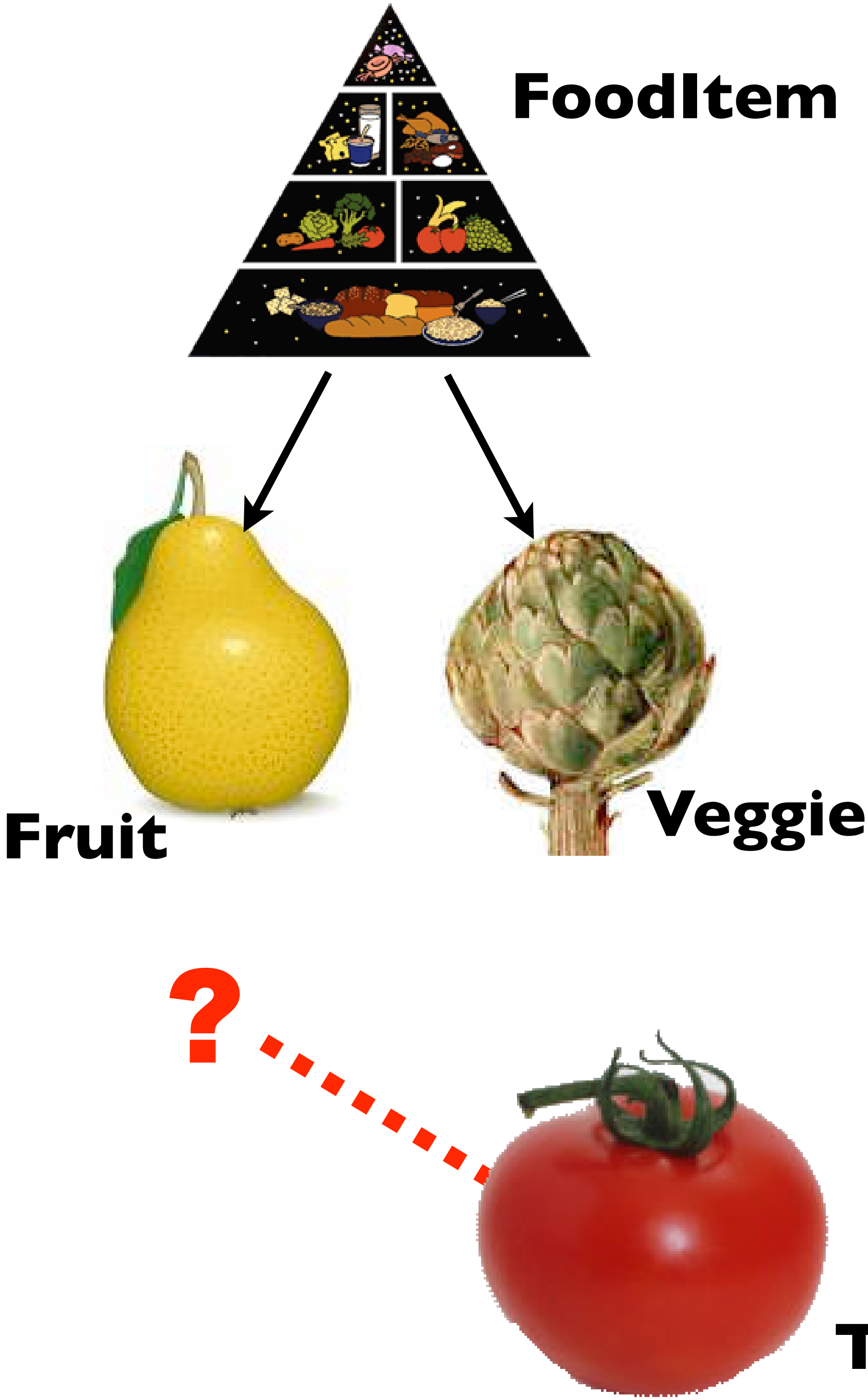
Templates, STL, & Some Conditional Stuff



Nobody seems to be able to agree whether **Tomato** should be derived from **Fruit** or from **Veggie**.

How could we solve this dilemma and make everybody happy?

What would be the problems with doing this, and how might we address those?

Stream Revue

- How do you read in an entire line from cin?
- What object do we use for opening files for output? For input?
- How would we check to see if our output object has any errors?
- What are these operators called? **<< >>**

More Revue

```
class time
{
   private:
      int hour, min;
};
```

```
time u, t;
t = u;
```

- How would we go about fixing this class so we can use cin/cout?
- How would we go about fixing this class so we can use the addition operator on it?
- How would we make this code compile and run properly?

Even More Revue!

```
class time
{
   public:
      void chime();
   protected:
      int hour, min;
};

class secTime : public time
{
   protected:
      int sec;
};
```

```
void doStuff( time t )
{
   // ...
}
```

```
int main()
{
     secTime st;
     doStuff( st );
}
```

Does this code work?
What be wrong with it?

Swappety Swap Swap

- Here we have a perfectly good swap function
- This works really well - as long as we want to only swap *integers*!

```
void swap( int& a, int& b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

- What if we want to swap some floating point values? Will this work?

```
float a = 5.2, b = 7.6;
swap( a, b );
```

bad!

Lots of Overloading

```
void swap( int& a, int& b )
{
    int temp = a;
    a = b;
    b = temp;
}

void swap( float& a, float& b )
{
    float temp = a;
    a = b;
    b = temp;
}

void swap( Cow& a, Cow& b )
{
    Cow temp = a;
    a = b;
    b = temp;
}
```

- Nope.
- To make this work for floating point values, we'd need to write a whole 'nuther function, that does the *exact same thing*!
- Only difference is the type.
- This is kinda dumb.

Intro to Templates

- If the function is **exactly** the same except for the type, we can generalize it so it will work for *any* type!
- This is done via the magical and amazing wonder of **C++ templates!**
- This allows us to write a function *once*, and use it for *any* C++ type - built in type, class, etc.
- This is C++'s implementation of the *generic programming paradigm*

A Generic Swap Function That Doesn't Suck

```
template <class T>
void swap( T& a, T& b )
{
    T temp = a;
    a = b;
    b = temp;
}
```

- Now the compiler will replace **T** with whatever type we want! (int, float, MooCow, etc... whatever)

- This is the exact same thing as the integer version, except:
    - All **int**'s have been replaced with **T**'s
    - There's a new line that declares this to be a template function

Explaining Further

```
template <class T>
void swap( T& a, T& b )
{
    T temp = a;
    a = b;
    b = temp;
}
```

This says the following (single) function is a template function

We can also use the **`typename`** keyword here instead of **`class`** - the two are equivalent

T is conventional, but we can use any name to “rename” the type

Calling It

- Now that we've got this generic swap function, we have to call it
- The function doesn't actually exist until we tell it what type to use
- We do that by appending <type> onto the function name

```
float a = 5, b = 7;
swap<float>( a, b );
```

The type we want the function to swap

Types

```
template <class T, class U>
void swap( T& a, T& b, U& c )
{
    U randomVar = c;
    T temp = a;
    a = b;
    b = temp;
}

int main()
{
    MooCow cow;
    int a = 5, b = 10;

    swap<int,MooCow>(a,b,cow);
    return 0;
}
```

- You're not limited to a single type; template functions can take multiple types!
- This template function takes *two* types
- Could be anything; we're giving it **int** and **MooCow**

Template Classes

- So far today, we've only done template *functions*
- We can template-ize entire *classes* too!
- This is arguably more useful: there are many classes that can be used for many different types!
- Like container classes: stack, queue, binary tree, etc.

The Int Version

```
class array
{
  public:
    int get( int ix );
    void set( int ix, int val );

  private:
     int data[10];
};

int array::get( int ix )
{
   return data[ix];
}

void array::set(int ix, int val)
{
   data[ix] = val;
}
```

- Here's a complete implementation of a simple array class
- It can only use **int**s - that's all it's written for!
- With templates we can make the class generic and reusable!

Class Declaration

```
template <class T>
class array
{
  public:
    T get( int ix );
    void set( int ix, T val );

  private:
     T data[10];
};
```

The template line applies *only* to the single thing (class or function) that follows it!

- This is the template-ized version - changes highlighted in red
- This class will be instantiated with type T - T could be any type!
- So all **int**s have been replaced with **T**s in the class declaration

Class Definition

```
template <class T>
T array<T>::get( int ix )
{
    return data[ix];
}

template <class T>
void array<T>::set(int ix, T val)
{
    data[ix] = val;
}
```

(functions from the template array class)

- *Each* member function in the class needs its own template line (when defined outside the class)
- Also, `array::get()` isn't enough - now we need to use `array<T>::get()`

Instantiating Template Classes

- When you call a template *function*, you pass in the types as part of the *function name*:

```
swap<int>( a, b );
```

- When you instantiate a template *class*, the types become a part of the *class name!*

```
array<float> stuff;
stuff.set( 0, 3.234 );
```

What's Happening?

- Each time you instantiate a template class with a new type (or set of types), the compiler creates an entirely different class!

The compiler will generate a different set of code for:

```
array<float>
```

... than it will for:

```
array<MooCow>
```

Non-Type Parameters

```
template <class T, int N>
void func( T& a )
{
    T bob = N*2;
    a = bob;
}
```

```
int var;
func<int, 17>( var );
```

- Templates can also be declared with *non-type* parameters: just regular types, like an integer
- In this example:
  - every **T** will be replaced by **int**
  - every **N** will be replaced by **17**

Default Template Values

```
void print( char* s, int repeat = 0 )
```

- In this normal function, if we don't supply a value for the repeat function, it's automatically set to 0.
- We can do the same sort of thing with templates:

```
template <class T=int, int N=23>
void func( T& a )
{
    T bob = N*2;
    a = bob;
}
```

- If we don't supply types to func(), they get set to **int** and **23**

```
int bob;
func<>( bob );
```

One Issue:

- Normally when we're designing big classes, we try and keep the definition separate from the declaration
  - Helps things compile faster!
  - Easier to deal with
- Since templates are compiled "on-demand", the *entire class* has to be in the same file.
  - This is usually a header file

- **Coding**: let's take the simple myArray class we made earlier, and turn it into a template class

Intro to the STL

- In the C language, if you wanted a data structure, you had to write it yourself
  - This was a pain
- With C++ and templates, we can create a generic library of data structures and routines that apply to nearly any data type
- There's a standard one called the **STL**: **S**tandard **T**emplate **L**ibrary

Stuff in the STL

- The STL contains a bunch of different data structures for your use: vector, list, deque, set, map, hash_set, etc.
- There are also implementations of algorithms that operate on these data structures (sorting, etc)
- The STL is very large and complicated - we're only going to cover some of the basics here
- STL can be hard to debug - check out the kinds of error messages you can get!

```
stl_algo.h: In function `void __merge_sort_loop<_List_iterator <int,int &,int *>,
int *, int>(_List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>, int
*, int)': instantiated from `__merge_sort_with_buffer <_List_iterator<int,int
&,int *>, int *, int>( _List_iterator<int,int &,int *>, _List_iterator<int,int
&,int *>, int *, int *)' instantiated from `__stable_sort_adaptive<
_List_iterator<int,int &,int *>, int *, int>(_List_iterator <int,int &,int *>,
_List_iterator<int,int &,int *>, int *, int)' instantiated from here no match for
`_List_iterator<int,int &,int *> & - _List_iterator<int,int &,int *> &'
```

STL Containers

- STL provides a bunch of **container types**: objects that contain other objects
- For example: the STL **vector** class behaves much like an array, but it handles all the memory management for you, and can grow itself as necessary
- **vector** is (duh) a template class, so you get to tell the compiler what type the vector holds:

```
vector<int> bunchOfInts;
```

- Here’s a simple example of the vector class in action:

```
#include <vector>
using namespace std;

vector<int> vec;  // or std::vector
int a = 2;
int b = -5;

vec.push_back(a);
vec.push_back(9);
vec.push_back(b);

for( int i = 0; i < vec.size(); i++ )
{
    cout << vec[i] << endl;
}
```

STL With Custom Classes

- STL containers work fine with built-in types, but to use them with custom classes, the class need to have these things defined:
  - default constructor
  - copy constructor
  - assignment operator
  - operator< (sometimes)
  - operator== (sometimes)

find() in STL Containers

- Most STL containers support the find() function, which lets you search for a value
- But what should find() return?
- A position/index would be OK for a vector, but wouldn't work so well for something like a set, which has no inherent order!
- Instead, STL uses **iterators** - small C++ objects that work like intelligent pointers
- So find() returns an iterator that points to the found value

Iterators

- Example: vector (again)

```
vector<int> vec;
vector<int>::iterator iter;

... // put stuff in the vector

for( iter = vec.begin(); iter != vec.end(); iter++ )
{
    cout << *iter << endl;
}
```

the iterator type for each STL class is declared *in* the class!

- We're using an iterator like we would a pointer!
- This is the "standard" way to traverse through an STL container

Using the find() function

- The find() function doesn't deal with a container (like a vector or a list) - it deals entirely with iterators

```
vector<int> vec;
vector<int>::iterator iter;

iter = find( vec.begin(), vec.end(), 42 );
```

starting iterator of the range we're searching in

ending iterator of the range we're searching in

value we're searching for - what type is this? (In general, that is)

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;
```

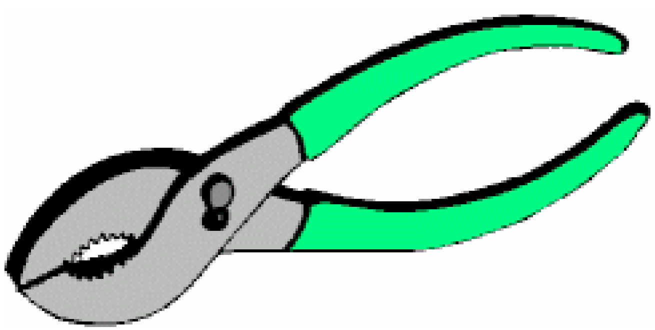
Anatomy of the Ternary Operator

```
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



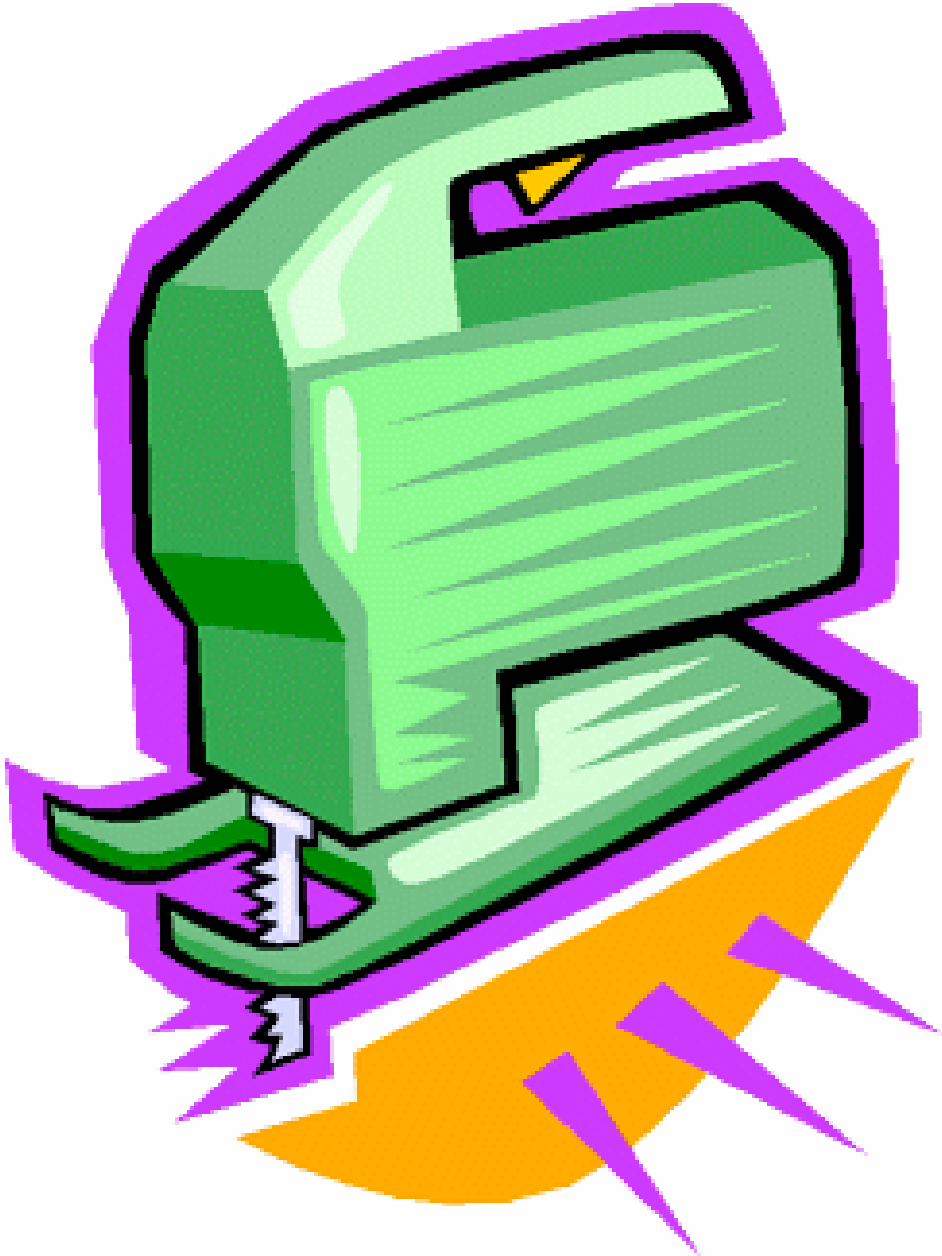
Usages

- 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

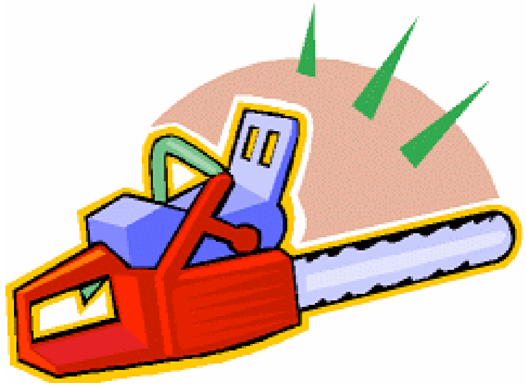
switch keyword

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