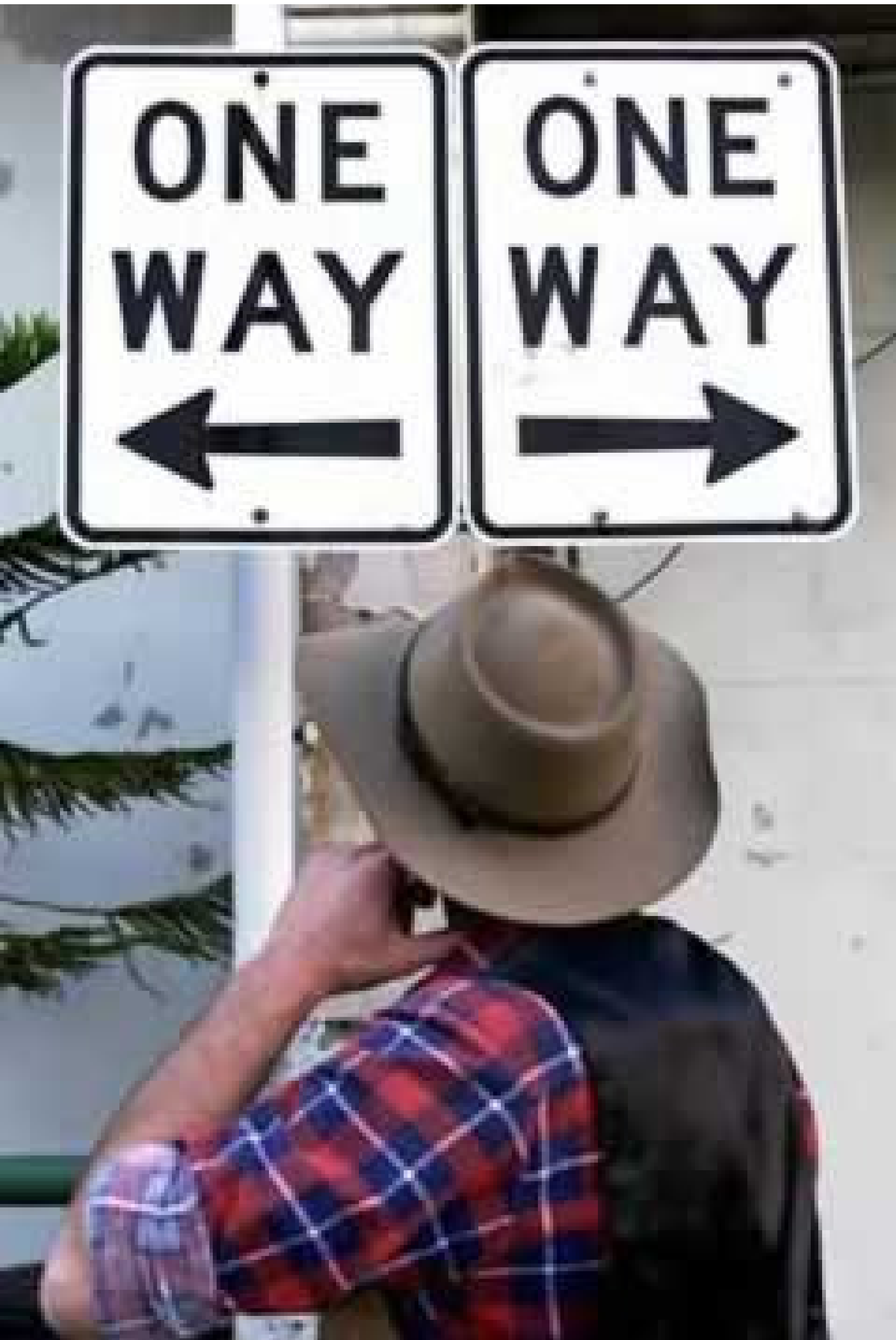


Operator Overloading, Again, And Conversion Operators

Problems with Pointers

- Problems with pointers:
  - What are they pointing to? Can you be sure it’s anything useful?
  - Dereferencing a NULL pointer causes problems
  - Dereferencing a “wrong” pointer also causes problems
  - What happens with uninitialized pointers?
  - All that funny syntax to deal with!

Introducing References! (again!)

- **References** are a C++ feature to deal with some of those issues
- A reference variable links to another variable:

```
int normalVariable = 42;
int& reference = normalVariable;

cout << normalVariable << endl;
cout << reference << endl;
```

Here, reference is linked to `normalVariable`! `reference` doesn't have its own memory location – it just uses `normalVariable`'s

Declaring Reference Variables

- A reference variable is declared by sticking an ampersand (&) after the type:

```
int normalVariable = 42;
int& reference = normalVariable;
```

- The same rules apply as for pointers: the "&" only applies to the *first* name to follow it

```
int bob = 42;
int& a = bob, b = bob;
```

- In this example, "a" is a reference – b is *not* a reference, but instead a *copy* of bob

More Reference Declaration Stuff

- Also like pointers, the spacing around the & doesn't matter:

```
int &a = bob;     // same-same
int& b = bob;
```

- There can be an unlimited number of references to a "normal" variable:

```
int normalVariable = 42;
int& a = normalVariable;
int& b = normalVariable;
int& c = normalVariable;
```

Using Reference Variables

- There is **no difference** between reference variables and regular variables when it comes to usage!

```
int normalVariable = 42;
int& reference = normalVariable;

reference++;
normalVariable++;
```

Reference Rules

- A reference variable:
  - *must* be initialized to another variable
  - *can't* be changed after initialization
    - there's no syntax for doing this!
  - can never be NULL
    - … so no need to worry about dereferencing a NULL pointer
    - Why can a reference never be NULL?

Things to Remember…

- Remember: pointers contain an address!
  - There’s a difference between changing the pointer’s address and changing the value of what it points to
- Reference variables hide this from you
  - With a reference, you *can’t* change the address or what it points to (no pointer arithmetic)
  - You can think of a reference variable as another way to access whatever variable it links to

Not Exactly New

- We’ve seen reference variables before:

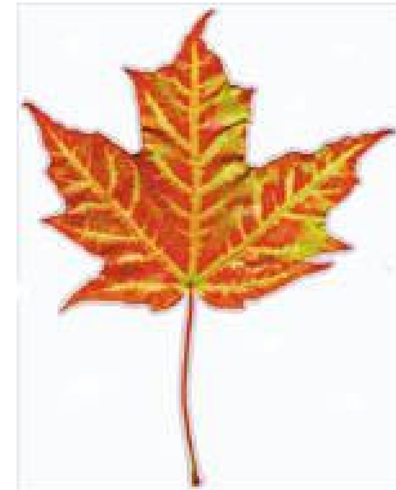
```
void swap( int& a, int& b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

- So what’s actually going on here?

Returning References

- A reference is a type (just like any other variable) and can be returned from a function:

```
int& exampleFunction()
{
    int variable = 10;
    return variable;
}
```



- This is valid syntax, but it has a problem – what do we need to be careful of when returning a reference?

So…

- What’s *good* about references?
- What’s *not so good* about references?
- When would you use a reference ?
- When would you use a pointer?

Time for ***review!!!***

- What’s the difference between a regular constructor and a copy constructor?
- What’s the difference between a copy constructor and operator=?
- What is operator overloading?
- How do you overload an operator?

Question

```
void doStuff( int x )
{
    cout << x << endl;
}
```

- Say we've got a very simple doStuff function…

- Can we do this?

```
float bob = 5.2;
doStuff( bob );
```

- Why does this work?

Question

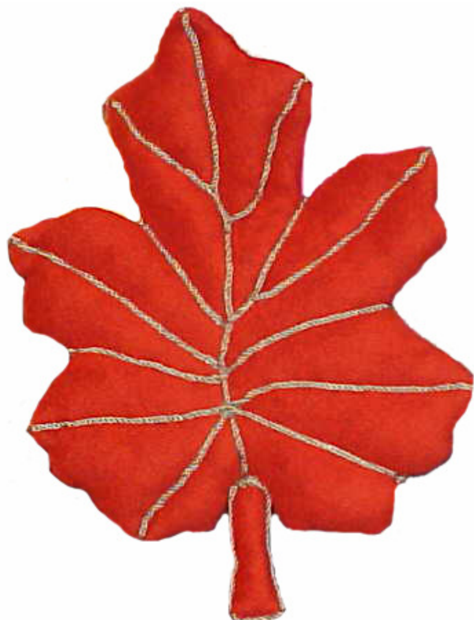
```
class Complex
{
public:
    Complex();

private:
    float real, imag;
};
```

- Say we’ve got a very simple Complex class…

- Can we do this?

```
Complex number;
char whatever[100];
strcpy( whatever, number );
```

- Why would this **not** work?



Type Conversions

- Remember this stuff?

```
float bob = 5.2;

// implicit type conversion
doStuff( bob );

// explicit type conversion
doStuff( (int)bob );
```

- Whether explicitly or implicitly, C++ will convert types when it can
- We can add this functionality to classes, too!

Conversion Operators

```
class Complex
{
public:
   Complex();
   operator int();

private:
   float real, imag;
};

Complex::operator int()
{
   return (int)real;
}
```

- The **operator int()** function automatically gets called when you try to convert the code to an integer
- This means you can use Complex anywhere you'd use an integer – Complex gets automatically converted to an int

Anatomy of a Conversion Operator

no return type (why?)

type this function converts to

operator keyword

```
Complex::operator int()
{
    return (int)real;
}
```

parenthesis close out the function signature

Finish the Example

```
class Complex
{
public:
   Complex();
   operator ???();

private:
   float real, imag;
};

Complex::operator ??()
{
   return ??;
}
```

- Let's say we wanted to do a string conversion operator:

```
Complex number;
cout << number << endl;

// this should print out
// the word "hello"
```

- How would we do that?

Let’s Write Some Code…

A simple string class!

This will tie together a lot of the concepts we’ve talked about so far