Objects and nesting and pointers, oh my!

- If we have a pointer variable `ptr`, we access ***the thing that a pointer points to*** with the syntax

```
*ptr
```

- To access a field or method of a class through an object variable, use the syntax

```
variable.field    OR   variable.method()
```

- So what if `ptr` points to an object?

-
  ```
  string s = "Hello";
  string *sptr = &s;
  string *sptr2 = new string("Goodbye");
  ```

- We access the string through the pointer using the same syntax:

  ```
  cout << s;        // regular access
  cout << *sptr;    // through pointer
  cout << *sptr2;   // through pointer
  ```

- Problem occurs if we want to get the length of the string through the pointer:

-
  ```
  string s = "Hello";
  string *sptr = &s;
  cout << *sptr.length();    // error
  ```

- Reason: the dot operator has higher precedence than the dereference (*) operator, so C++ interprets this as:

  ```
  cout << *(sptr.length());
  ```

- Two ways to fix this.
- Method 1: Use parentheses to change order of operations:

-
  ```
  string s = "Hello";
  string *sptr = &s;
  cout << (*sptr).length();    // OK
  ```
- Method 2: Use the arrow operator, which combines the dereference * and dot operator into one:

-
  ```
  cout << sptr->length();      // OK
  ```
- Method 2 is much more common.

Rule

- To access a field or method of a class through an **object variable**, use the syntax

  ```
  variable.field    OR   variable.method()
  ```

- To access a field or method of a class through a **pointer to an object**, use the syntax

  ```
  ptr->field   OR   ptr->method();
  ```

```
class thingy {
  public:
  int x;
  void f();
};
```

```
thingy thing;
cout << thing.x;
thing.f();

thingy *thing_ptr = &thing;
cout << thing_ptr->x;
thing_ptr->f();

thingy *tptr2 = new thing;
cout << tptr2->x;
tptr2->f();
```

```
class thingy {
  public:
  int x;
  void f();
};
```

```
vector<thingy> tvec;
// add things to tvec
cout << tvec[0].x;
tvec[0].f();

vector<thingy*> ptrvec;
// add things to ptrvec
cout << ptrvec[0]->x;
ptrvec[0]->f();
```

```
class thingy {
  public:
  int x;
  void f();
};
```

```
class thingy {
  public:
  int x;
  void f();
};

class doohickey {
  public:
  int y;
  void g();
  thingy t;
  thingy *tptr;
}
```

```
doohickey doohick;
cout << doohick.y;
doohick.g();
cout << doohick.t.x;
doohick.t.f();

doohickey *dptr = &doohick;
cout << dptr->y;
dptr->g();
cout << dptr->t.x;
dptr->t.f();
```

```
class thingy {
  public:
  int x;
  void f();
};

class doohickey
{
  public:
  int y;
  void g();
  thingy t;
  thingy *tptr;
}
```

```
doohickey doohick;
doohickey *dptr = &doohick;

doohick.tptr = new thingy;

cout << doohick.tptr->x;
doohick.tptr->f();

cout << dptr->tptr->x;
dptr->tptr->f();
```

```
class thingy {
   public:
   int x;
   void f();
};

class doohickey
{
   public:
   int y;
   void g();
   thingy t;
   thingy *tptr;
}
```

```
vector<doohickey> dvec;
vector<doohickey*> dptrvec;
// add stuff to vectors

cout << dvec[0].t.x;
cout << dvec[0].tptr->x;
cout << dvecptr[0]->t.x;
cout << dptrvec[0]->tptr->x;
```

```
class thingy {
   public:
   int x;
   void f();
};

class doohickey
{
   public:
   int y;
   void g();
   thingy t;
   thingy *tptr;
}
```