LOOPS AND FUNCTIONS

Quick Review...

- variables, data types, expressions
    - declare an unsigned integer named **score**
    - declare a double-precision floating-point value named **distance** with an initial value of 43.523
- conditional statements
- print "hello" if both of these are true:
    - **distance** is less than 25.0
    - a bool named **running** is true or **score** is greater than 100

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

int main()
{
   int gazonk, foo = 2 * 5;
   int baz = 10 - foo;

   if( baz )
      if( foo )
         cout << "Alpha" << endl;
   else
      cout << "Beta" << endl;

   cout << "gazonk: " << gazonk << endl;

   return 0;
}
```

Quick Review 2...

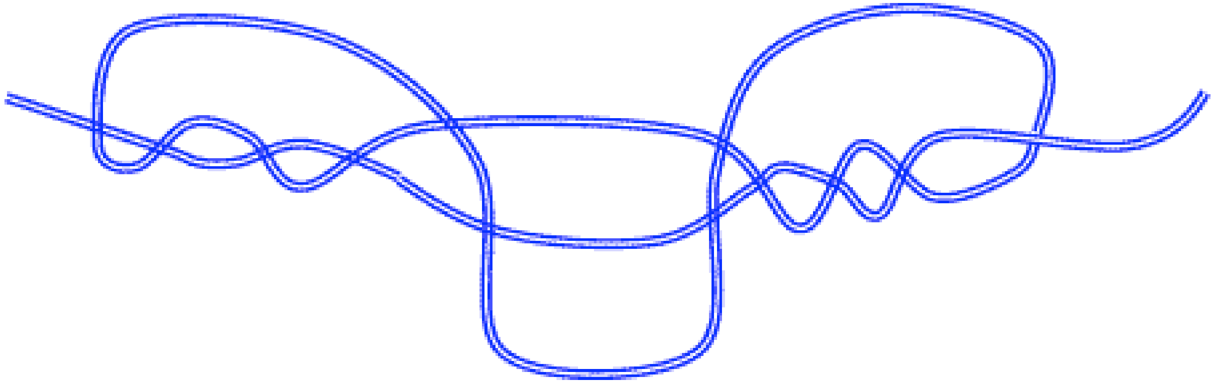
What is the output of this program?

Quick Review 3...

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

int main()
{
    double foo = (2.0 * 5.0) / 1.0;
    int baz = 10 - foo;

    if( baz )
        cout << "Alpha" << endl;

    return 0;
}
```

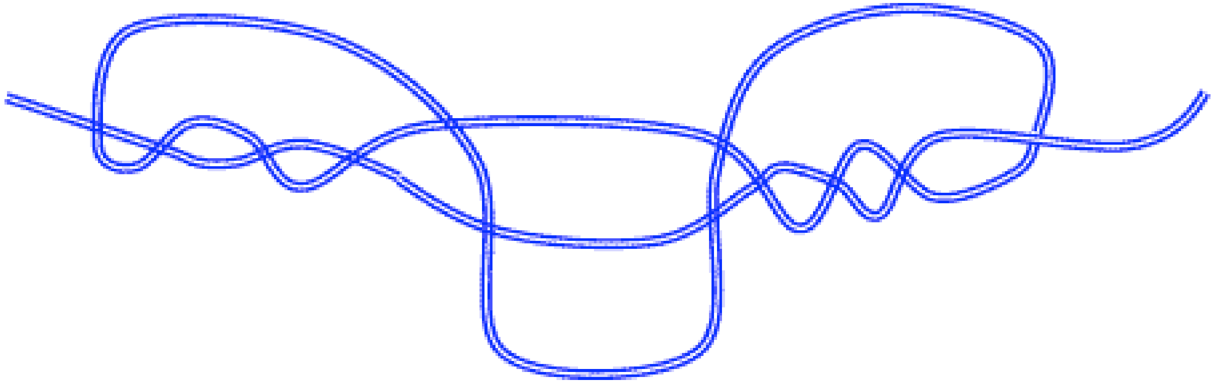
What is the output of this program?

LOOPS



- Computers are very good at doing repetitive tasks
- Loops aid in doing repetitive work
- Nearly all complex programs will have loops

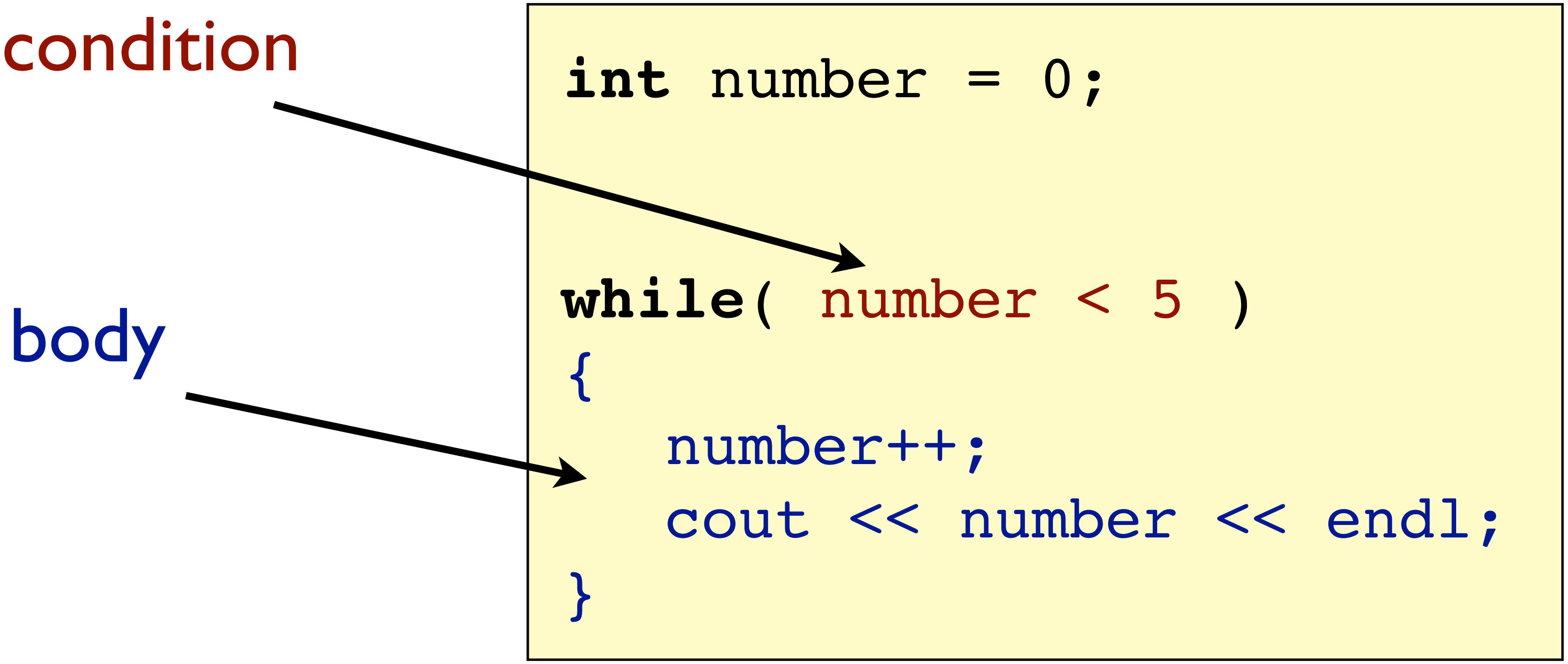
LOOPS



- C++ has three kinds of loops:
  - for loop
  - while loop
  - do-while loop
- Each of these work kind of like the if statement: they execute the single statement (or block of statements) that follows them

`while` Loop

- Condition is checked at the beginning of every iteration of the loop
- If the condition evaluates to true, the body of the loop is executed



condition

body

```
int number = 0;


while( number < 5 )
{
   number++;
   cout << number << endl;
}
```

`while` Loop

- One way to think of this:
  - syntax and operation of a while loop is the same as a for loop...
  - ... except it will execute the body *until* the condition is true
- Again, watch out for stuff like this:

```
while( tired );
   sleep();
```

`do-while` Loops

- A while loop checks the condition *before* every iteration of the loop
  - so if the condition is never true, the loop will never execute
- A **do-while** loop checks the condition at the *end* of every iteration
  - side-effect: the body of the loop will always execute at least once, even if the condition is never true

Anatomy of a do-while

```
int number = 0;

do
{
    number++;
    cout << number << endl;
}
while( number < 5 );
```

do keyword
comes immediately before the body of the loop

body
again, single statement or block of statements

condition
checked *after* each iteration of the loop has executed

semicolon
the while is at the *end* of the loop, so it must be terminated by a semicolon

The **`for`** loop

- C-style for-loops are used in C, C++, Java, Perl, PHP, and a bunch of others
- The `for`-mat (heh heh) of a `for` loop:

```
for( initialization; condition; update )
{
  // body of loop
}
```

initialization: Executed first - just once. Used to setup any counter variables used in the loop.
ex: int i = 0; w = 4;

condition: Just like a `while`, `do-while`, or `if`. Checked *before* every iteration, as in a `while` loop.
ex: i < 20; w != 8;

update: Executed *after* each iteration, used to update variables (increment, decrement, etc).
ex: i++; q += 4; k *= 5

```
for( initialization; condition; update )
{
   // body of loop
}
```

FOR-LOOP EXAMPLES

```
for (int i=0; i<10; i++)
{
   cout << “i = ” << i << endl;
}
```

```
for (int i=0; i<=10; i++)
{
   cout << “i = ” << i << endl;
}
```

```
char i;
for (i =‘a’; i<=‘z’; i++)
{
   cout << “i =“ << i << endl;
}
```

does this work?
why or why not?

```
for(char i=‘z’; i>=‘a’; i--)
{
   cout << “i = “ << i << endl;
}
```

what kind of loop would you use for...

- Printing out every even number between 0 and 100?
- Getting input from the user and making sure it is valid?
- Waiting for the time to be 10:00 AM before continuing?

Infinite Loops

- An infinite loop is a loop where the "condition" is always true, so the loop can never terminate
- Be careful of these!

```
for( ; ; )
{
}
```

```
int i = 0;
while( i < 10 )
{
}
```

```
while( true )
{
}
```

give me a `break;`

- The break keyword breaks out of the current loop
- breaks out of the *current* loop only
- any problems with this?

```
while( true )
{
    while( true )
    {
        if( rand() % 10 == 5 )
            break;
    }
}
```

give me a

break;

`break` is useful but a bit ugly - it is usually a bit more elegant to rewrite the loop condition instead.

How could we rewrite this?

```
// class algorithm
while( !classOver )
{
   stareAtClock();

   if( reallyBored )
      break;
}

doFunStuff();
```

continue;

...skips the rest of the loop body and moves straight onto the next iteration.

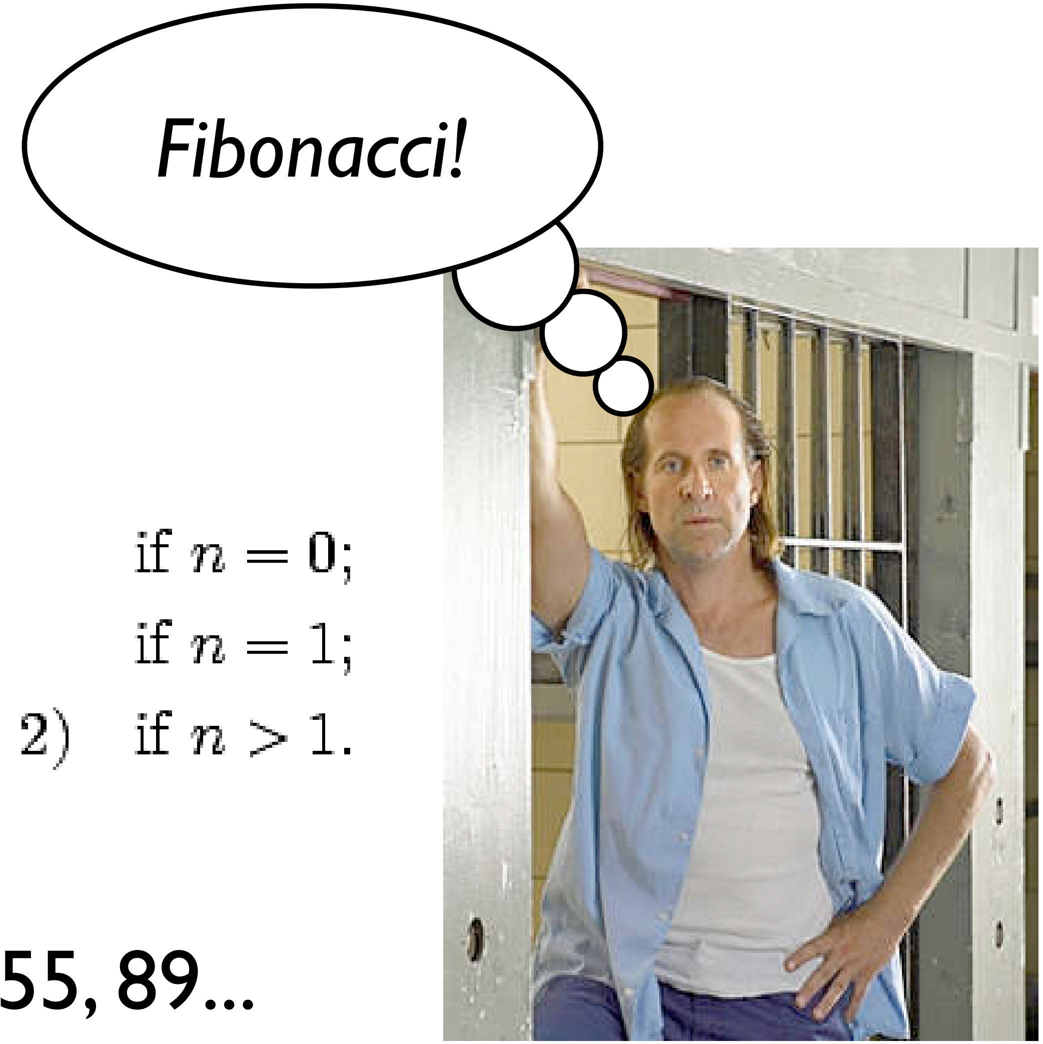
```
// print grades
for( int i = 0; i < numStudents; i++ )
{
   if( student[i].droppedClass )
      continue;

   cout << student[i].name << endl;
   cout << student[i].grade << endl;
   cout << student[i].classRank << endl;
}
```

In-class programming exercise

Let's make a program that prints out the Fibonacci sequence.

$$F_n := F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...



Functions

- Functions are a way to group chunks of code together so they can be reused later
  - ... otherwise you end up with huge, hard-to-maintain chunks of code
- Enables you to structure a program in a more modular way
- Functions in programming are similar to functions in mathematics.

Functions, cont.

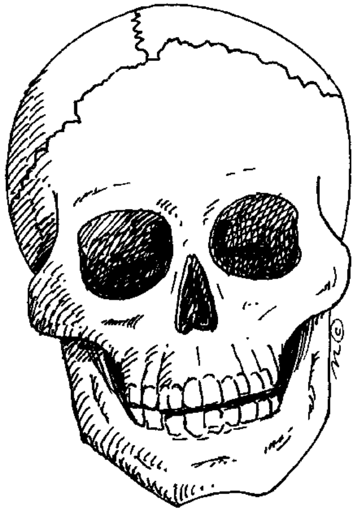
- Each function has its own code - just like the code in the main function
- Each function can access its own variables, but *not* the variables from any other function
- Functions can also access *global variables* - variables declared outside of any function, including the main function

Function Calls

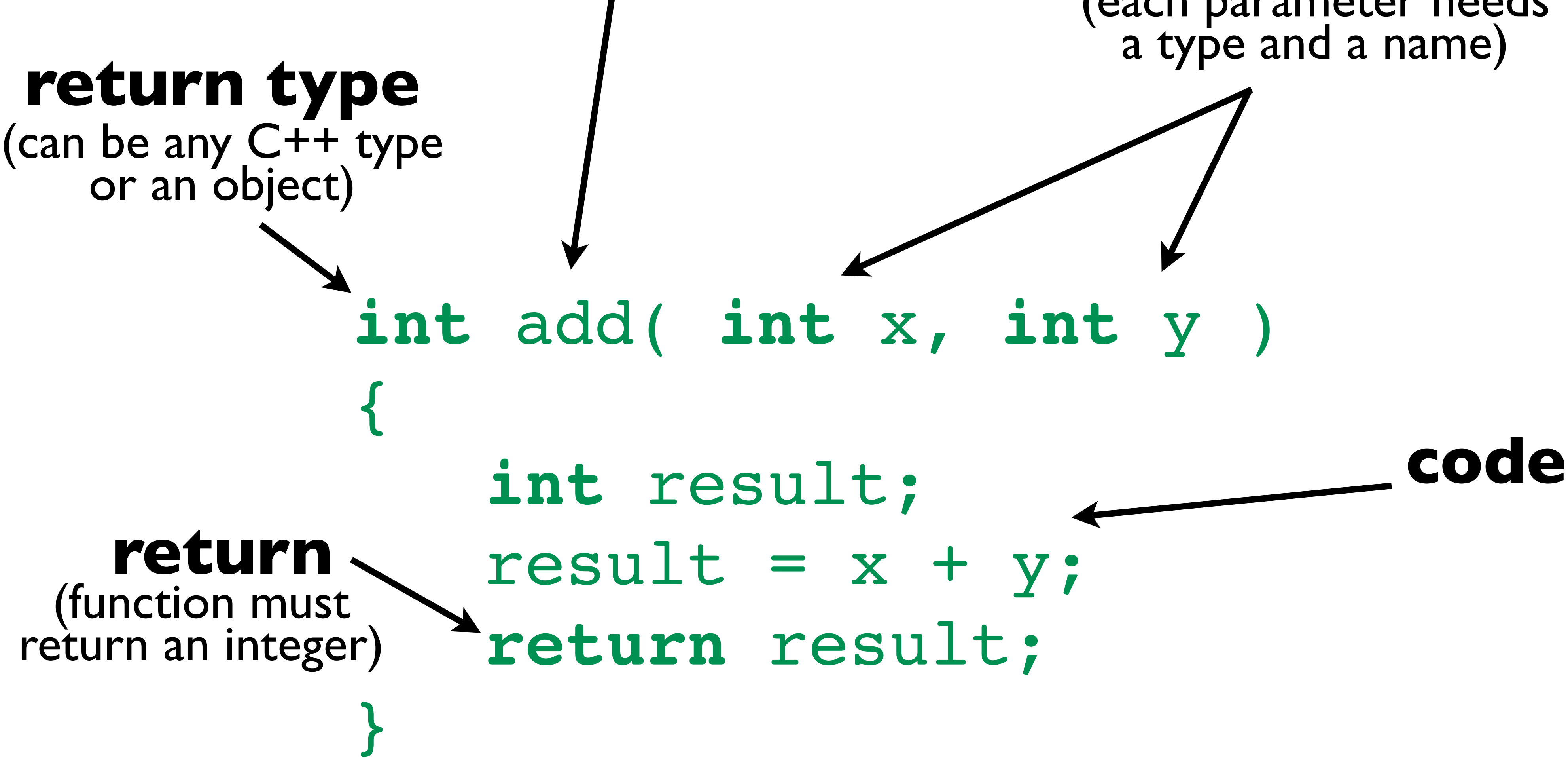
- Function calls cause the following to happen:
  - The currently executing function is suspended
  - Program control is passed to the the function being invoked
  - When the function has finished executing, the suspended calling function resumes execution

useless example!

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

int timesTwo( int input )
{
   int output = input * 2;
   return output;
}

int main()
{
   cout << “two times two is: “
        << timesTwo(2) << endl;

   return EXIT_SUCCESS;
}
```



Anatomy of a Function



function name

parameters
(each parameter needs
a type and a name)

return type
(can be any C++ type
or an object)

```
int add( int x, int y )
{
    int result;
    result = x + y;
    return result;
}
```

code

return
(function must
return an integer)

function return types

- A function has to have *some* return type declared
- Return types can be any basic C++ data type
- Can also be any object type (that bit comes later)
- If a function doesn't return a type, the return type is **void**
  - with a void return type, returning anything causes a compiler error

Function Parameters

- Parameters are how we provide input to the function (return value is the output)
- Each parameter has a type and a name... no two names can be the same.   (why not?)

```
int add( int x, int y )
{
}
```

`int x, int y` are the parameters, indicating this function will need to be called with two integers as input.

How to call functions

- You call a function using its name, followed by the parameters in parenthesis, separated by commas

```
int max = maximum( 3, 50 );
```

- Even if a function has *no* parameters, you still need to follow the function name with ()

```
int ans = UltimateAnswer();
```

- The compiler makes sure you call a function with the correct number of arguments:

```
int max = maximum();
```

```
simpleinterest.cpp:6: too few arguments to function `int maximum(int, int)
```

- The compiler also performs type-checking on the different arguments.

```
float var1 = 12.3;
float var2 = 10.5;
int max = maximum( var1, var2 );
```

```
simpleinterest.cpp:35: warning: passing `float' for argument passing 1 of `int maximum (int, int)
simpleinterest.cpp:35: warning: passing `float' for argument passing 2 of `int maximum (int, int)
```

Why is this a warning and not an error?

quick detour: type conversion

- Often the compiler can automatically convert one type to another - this is called an ***implicit type conversion***
- When this can be done without losing data, the compiler will usually just do it quietly
    - int to float: 32 becomes 32.0, etc.
- Some types can be converted but not without changing the value
    - float to int: 56.8 gets truncated; becomes 56
    - The compiler will issue a warning here

quick detour: type conversion

- You can also do an ***explicit type conversion***, in which you force the compiler to convert the type, regardless of consequences

```
float baz = 38.6;

// these are all equivalent
int foo = (int)baz;
int foo = (int)(baz);
int foo = int(baz);
```

- This lets the compiler know that the conversion was intended, and usually makes the warnings go away

Question:

```
int mystery( int x, int y, int z)
{
   int value = x;

   if( y > value )
      value = y;

   if( z > value )
      value = z;

   return value;
}
```

what is the output the following statement?

```
cout << mystery(6, 2, 5) << endl;
```

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?

(Another) Question:

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

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

Will this work? Why or why not?

Answer: No.

```
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!

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”

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

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

References and Function Prototypes

```
void increment( int );

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

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

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

will this compile?

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

```
void blegh( char letter )
{
}

void blegh( char letter, int reps )
{
}

void blegh( int number )
{
}

void blegh( float realNum )
{
}

void blegh( bool maybe )
{
}
```

which function gets called?

```
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.