RANDOM CATCH-UP STUFF

A new thing...

- We often find ourselves doing stuff like this:

```
int bob;

if( someConditionIsTrue )
    bob = 17;
else
    bob = 96;
```

- ... where we just want to execute a single statement based on the outcome of some condition (here, setting a value).

A Shortcut:

- C++ provides us a nifty shortcut to do this sort of thing:
- **The TERNARY OPERATOR!!!!**
  - (what does ternary mean?)

An Example

This unwieldy piece of code:

```
int bob;

if( someCondition )
    bob = 17;
else
    bob = 96;
```

can be reduced to this:

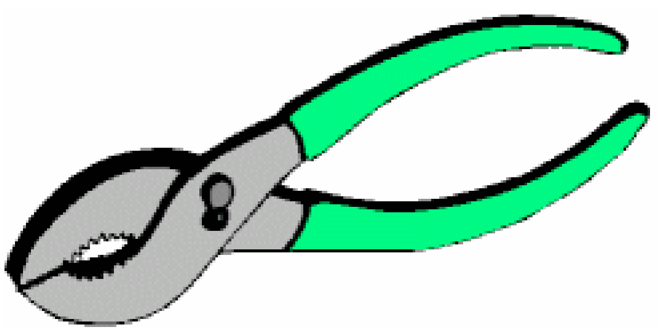
```
int bob = someCondition ? 17 : 96;
```

Anatomy of the Ternary Operator

```
condition ? truePart : falsePart
```

this would go in the if statement

the *single statement* that gets executed if **condition** is true



the *single statement* that gets executed if **condition** is false

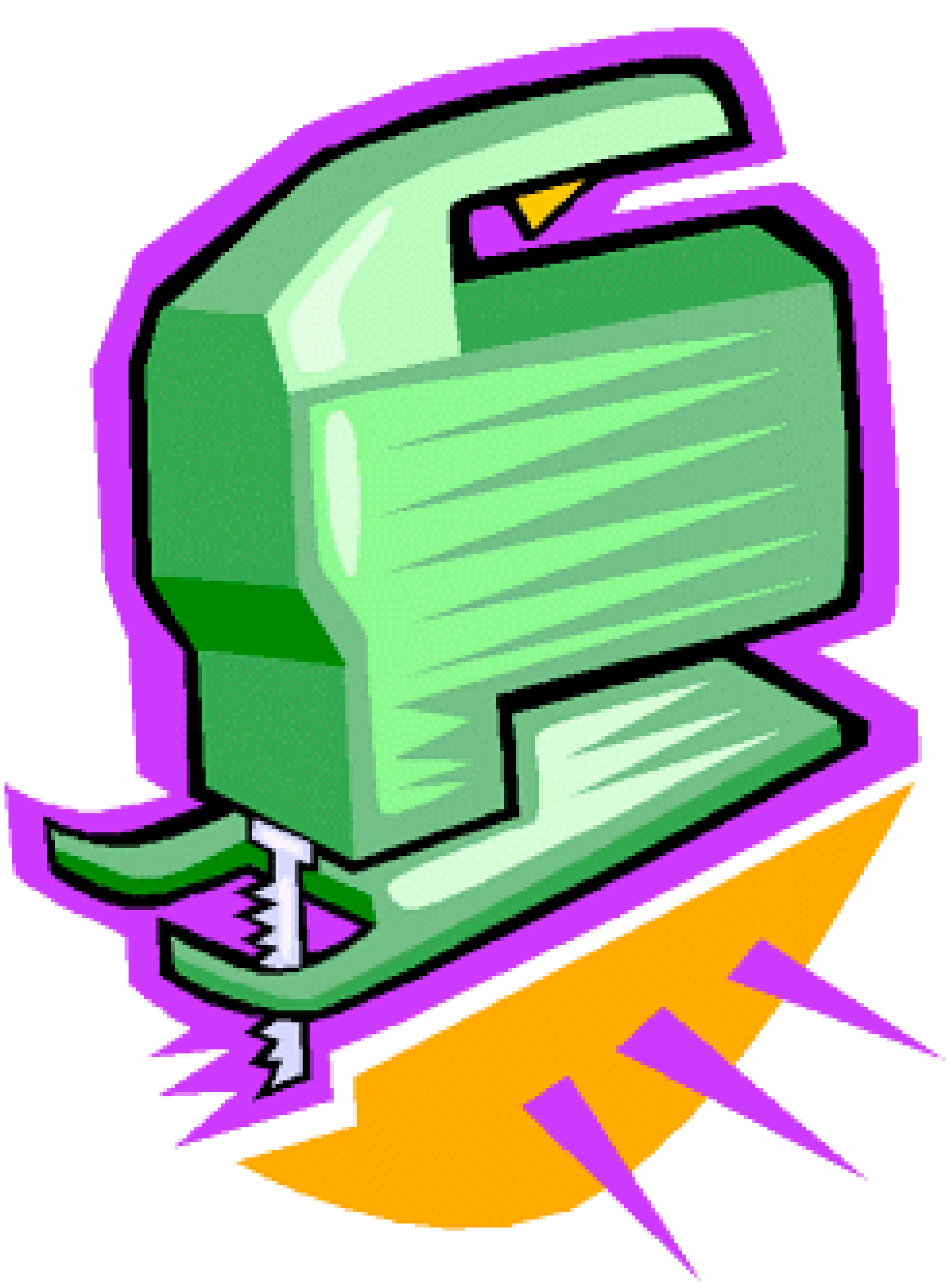
Usages

- What is this good for?
- Shortening code

```
int max( int a, int b )
{
    return a > b ? a : b;
}
```

- Assigning const values conditionally

```
bool correct = getValue();
const int PI = correct ? 3.14 : 92.8;
```



Question

- Hopefully you should know the answer to this by now...
- Why might the ternary operator not always be a good idea?

Bad Code!

- On the other end of the conditional execution scale:
- When you are testing a single value against a lot of conditions, you get a lot of hard-to-read code
- Like this!

```
int input = getInput();

if( input == 0 )
    doStuff();
else if( input == 1 )
    doSomethingElse();
else if( input == 2 )
    doAThirdThing();
else if( input == 3 )
    playSpades();
else if( input == 4 )
    watchScrubs();
else if( input == 5 )
    goBirdWatching();
else if( input == 6 )
    eatHamburger();
```

the switch statement

- The switch statement is often a more elegant, sometimes faster way to do this
- **switch** tests a single *integer* variable against a large number of conditions
- Here we're checking input against 0 - 6

```
int input = getInput();

switch( input )
{
    case 0: doStuff();
            break;
    case 1: doSomethingElse();
            break;
    case 2: doAThirdThing();
            break;
    case 3: playSpades();
            break;
    case 4: watchScrubs();
            break;
    case 5: goBirdWatching();
            break;
    case 6: eatHamburger();
            break;
}
```

ARROWED!!!

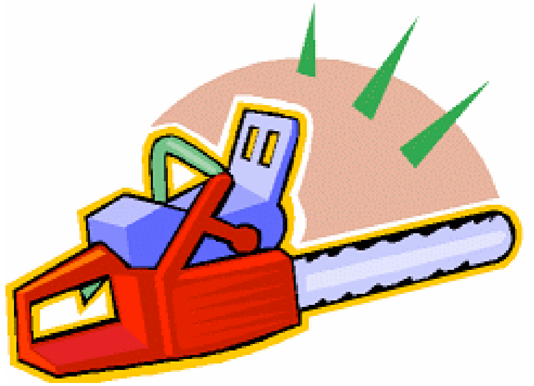
this can be any *integer* expression - in parenthesis, just like an if statement

switch keyword

case statement: must be unique!

entire switch statement enclosed in curly braces

```
int input = getInput();

switch( input )
{
    case 0: doStuff();
            break;
    case 1: doSomethingElse();
            break;
    case 2: doAThirdThing();
            break;
    case 3: playSpades();
            break;
    case 4: watchScrubs();
            break;
    case 5: goBirdWatching();
            break;
    case 6: eatHamburger();
            break;
}
```



Case Statements

- When the input value is equal to a *case value*, everything until the next **break** is executed
- Even code in other case statements!
  - this is called falling through
- Any code that can go in a function can go in a case statement

```
char grade = getGrade();

switch( grade )
{
    case ‘A’: callMom();
              cout << “yay!”;
              postOnFridge();
              break;

    case ‘D’: sigh();

    case ‘F’: grumble();
              cout << “boo.”;
              studyHarder();
              break;
}
```

Default Statements

- Code in the **default** statement is executed if none of the case statements are true
- There can be only one of these per switch statement

```
char grade = getGrade();

switch( grade )
{
    case 'A': callMom();
              cout << "yay!";
              postOnFridge();
              break;

    case 'D': sigh();

    case 'F': grumble();
              cout << "boo.";
              studyHarder();
              break;

    default:  cout << "meh.";
              eatHamburger();
              break;
}
```

A Random Note About C++ Conditionals

```
bool one()
{
   cout << “one()” << endl;
   return false;
}

bool two()
{
   cout << “two()” << endl;
   return false;
}

int main()
{
    if( one() && two() )
        cout << “true” << endl;
    return 0;
}
```

What is the output of this program?

Minimal Evaluation

- C++ uses a strategy called *minimal evaluation* or *short circuit evaluation* to avoid doing unnecessary work
- This comes into play with the && operator, which is evaluated left-to-right:

(returns false)

```
if( one() && two() )
    cout << "true" << endl;
```

Minimal Evaluation

- Keep minimal evaluation in mind when writing conditional expressions
- This can actually be really handy!

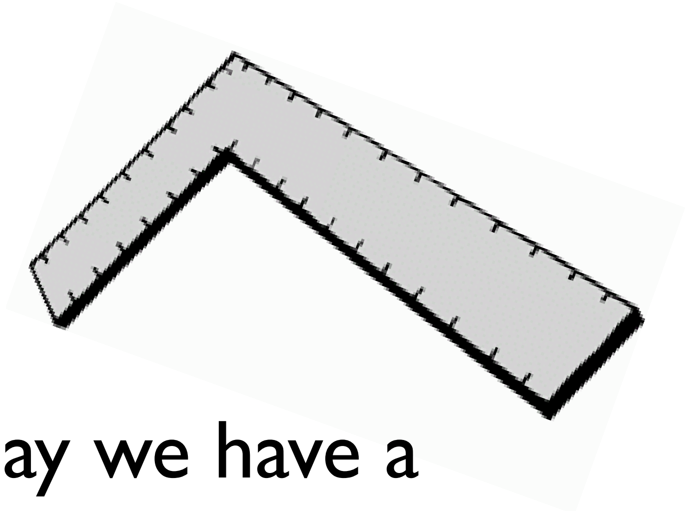
```
if( ptr && ptr->value == 42 )
{
    // do stuff
}
```

- Here, we won't access ptr->value unless ptr is non-null this space intentionally left blank

Inheritance

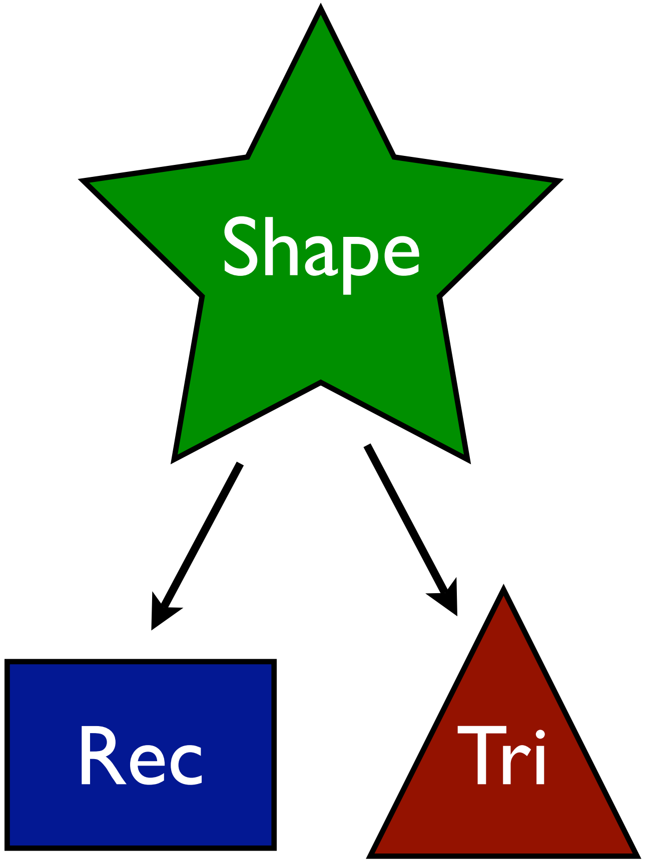
- **Inheritance** is a C++ feature in which one class can "inherit" the member functions and variables from another class
- The new class (the one doing the inheriting) is called the **derived class**
- The class we're inheriting from is called the **base class**

```
class Rectangle
{
  public:
    Rectangle();

    // skipping stuff...

    int area();
    void draw();

  private:
    Color innerColor;
    Color lineColor;
    int lineWidth;
    int x, y;
    int width, length;
    int id;
};
```



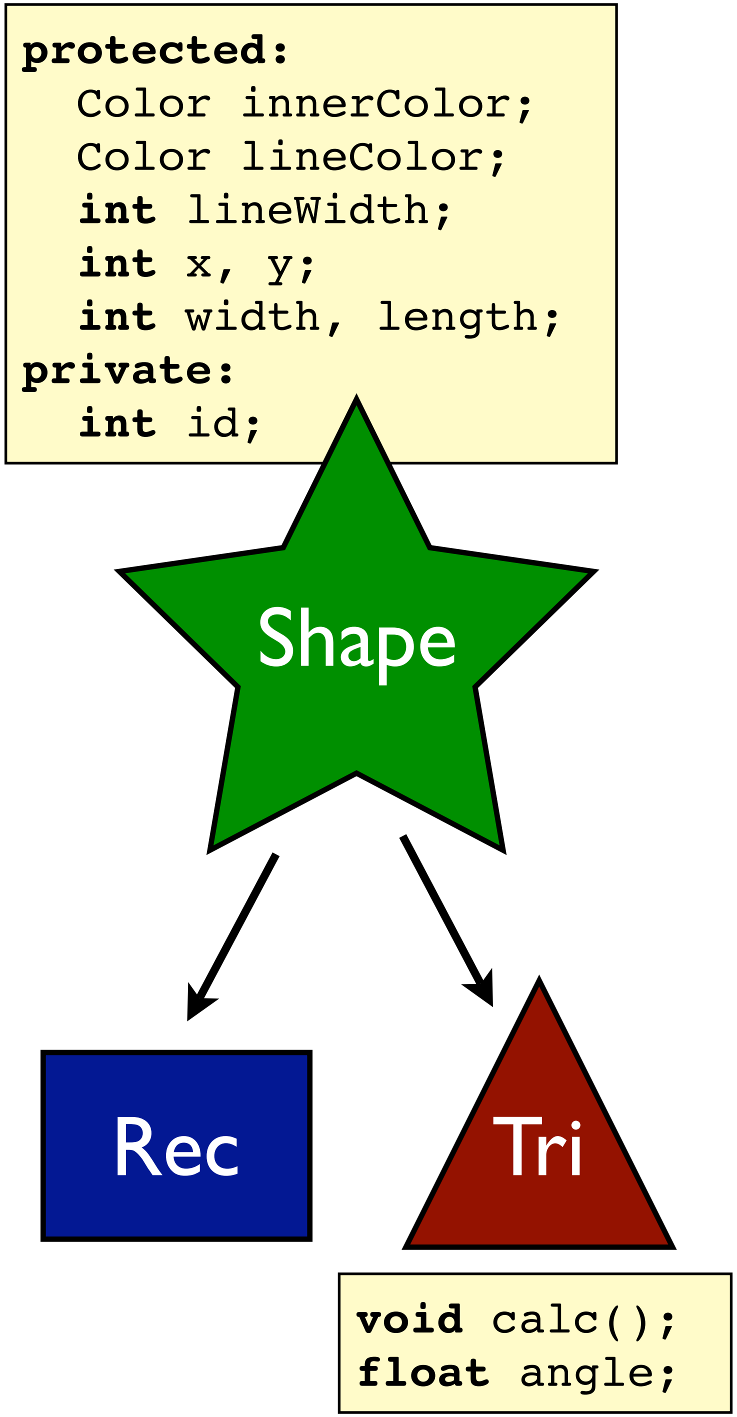
- Let's say we have a **Rectangle** class, with a fair amount of stuff in it
- We'd like to build a simple **Triangle** class
- Most of the code would be the same between these two classes!
- area(), draw() would change

Inheritance



- We could "inherit" most of **Triangle**'s code from **Rectangle**
- A better way: move most of Rectangle's code into a new base class - **Shape** - and derive both Triangle and Rectangle from Shape
- Triangle and Rectangle now only need to implement specific features: the general stuff can be stuck in the Shape class

Inheritance 2



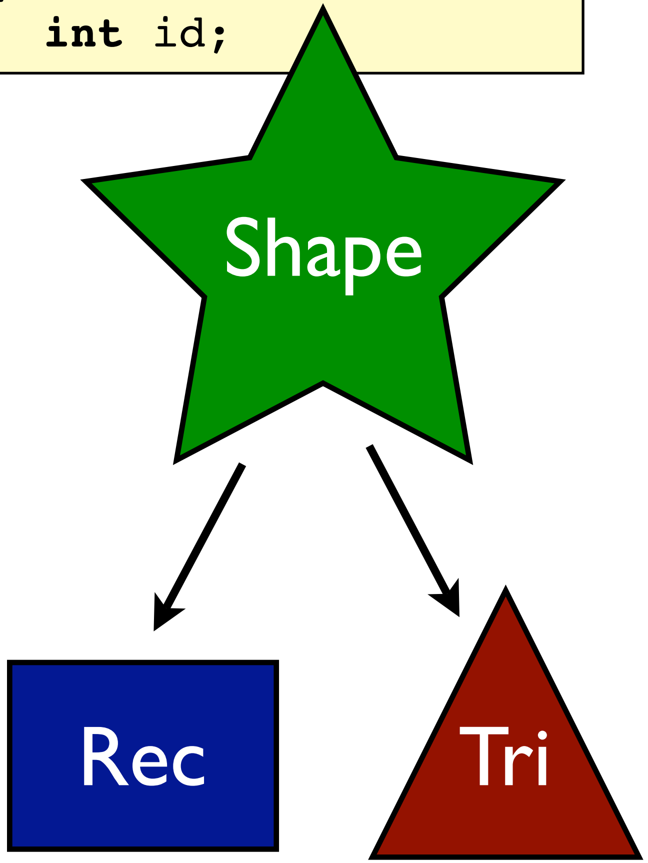
- Derived classes inherit everything in the base class(es)
- Each instance of Triangle has:
  - All the member variables and functions from the Shape class
  - And all the member variables and functions from Triangle
- Triangles has copies of x, y, id, etc. But can it *access* them?

Access Specifiers

- **public** means the same thing it always did
- **private** too: private members can only be accessed from within the class - not any others (including any derived classes!)
- **New! protected** variables can be accessed by the class *and* any derived classes - but not any other class!

Access

```
protected:
   Color innerColor;
   Color lineColor;
   int lineWidth;
   int x, y;
   int width, length;
private:
   int id;
```




```
void calc();
float angle;
```

- So, in this set of classes:
  - innerColor, lineColor, lineWidth, x, y, width, height are all accessible by **Shape**, **Triangle**, **Rectangle**, and no other classes
- id is *only* accessible by **Shape**
- Same access rules apply for member functions

...syntax

class name

colon

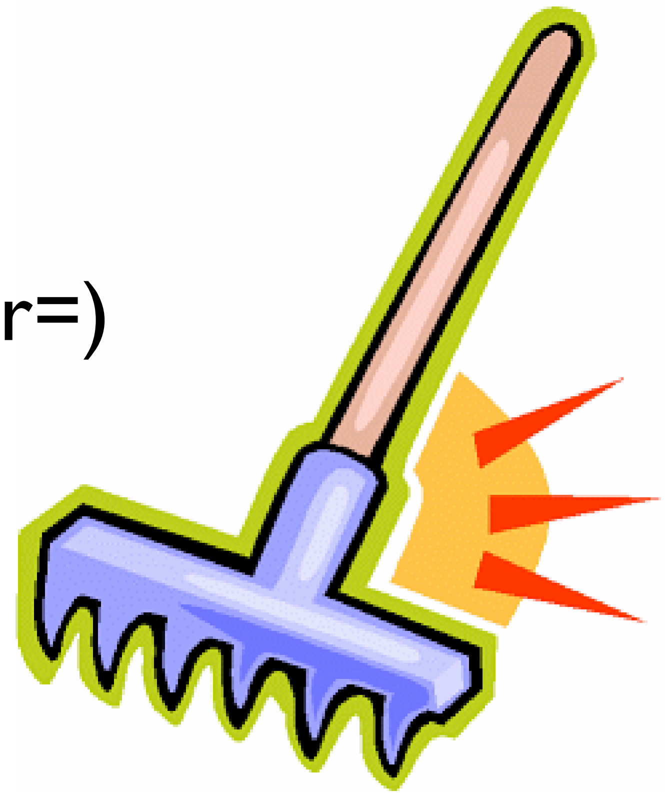
public, followed by base class name

```
class Triangle : public Shape
{
  public:
    Triangle();
    int area();

  private:
    void calc();
    // etc...
};
```

- Base class must already be declared here
- Triangle can have all its own stuff - methods, vars, whatever

Inheritance

- What gets inherited?
  - All member variables, (nearly) all functions
- What does **not** get inherited?
  - constructors and destructors
  - Assignment operators (operator=)
  - Friends



Constructors

- Remember, a constructor gets called for *every* class that gets instantiated
  - Sometimes it's a behind-the-scenes constructor, but there always is one!
- With inheritance, there are (at least) two classes involved: the base class and the derived class
- So, at least two constructors are getting called!

Snippet

```
class Base
{
  public:
    Base()
    { cout << "base\n"; }
};

class Derived : public Base
{
  public:
    Derived()
    { cout << "derived\n"; }
};


int main()
{
   Derived d;
   return 0;
}
```

- What is the output of this program?

Construction Order

```
class Base
{
  public:
    Base()
    { cout << "base\n"; }

   Base( int x )
    { cout << "base 2\n"; }
};

class Derived : public Base
{
  public:
    Derived()
    { cout << "derived\n"; }
};
```

- Base classes will always be constructed *before* any derived classes. (Why?)
- The base class constructor is getting called, even though it's not being called explicitly
- If Base has multiple constructors, which one gets called?

Constructor Init List

- C++ will call the default constructor for any base classes automatically
- If there *is* no default constructor (when would that be?) then we have to explicitly call one
- This requires special syntax called the **constructor init list**.

Constructor Init Lists

```
class Base
{
  public:
    Base()
    { cout << "base\n"; }

   Base( int x )
    { cout << "base 2\n"; }
};

class Derived : public Base
{
  public:
    Derived();
};
```

```
Derived::Derived()
  : Base(5)
{
}
```

Constructor Init List

- The constructor init list lets you pass parameters to the base class constructor
- This is like a function call: it will call the correct overloaded constructor

More CIL

```cpp
class Derived : public Base
{
  public:
    Derived();
  private:
    int x, y;
};
```

```cpp
Derived::Derived()
  : Base(5), x(5), y(18)
{
}
```

- The CIL can be used for regular member variables, too
- Here, x and y are integers being initialized in the Constructor Init List
- This happens before the constructor body executes!

Coding

- Let’s play with inheritance!