more stuff about

FUNCTIONS

than you ever wanted to know

Quickie Review

- How do you declare a function that takes 3 integer parameters and returns nothing?
- How do you call that function?
- What would happen if you tried to call that function using a floating point value as input?
- Write a loop that counts from 0 to 20, but only using even numbers.
- Write some code that gets a number between 1 and 100 from the user and validates it.

Question:

```
int main()
{
   cout << meaningOfLife() << endl;
   return EXIT_SUCCESS;
}

int meaningOfLife()
{
   return 42;
}
```

Will this work? Why or why not?

Nope.

```
int main()
{
     cout << meaningOfLife()
     << endl;
     return EXIT_SUCCESS;
}

int meaningOfLife()
{
     return 42;
}
```

compiler output:

```
prototype.cpp: In function `int main()':
prototype.cpp:8: `meaningOfLife' undeclared (first use this function)
prototype.cpp:8: (Each undeclared identifier is reported only once for each
function it appears in.)
prototype.cpp: In function `int meaningOfLife()':prototype.cpp:13: `int
meaningOfLife()' used prior to declaration
```

C++ files are compiled from top-to-bottom; the compiler doesn't "know" about meaningOfLife() because it hasn't "seen" it yet.

Function Prototypes

- Functions need to be either defined above the point at which they are called, or...
- There needs to be a **function prototype** above where that function is called.
- A function prototype is identical to the first line in the function body... just without a body, and followed by a semicolon.

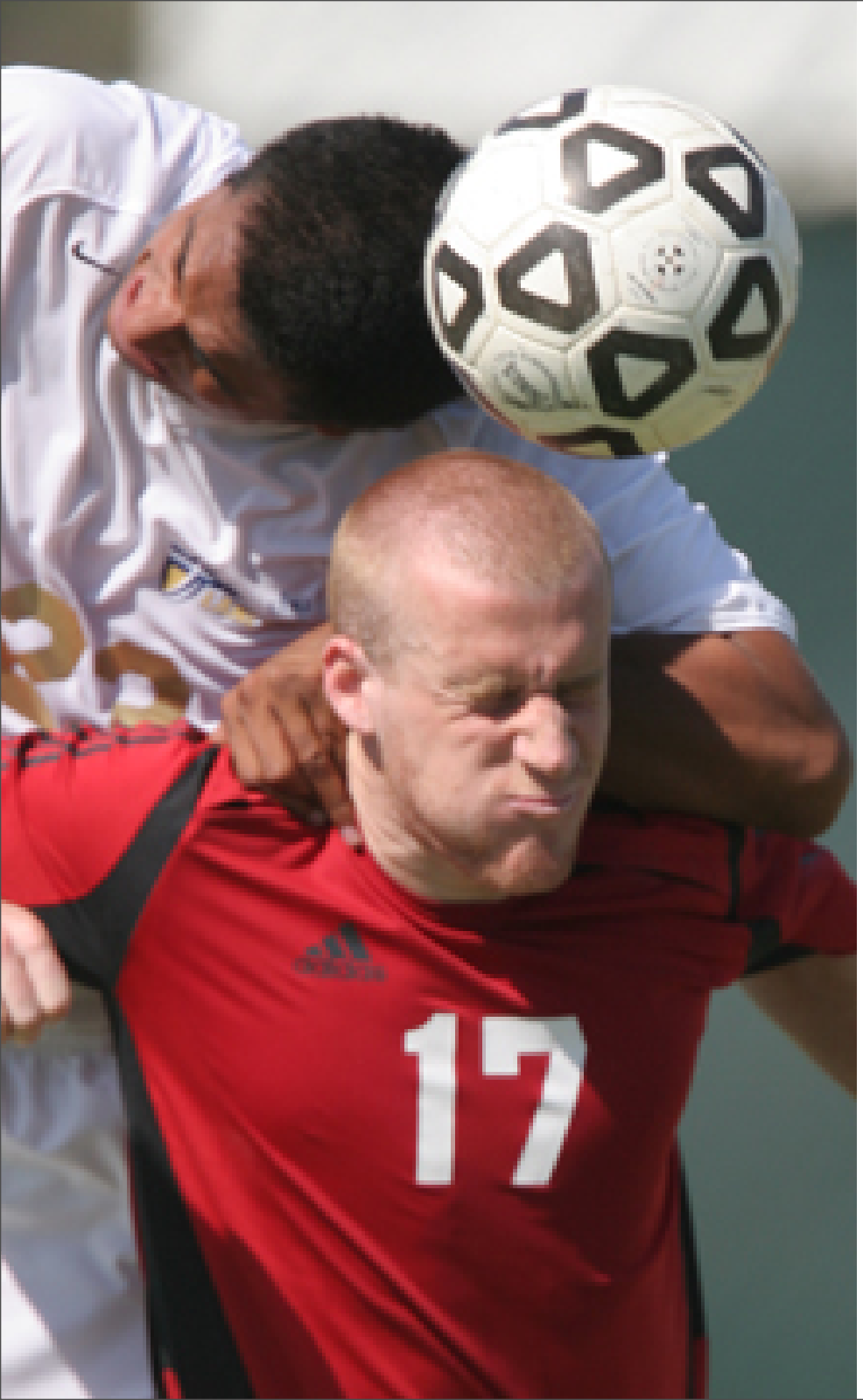
```
int meaningOfLife();
```

```
int meaningOfLife( bool isFun, int, int );  // prototype

int main()
{
    cout << meaningOfLife() << endl;
    return EXIT_SUCCESS;
}

int meaningOfLife( bool isExciting, int b, int c )
{
    return 42;
}
```

- A prototype requires a return value, a name, and argument types. It can also have argument names - these are optional.
- The argument names can be *different* than those used in the function.
- Everything else must be exactly the same!

Uses of Prototypes

- The compiler uses prototypes to validate function calls without needing to have the actual function around
- Before a function call can be compiled, the compiler needs to know that it has the appropriate function:
  - correct name
  - correct argument types (by type conversion if necessary)



Header Files

- Many, many function prototypes live in header files that are `#include`-d, like `<iostream>`
- The actual code for these functions are in other files, or in libraries that will be linked into the executable
- We'll cover how to do this later. Probably.

Quizlet

```
void increment( int );

int main()
{
    int var = 5;
    increment( var );
    cout << var << endl;
}

void increment( int x )
{
    x++;
}
```

- Does this compile?
- If so, what is the output?

Pass by Value

```
void increment( int );

int main()
{
    int var = 5;
    increment( var );
    cout << var << endl;
}

void increment( int x )
{
    x++;
}
```

- Default method of passing arguments is pass-by-value.
- This means that copies get made of each argument, and the function manipulates its own copies - as if they were local variables.
- What happens to the copies of the parameters when the function ends?

Pass by Value

```
void swap( int x, int y )
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

will this work?

- What happens to the copies of the parameters when the function ends?
  - They get discarded!
  - Any changes that were made to those variables are lost.
- What if you want a function to change the values of its parameters?

Pass by Reference

- An alternative is pass-by-reference, in which you pass a **reference** to the variable
- Then the function will manipulate the variable itself, not a copy (as in pass-by-value)
- Any changes to the variable will "stick"

references are denoted by an **&** between the type and the parameter name

```
void swap( int& x, int &y )
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

References and Function Prototypes

```
void increment( int );

int main()
{
    int var = 5;
    increment( var );
    cout << var << endl;
}

void increment( int& x )
{
    x++;
}
```

will this compile?

- The prototype and the function still have to match...
- ... including references!

Passing parameters by reference

```
void increment( int& );

int main()
{
    int var = 5;
    increment( var );
    cout << var << endl;
}

void increment( int& x )
{
    x++;
}
```

When looking at the function call, parameters passed by reference look exactly like those passed by value.

```
void doStuff( int& foo, int& baz, int reep )
{
    foo = 4;
    baz = foo * reep;
    foo++;
}
```

```
int phooey = 1, gazonk = 2;
doStuff( phooey, gazonk, 2 )
```

```
int phooey = 1, gazonk = 2;
doStuff( phooey, phooey, 2 )
```

```
int phooey = 1, gazonk = 2;
doStuff( phooey, 2, gazonk )
```

Are all of these examples valid?

Why or why not?

Passing by Reference

- When is pass-by-reference a good idea?
- Why should you be careful when using pass-by-reference?
- What side-effects does it have?

Default Arguments

- This is a nifty way to specify defaults for some (or all) arguments to a function
- When you're calling that function, you don't have to specify every argument if there is a default
- Very handy, very widely used

Default Arguments Example

```
void printLetterOnScreen( char letter,
                          int xPos = 10, int yPos = 10,
                          int repeatCnt = 1 )
{
    // do stuff
}
```

These are all valid ways to call this function:

```
printLetterOnScreen( 'g' );

printLetterOnScreen( 'p', 15 );

printLetterOnScreen( 'w', 15, 42 );

printLetterOnScreen( 'x', 15, 42, 5 );
```

Default Arguments Example

```
void printLetterOnScreen( char letter,
                          int xPos = 10, int yPos = 10,
                          int repeatCnt = 1 )
{
    // do stuff
}
```

- Only *trailing* arguments can have default values
  - If a argument has a default, *all* of the following arguments also need them
- When calling a function, "skipping" arguments is illegal

```
printLetterOnScreen( 'p', 15 );
```

15 will be the value of xPos, not yPos or repeatCnt

Default Arguments and Function Prototypes

- By convention, default arguments usually go in the the function prototype
- They can also be put in the function definition itself - but *not* in both places
  - some compilers allow this, as long as the default arguments match - g++ doesn't

Function Overloading

- Don't be fooled by the scary-sounding name: function overloading is a *good* thing!
- The idea: multiple functions can be defined with the same name
- The compiler will automagically pick which function to call, based on the number and type of arguments

overloading examples

which function gets called?

```
void blegh( char letter )
{
}

void blegh( char letter, int reps )
{
}

void blegh( int number )
{
}

void blegh( float realNum )
{
}

void blegh( bool maybe )
{
}
```

```
blegh( 25 );

blegh( 'a' );

blegh( false );

blegh( 'q', 5 );

blegh( 5 > 2 );

blegh( 97, 5 );

blegh( 32.0 );
```

Ambiguity

- When the compiler can't figure out which version of an overloaded function to call, the function is said to be **ambiguous**
- This isn't always obvious, as you saw with the 32.0
- The previous example, now with a default parameter:

```
void blegh( char letter )
{
}

void blegh( char letter, int reps = 0 )
{
}
```

`blegh( 'a' );`
goes to which function?

These are ambiguous, so you get a compiler error

Overloading and return types

- Overloaded functions need to have differing *parameters* - different *return types* is not enough

```
int doStuff()
{
    // ...
}
```

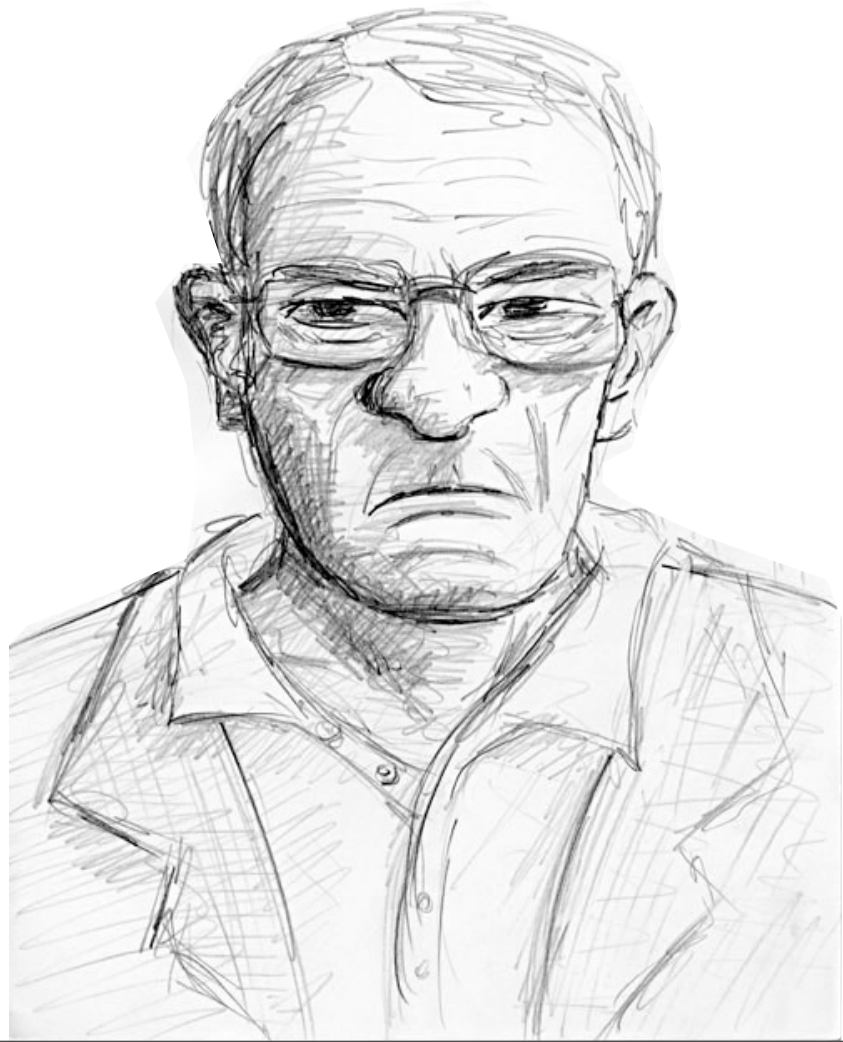
```
double doStuff()
{
    // ...
}
```

- This will cause a compiler error
- Why do you think this is?

More in-class Coding!

whoooo!

- Let's define some print functions that can print out different variable types, and at different positions.



Project 1:
Palindromic Numbers

- Project One: now available on the class website
- Due: next Friday, September 8, at 11:59 PM (electronically submitted)

Palindromic Numbers

- Palindromic numbers read the same front-to-back and back-to-front
  - e.g., 12321, 99, 1221, etc.
- Algorithm to generate a P.N. from an integer:
  - Reverse the number
  - Add the reversed number to the original number to get a new number
  - If you've made a palindrome, great! Otherwise repeat this process using the new number
- This works for most - not all - positive integers

Project One:

- Get (and validate) a starting and ending number from the user, between 10 and 1,000 (why?)
- For each number between the starting and ending numbers (inclusive), find out if that integer can be used to generate a palindrome in <= 12 steps
- If yes: print the number, the palindrome, and the number of steps it took
- If no: print the number and a message saying that no palindrome could be generated.

What to do:

- Write, debug, and test your code.
- Write a README file with:
  - your name
  - compilation instructions (include the exact command you used to compile)
  - the amount of time you spent on this project
  - anything notes you'd to include (in particular, anything you'd like me to know when grading)
- Submit a directory containing your README and code using the CS dept's submit procedure (check the web site)

Thoughts

- Be sure to read the actual assignment (posted on the website)
- This isn't a hard assignment, but there's some tricky steps in here.
- What are they?
- What are the individual "chunks" of code you could write and test individually?
- How will you structure your program to make it clean and readable?