day];
    }
}

int SmartBottle::consumed_avg() {
    double avg = 0.0;
    for (int k = 0; k < NDAYS; k++) {
        avg += _drank[k];
    }
    return avg / NDAYS;
}

constructor
SmartBottle::SmartBottle(int max) {
    _max_amount = max;
    _amount = 0;
    _day = 0;
    for (int k = 0; k < NDAYS; k++) {
        _drank[k] = 0;
    }
}

member functions
void SmartBottle::add(int noz) {
    if (noz > 0) {
        _amount += noz;
        if (_amount > _max_amount) {
            _amount = max_amount;
        }
    }
}

void SmartBottle::drink(int noz) {
    if (noz > 0) {
        if (noz > _amount) {
            noz = _amount;
        }
        _amount -= noz;
        _drank[_day] += noz;
    }
}

void SmartBottle::nextday() {
    _day++;
    if (_day == NDAYS) {
        _day = 0;
    }
    _drank[_day] = 0;
}