FUNCTIONS

OVERVIEW

OVERVIEW

- **In real life, we often find ourselves doing the same task over and over**
  - Example: make toast and jam for breakfast in the morning
  - Example: drive our car from home to work and back

- **Once we have decided on our favorite way to do these tasks, we can write down the steps we normally take**
  - Make a recipe card for making breakfast items
  - Write down series of turns to travel from A to B
  - This written descriptions will let others reuse our work

OVERVIEW

- **Functions in C++ can be used much in the same way**
  - We write down the code we want to reuse when we declare the function
  - Then to execute this code, we call the function from the main program

- **This approach is a big time saver for programmers because they can focus on implementing and debugging smaller pieces of code, and then reuse this work**

OVERVIEW

- **Functions play an important role in software development**
  - Break a big problem down into a sequence of smaller problems that we know how to solve
  - Implement and test solutions to each of the smaller problems using functions
  - Use the functions we created above to create an effective solution to the big problem

- **Functions in the same category are often packaged together to create function libraries**
  - C++ has created function libraries for input/output, string manipulation, mathematics, random numbers, etc.

OVERVIEW

- **Lesson objectives:**
  - Learn the syntax for declaring functions in C++
  - Learn how to call and trace the execution of functions
  - Learn how to pass value and reference parameters
  - Study example programs showing their use
  - Complete online labs on functions
  - Complete programming project using functions

FUNCTIONS

PART 1

DECLARING FUNCTIONS

DECLARING FUNCTIONS

- **In order to declare a function in C++ we need to provide the following information:**
  - The name of the function
  - List of operations to be performed
  - Type of data value to be returned
  - Types and names of parameters (if any)
  - Declaration of local variables (if any)
  - Value to be returned to the main program (if any)

DECLARING FUNCTIONS

- **Function return values**
  - The return( ) statement exits the function, and returns a value back to the program where the function was called
  - You can use the return( ) statement anywhere in function, but the bottom of the function is preferred

- **The type of the return value depends on the application**
  - Typical mathematical functions return float values
  - Typical I/O functions return input data or error/success flag
  - Some functions perform calculations but return no value (we use the special data type “void” in this case)

DECLARING FUNCTIONS

- **Consider the following programming task:**
  - Prompt the user to input a value between 1..9
  - Read the integer value from the user
  - Loop until a valid input value is entered
    - Prompt the user to input a value between 1..9
    - Read the integer value from the user
  - Return the number to main program

- **We can declare a C++ function to package this code together so it can be reused in different programs**

DECLARING FUNCTIONS

```
// Function declaration example
int ReadNum()
{
   int Number = 0;
   while ((Number < 1) || (Number > 9))
   {
      cout << ”Enter a number between 1..9: ";
      cin >> Number;
   }
   return( Number );
}
```

Name of the function we are declaring

DECLARING FUNCTIONS

```
// Function declaration example
int  ReadNum()
{
   int Number = 0;
   while ((Number < 1) || (Number > 9))
   {
      cout << ”Enter a number between 1..9: ";
      cin >> Number;
   }
   return( Number );
}
```

Operations to be performed by the function

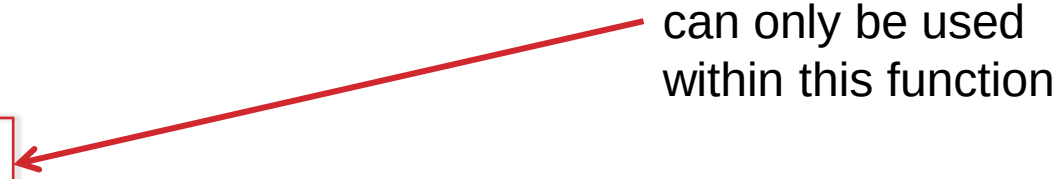
DECLARING FUNCTIONS

```
// Function declaration example
int ReadNum()
{
   int Number = 0;
   while ((Number < 1) || (Number > 9))
   {
      cout << ”Enter a number between 1..9: ";
      cin >> Number;
   }
   return( Number );
}
```

Data type of the function return value

DECLARING FUNCTIONS

```
// Function declaration example
int  ReadNum()
{
   int Number = 0;
   while ((Number < 1) || (Number > 9))
   {
       cout << ”Enter a number between 1..9: ";
       cin >> Number;
   }
   return( Number );
}
```



DECLARING FUNCTIONS

```
// Function declaration example
int  ReadNum()
{
   int Number = 0;
   cout << ”Enter a number between 1..9: ";
   cin >> Number;
   while ((Number < 1) || (Number > 9))
   {
       cout << ”Invalid.  Please enter a number between 1..9: ";
       cin >> Number;
   }
   return Number;
}
```

Value returned by the function to the main program

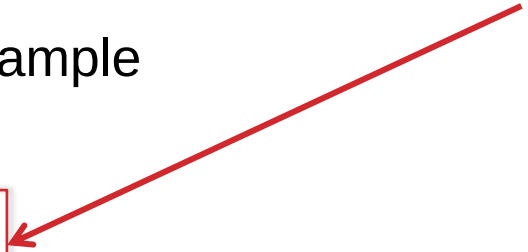
CALLING FUNCTIONS

- **Now that we have declared the function ReadNum how can we use this function in the main program?**
- **We need to specify the following in a function call**
  - The name of the function we wish to execute
  - Variables passed in as parameters (if any)
  - Operations to be performed with return value (if any)

- **The main program will jump to function code, execute it, and return with the value that was calculated**
- **Return values can be used to assign variables or they can be output (we should not ignore them)**

CALLING FUNCTIONS

```
int main()
{
…
   // Function usage example
   int Num = 0;
   Num = ReadNum();
   cout << "Your Lucky number is " << Num << endl;
…
}
```

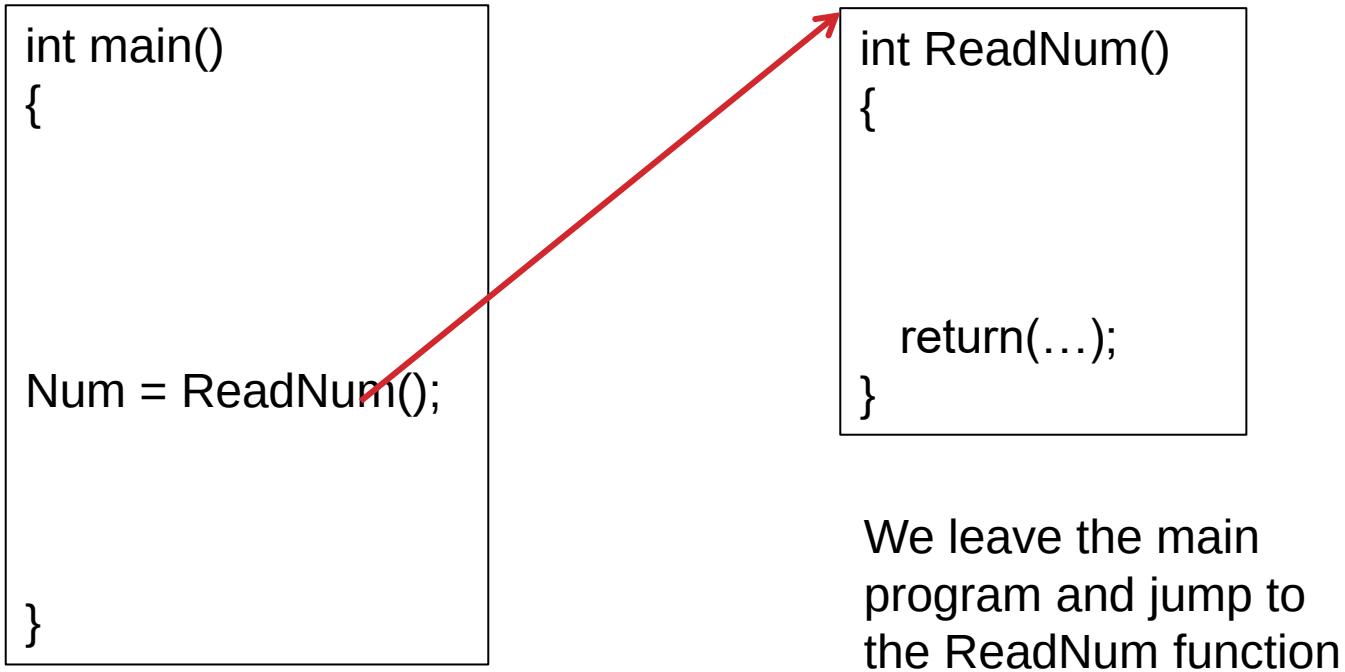


This function call will cause the main program to jump to the ReadNum function, execute that code and store the return value in variable Num

CALLING FUNCTIONS

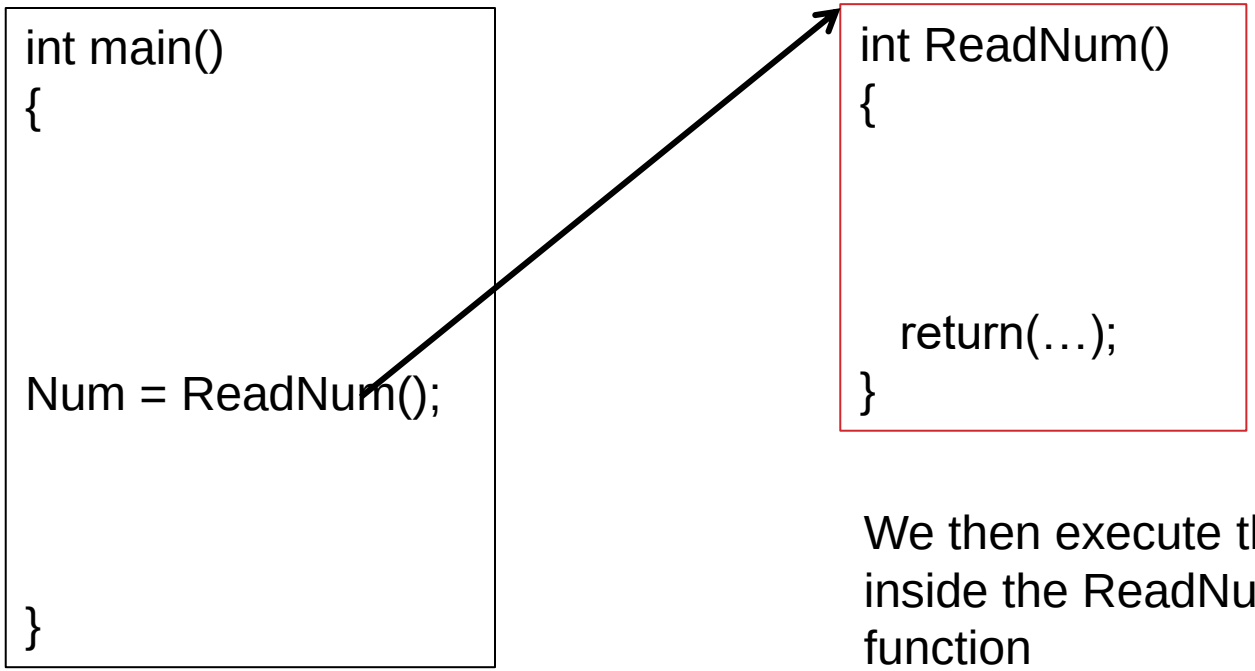
- **We can trace the executions functions in a program using the “black box” method**
  - Draw a box for the main program
  - Draw a second box for the function being called
  - Draw an arrow from the function call in the main program to the top of the function being called
  - Draw a second arrow from the bottom of the function back to the main program labeled with the return value
  - Using this black box diagram we can visualize the execution of the program by following the arrows

CALLING FUNCTIONS



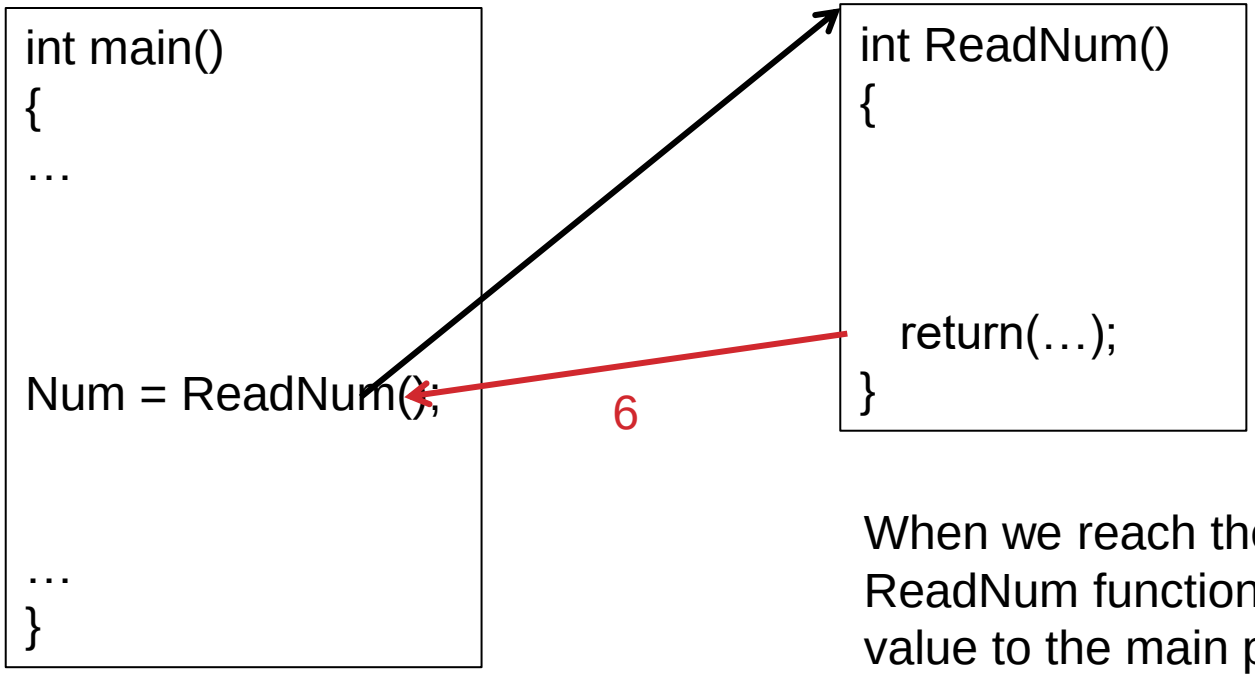
We leave the main program and jump to the ReadNum function

CALLING FUNCTIONS



We then execute the code inside the ReadNum function

CALLING FUNCTIONS



When we reach the bottom of the ReadNum function we return a value to the main program.
If the user enters the number 6 the return value would be 6

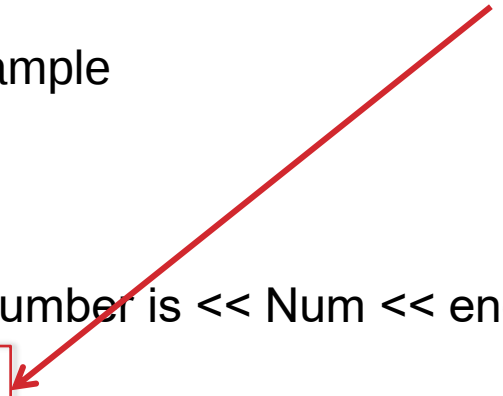
CALLING FUNCTIONS

```
int main()
{
…
   // Function usage example
   int Num = 0;
   Num = ReadNum();
   cout << "Your Lucky number is " << Num << endl;
…
}
```

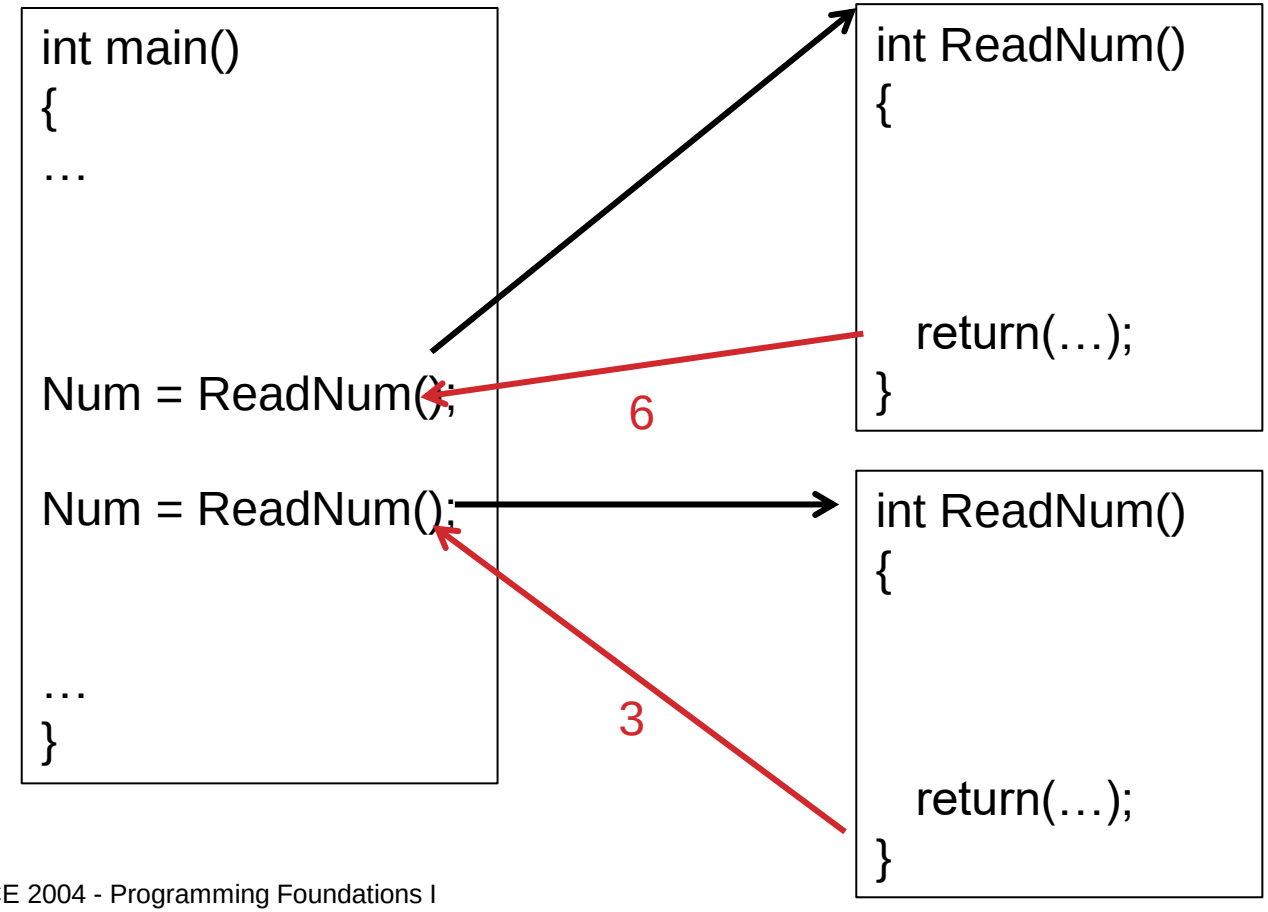
CALLING FUNCTIONS

```
int main()
{
…
   // Function usage example
   int Num = 0;
   Num = ReadNum();
   cout << “Your lucky number is << Num << endl;
   Num = ReadNum();
   cout << "Your second lucky number is " << Num << endl;
…
}
```

Each time we call ReadNum the code in the function will be executed and a new value will be returned



CALLING FUNCTIONS



FUNCTIONS WITHOUT RETURN VALUES

- **What happens if a function does not have a return value?**
  - In some cases, we just want a function to print out something like a help message or a command menu
  - In this case, the function does not need to return anything to the main program, so we can omit the return statement
  - When the code reaches the bottom of the function, it will automatically return to the location the function was called in the main program

FUNCTIONS WITHOUT RETURN VALUES

```
// Function declaration example
void PrintMenu()
{
   cout << “Welcome to ACME bank:\n”;
   cout << “Please enter one of the commands below:\n”;
   cout << “   d – Deposit money into your account\n”;
   cout << “   w – Withdraw money from your account\n”;
   cout << “   t – Transfer money between accounts\n”;
   cout << “   q – Quit this banking program\n”;
}
```

FUNCTIONS WITHOUT RETURN VALUES

```
int main()
{
  char Command;
   …
   // Function usage example
   PrintMenu();
   cin >> Command;
   cout << “Command:“ << Command << “:\n”;
   …
}
```

When we call PrintMenu, the program will jump to the function, print the messages, and then return to this location

LOCAL AND GLOBAL VARIABLES

- **"Local variables" are defined inside a function, and can only be accessed within the function**
  - Local variables to store temporary values
  - Initialize local variables each time the function is called

- **"Global variables" are defined at top of program before the function definitions, and can accessed anywhere in the program**
  - Global variables should only be used for constants like PI
  - Initialize global variables each time the program is run

LOCAL AND GLOBAL VARIABLES

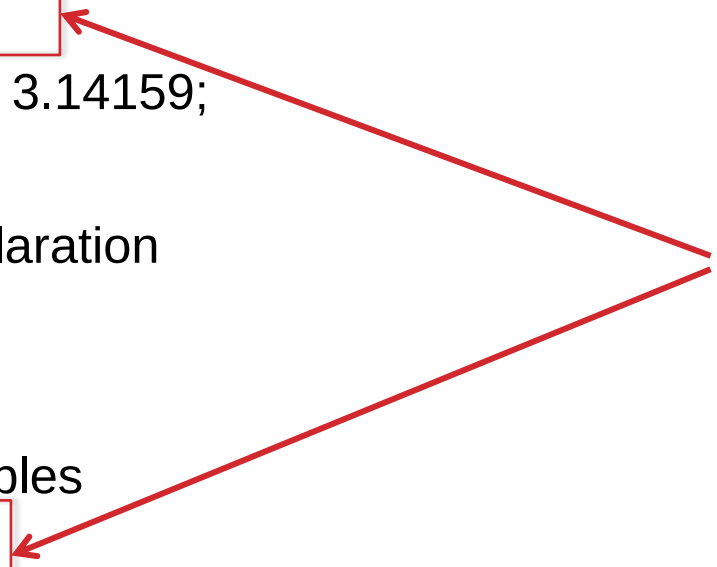
```
// Global variables
int Value = 42;
const float PI = 3.14159;

// Function declaration
float Silly()
{
  // Local variables
  float Value;
  int Data;
   …
}
```

These global variables can be used anywhere in the program

These local variables can only be used in the function Silly

LOCAL AND GLOBAL VARIABLES

```
// Global variables
int Value = 42;
const float PI = 3.14159;

// Function declaration
float Silly()
{
  // Local variables
  float Value;
  int Data;
   …
}
```



In this case, the local variable Value “hides” the global variable with the same name

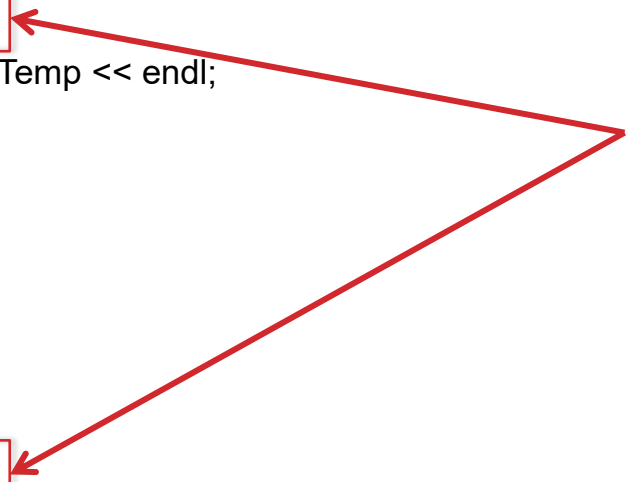
We can not access the global variable from within the Silly function

LOCAL AND GLOBAL VARIABLES

```
    …
   // Code inside Silly function
   cout << "Enter your age in months: ";
   cin >> Data;              // Changes local variable Data
   Value = Data * PI;        // Uses global constant PI
   return(Value);            // Returns local variable Value
}
```

LOCAL AND GLOBAL VARIABLES

```
// Function declaration
void FunctionOne ()
{
   // Local variables
   float Temp = 1.7;
   cout << “Temp: “ << Temp << endl;
}

// Function declaration
float FunctionTwo()
{
   // Local variables
   int Temp = 0;
   cout << “Temp: “ << Temp << endl;
    …
}
```



We are allowed to reuse the names of local variables in different functions

These variables are stored in different memory locations and do not effect each other in any way

SUMMARY

- In this section we have shown how a function can be declared in in C++
- We have also shown how a function can be called from within the main program
- Finally, we discussed the difference between local and global variables in a C++ program

FUNCTIONS

PART 2

FUNCTION PARAMETERS

VALUE PARAMETERS

- **A function may need some input to perform desired task**
  - We can pass data into function using parameters
  - We need to specify the data type and the names of parameters when defining a function
  - We need to provide the values of parameters when calling the function (hence the name value parameter)

- **Example**
  - To calculate the square of a number in a function, we need to pass in the number to square as a value parameter

VALUE PARAMETERS

```
// Function with value parameter
float Square( const float Number )
{
   float Result = Number * Number;
   return( Result );
}
```

Number is a value parameter with a float data type

We can use Number in our function to calculate desired return value

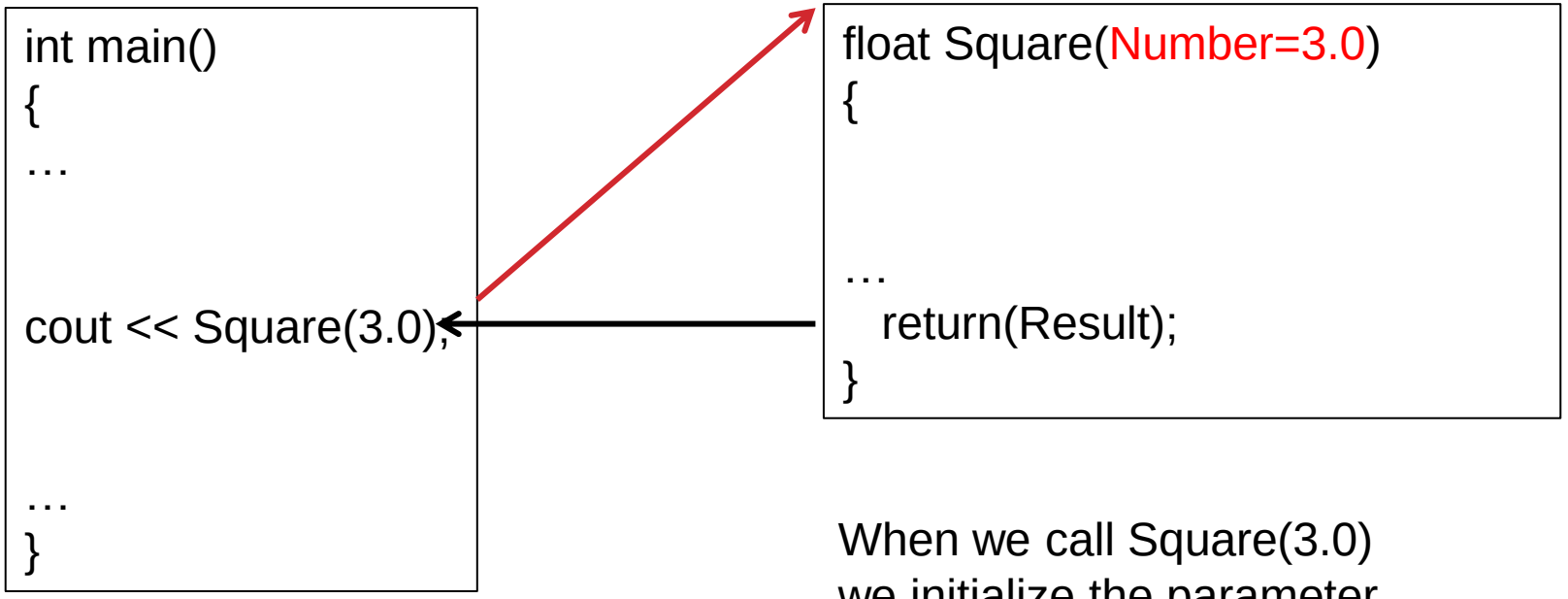
VALUE PARAMETERS

```
int main()
…
// Calling the Square function
cout << Square(3.0) << endl;
float Num = 5.0;
cout << Square(Num) << endl;
…
}
```

Call function with value 3.0

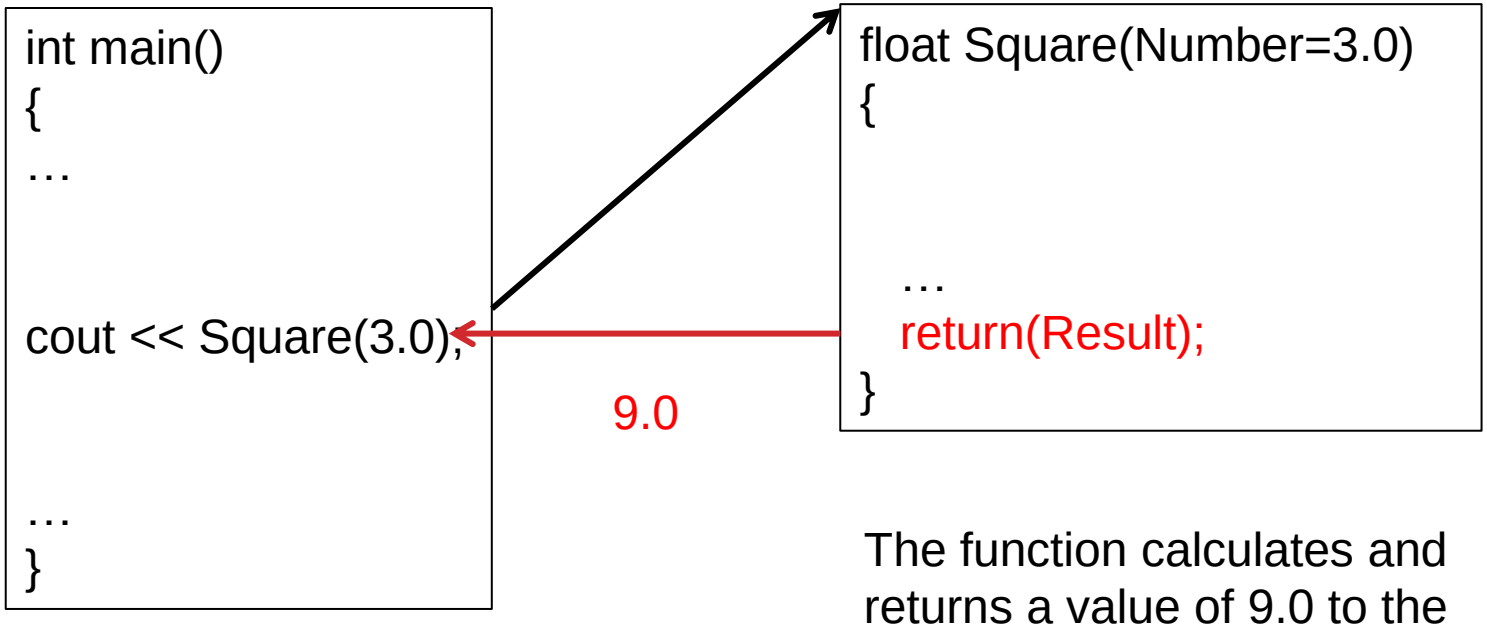
Call function with value 5.0

VALUE PARAMETERS



When we call Square(3.0)
we initialize the parameter
Number with a value of 3.0

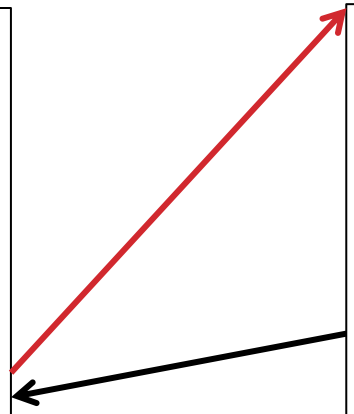
VALUE PARAMETERS



The function calculates and returns a value of 9.0 to the main program

VALUE PARAMETERS

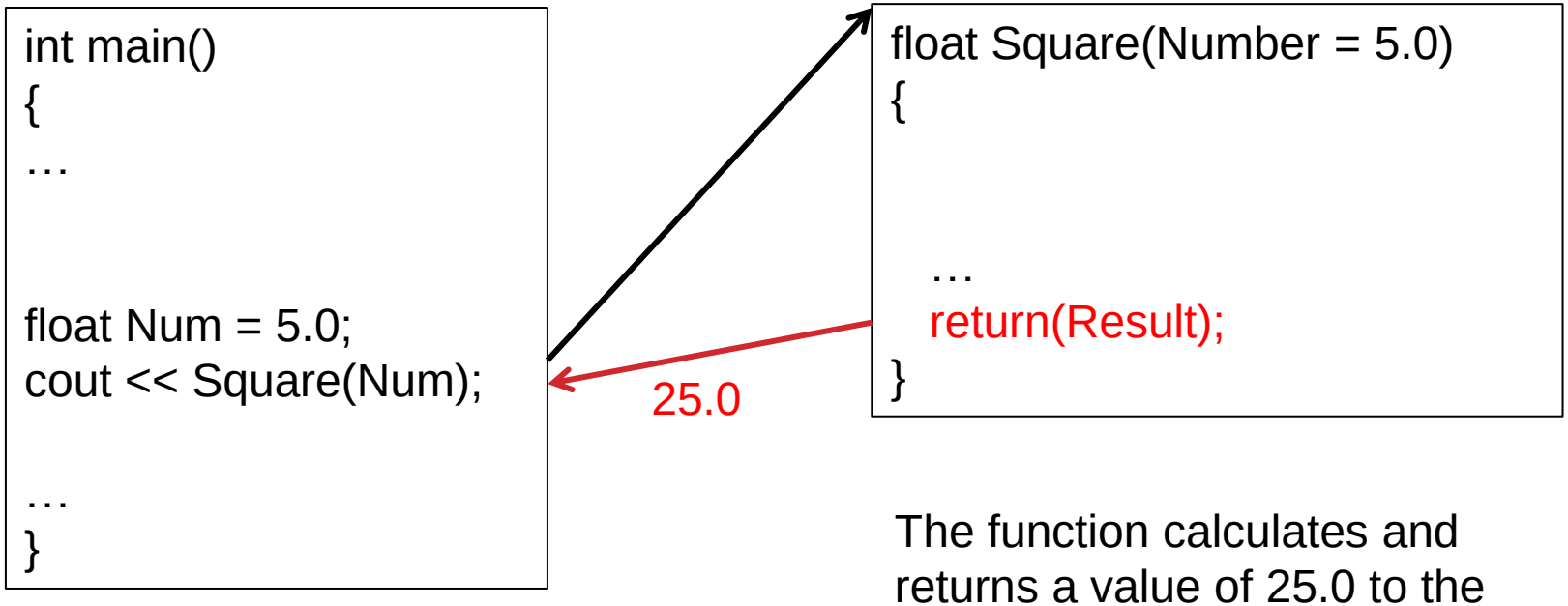
```
int main()
{
…


float Num = 5.0;
cout << Square(Num);

…
}
```



```
float Square(Number=5.0)
{


   …
   return(Result);
}
```

When we call Square(Num) we initialize the parameter Number with a value of 5.0

VALUE PARAMETERS



The function calculates and returns a value of 25.0 to the main program

VALUE PARAMETERS

- **Parameters versus globals:**
  - The recommended way to send information into functions is to use value parameters
  - Some people use global variables for this purpose, but this is an ugly and error prone approach, so don't be tempted to follow their example
- **Const versus no const**
  - We use the keyword "const" before the value parameter type so the variable can not be modified inside the function
  - Some people omit the "const", and modify the value parameter, but this may cause confusion in some cases, so this practice is discouraged

SUM DIGITS EXAMPLE

- **How can we calculate the sum of the digits in an integer between 0..99999?**
  - Find each of the 5 digits in the number
  - Add them up to get the sum of digits

- **How can we implement this using functions?**
  - We can implement a SumDigits function to extract the individual digits of the number and calculate their sum
  - We can write a main program that prompts the user for input, does error checking, and calls SumDigits to calculate the final answer

SUM DIGITS EXAMPLE

- **Which function should we implement first?**
  - There is no right way or wrong way to do this
- **If we implement and test the main program before SumDigits, this is called a top down approach**
  - In this case, we need to implement a “dummy” version of SumDigits to test the main program
- **If we implement and test SumDigits before the main program, this is called a bottom up approach**
  - In this case, we need a “dummy” version of the main program to test the SumDigits function

SUM DIGITS EXAMPLE

```
// Function to calculate sum of digits
int SumDigits(const int Number)
{
  // Return answer
  return (42);
}
```

With a top down approach we create a dummy version of this function that has the correct parameters but does not do any real work

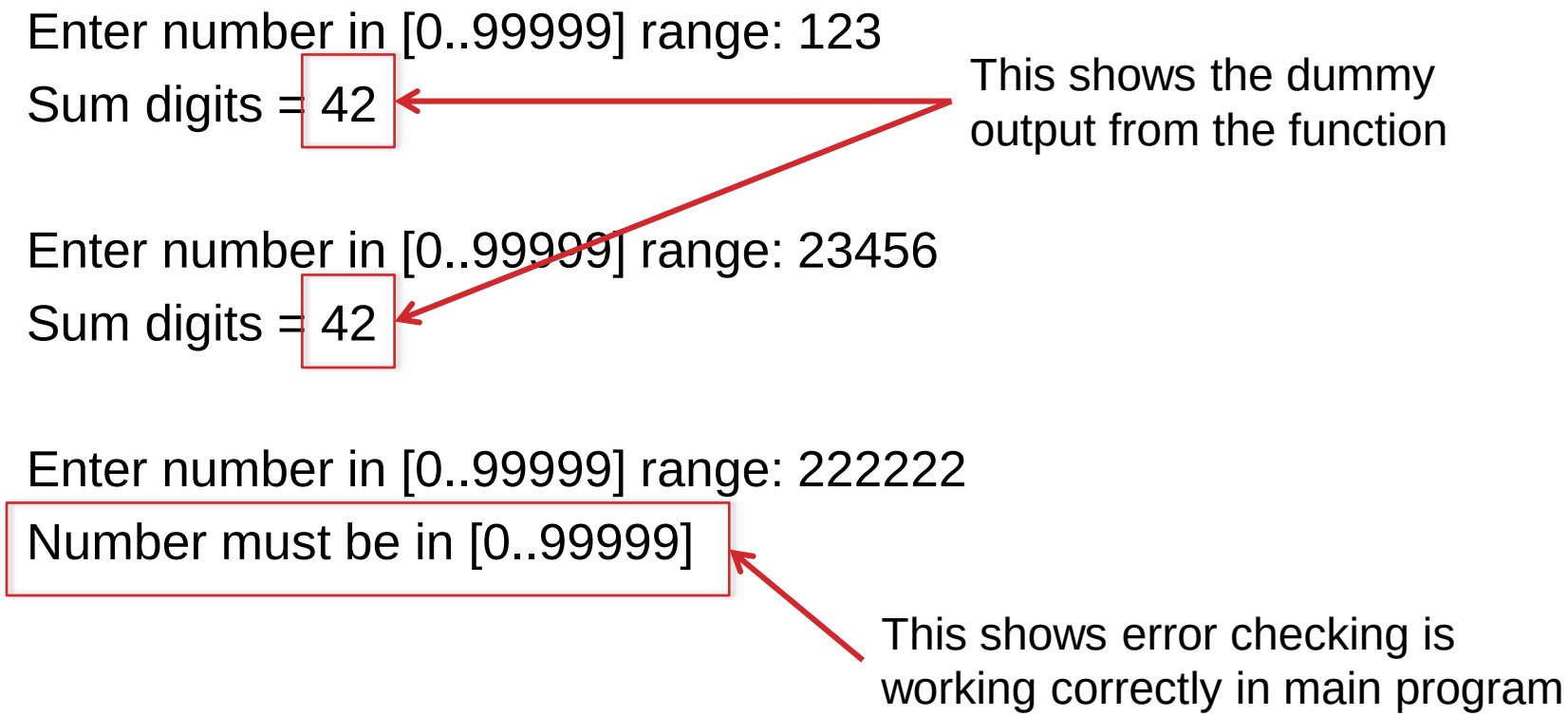
SUM DIGITS EXAMPLE

```
// Main body of program
int main()
{
   int Input;
   cout << "Enter number in [0..99999] range: ";
   cin >> Input;
   if ((Input < 0) || (Input > 99999))
      cout << "Number must be in [0..99999]" << endl;
   else
      cout << "Sum digits = " << SumDigits(Input)<< endl;
   return 0 ;
}
```

With a top down approach we write a main program that calls the dummy function

SUM DIGITS EXAMPLE

Initial program output:



SUM DIGITS EXAMPLE

- **How can we find the individual digits of the number so we can calculate their sum?**
  - The first digit is easy, just divide by 10000 to trim off the last four digits of the number

    ```
    int Digit1 = Number / 10000;
    ```

  - The last digit is also easy, just use the modulo 10 operation to trim off the first four digits of the number

    ```
    int Digit5 = Number % 10;
    ```

SUM DIGITS EXAMPLE

- The other digits require a careful combination of modulo and division operations to trim off digits from the front and the back of the number

```
int Digit2 = (Number % 10000) / 1000;
int Digit3 = (Number % 1000) / 100;
int Digit4 = (Number % 100) / 10;
```

- Notice there is a very nice symmetry to these calculations, which is normally a sign that you have the correct logic

SUM DIGITS EXAMPLE

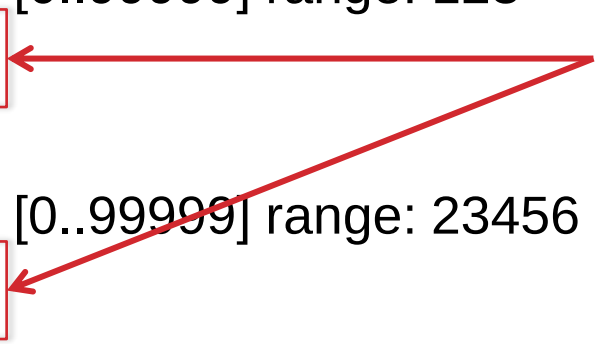
```
// Function to calculate sum of digits
int SumDigits(const int Number)
{
  // Get digits and calculate their sum
  int Digit1 = Number / 10000;
  int Digit2 = (Number % 10000) / 1000;
  int Digit3 = (Number % 1000) / 100;
  int Digit4 = (Number % 100) / 10;
  int Digit5 = Number % 10;
  return (Digit1+Digit2+Digit3+Digit4+Digit5);
}
```

With a top down approach we now fill in the correct SumDigits function and run it with our previous main program

SUM DIGITS EXAMPLE

Final program output:

```
Enter number in [0..99999] range: 123
Sum digits = 6

Enter number in [0..99999] range: 23456
Sum digits = 20

Enter number in [0..99999] range: 222222
Number must be in [0..99999]
```



SQUARE ROOT EXAMPLE

- **How can we calculate the square root R of a number N?**
  - Use the ancient Babylonian method
  - Make initial guess of square root value R = N / 2
  - Calculate N / R
  - The correct answer is somewhere between R and N / R
  - Make new guess R = (R + N / R) / 2
  - Keep iterating until the value of R converges
    (or we get tired of doing this process)

SQUARE ROOT EXAMPLE

- **Example: Calculating square root of 49**
  - Initial guess R = 49 / 2 = 24.5
  - Calculate N/R = 49 / 24.5 = 2
  - New guess R = (24.5 + 2) / 2 = 13.25
  - Calculate N/R = 49 / 13.25 = 3.69
  - New guess R = (13.25 + 3.69) / 2 = 8.47
  - Calculate N/R = 49 / 8.47 = 5.78
  - New guess R = (8.47 + 5.78) / 2 = 7.125
  - Calculate N/R = 49 / 7.125 = 6.877
  - New Guess R = (7.125 + 6.877) / 2 = 7.001

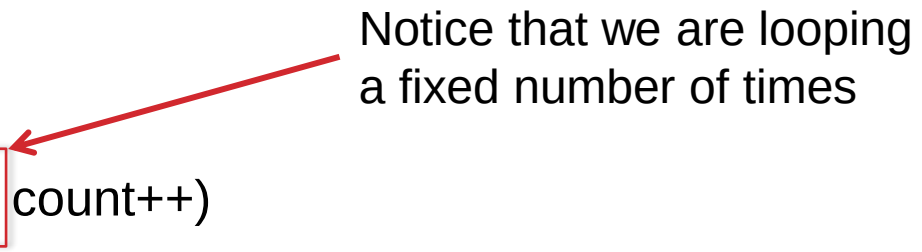
SQUARE ROOT EXAMPLE

```
// Function to calculate square root
double my_sqrt(const double number)
{
    double root = number / 2;
    for (int count = 0; count < 10; count++)
        root = (root + number / root) / 2;
    return (root);
}
```

With a bottom up approach we implement this function first before completing the main program

SQUARE ROOT EXAMPLE

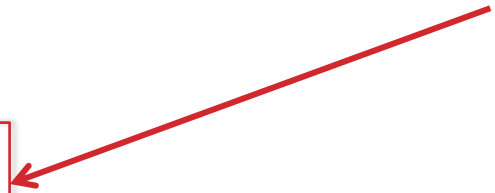
```
// Function to calculate square root
double my_sqrt(const double number)
{
   double root = number / 2;
   for (int count = 0; count < 10; count++)
   {
      cout << count << " " << root << endl;
      root = (root + number / root) / 2;
   }
   return (root);
}
```



SQUARE ROOT EXAMPLE

```
// Main body of program
int main()
{
   double Number = 49;
   double Root = my_sqrt(Number);
   cout << "Root = " << Root << endl;
   return Root ;
}
```

With a bottom up approach we use a simple main program to debug the my_sqrt function



SQUARE ROOT EXAMPLE

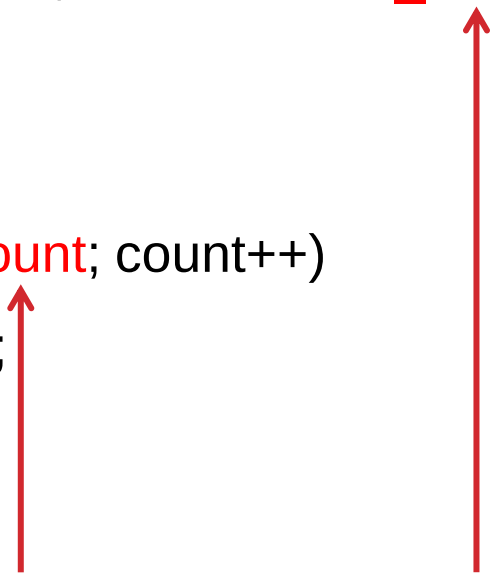
Sample Program Output:

```
0 24.5
1 13.25
2 8.47406
3 7.12821
4 7.00115
5 7
6 7
7 7
8 7
9 7
Root = 7
```

Notice that we have converged to the correct answer after 5 iterations

SQUARE ROOT EXAMPLE

```
// Function to calculate square root
double my_sqrt(const double number, const int max_count)
{
   double root = number / 2;
   for (int count = 0; count < max_count; count++)
      root = (root + number / root) / 2;
   return (root);
}
```

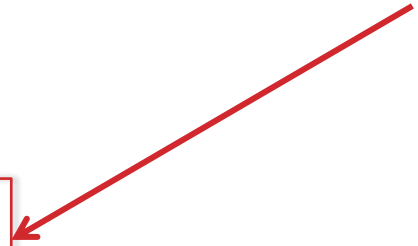


We can add a second parameter to control the number of loops

SQUARE ROOT EXAMPLE

```
// Main body of program
int main()
{
   double Number = 49;
   double Root = my_sqrt(Number, 42);
   cout << "Root = " << Root << endl;
   return Root ;
}
```

Now we call the my_sqrt function with two parameters



REFERENCE PARAMETERS

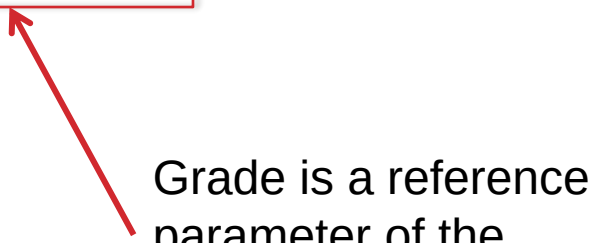
- **In some applications, a function may need to change the value of one or more variables in the main program**
  - We can use reference parameters to accomplish this
- **How do we define reference parameters?**
  - We need to specify the data type and the name of reference parameters when defining the function
  - To specify that this is a reference parameter, we add "&" between the data type and parameter name

REFERENCE PARAMETERS

- **How do we call functions with reference parameters?**
  - When calling the function, we must provide the names of the variables we want to pass in as reference parameters
  - This makes the reference parameter an alias for the variable we passed in
  - We now have two names for the same piece of memory in the program
  - Changes to the reference parameter inside the function will really change the variable you passed into the function
  - We can not use a literal constant or an expression as the parameter to the function, we can only use variable names

REFERENCE PARAMETERS

```
// Function with reference parameter
void GetGrade(const float Average, char & Grade)
{
   if (Average >= 90) Grade = 'A';
   else if (Average >= 80) Grade = 'B';
   else if (Average >= 70) Grade = 'C';
   else if (Average >= 60) Grade = 'D';
   else Grade = 'F';
}
```



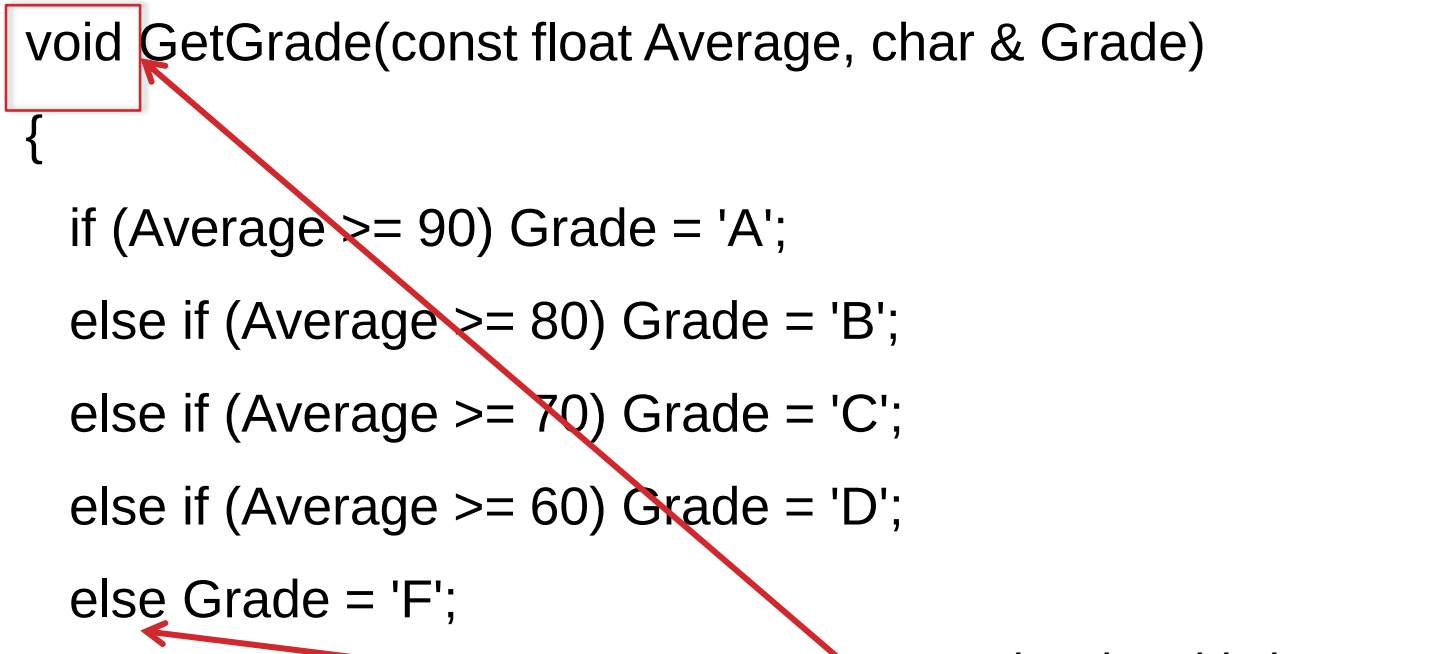
REFERENCE PARAMETERS

```
// Function with reference parameter
void GetGrade(const float Average, char & Grade)
{
  if (Average >= 90) Grade = 'A';
  else if (Average >= 80) Grade = 'B';
  else if (Average >= 70) Grade = 'C';
  else if (Average >= 60) Grade = 'D';
  else Grade = 'F';
}
```

Changes to Grade in this function will modify some variable in the main program

REFERENCE PARAMETERS

```
// Function with reference parameter
void GetGrade(const float Average, char & Grade)
{
   if (Average >= 90) Grade = 'A';
   else if (Average >= 80) Grade = 'B';
   else if (Average >= 70) Grade = 'C';
   else if (Average >= 60) Grade = 'D';
   else Grade = 'F';
}
```



Notice that this is a void function and there is no return statement

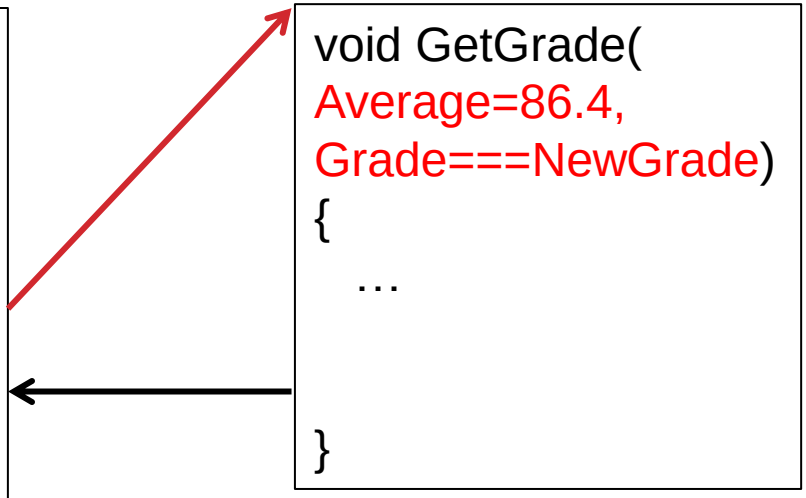
REFERENCE PARAMETERS

```
int main()
{
…
// Calling function with reference parameter
float NewScore = 86.4;
char NewGrade = '?';
GetGrade(NewScore, NewGrade);
cout << "Your final grade is " << NewGrade << endl;
…
}
```

Here we call the GetGrade function with NewGrade as a reference parameter

REFERENCE PARAMETERS

```
int main()
{
…

float NewScore = 86.4;
char NewGrade = '?';
GetGrade(NewScore, NewGrade);

…
}
```



```
void GetGrade(
Average=86.4,
Grade===NewGrade)
{
   …


}
```

When we call GetGrade the parameter Average is 86.4 and parameter Grade is an alias for NewGrade

REFERENCE PARAMETERS

```
int main()
{
…

float NewScore = 86.4;
char NewGrade = '?';
GetGrade(NewScore, NewGrade);
                          = 'B'

…
}
```

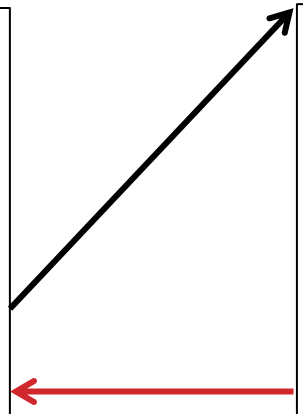
```
void GetGrade(
Average=86.4,
Grade===NewGrade)
{
…
Grade = 'B';

}
```

When we change Grade to be equal to 'B' we are really changing NewGrade to 'B'

REFERENCE PARAMETERS

```
int main()
{
…

float NewScore = 86.4;
char NewGrade = '?';
GetGrade(NewScore, NewGrade);
                                = 'B'

…
}
```

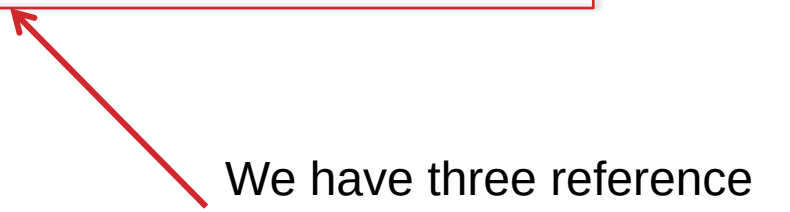
```
void GetGrade(
Average=86.4,
Grade===NewGrade)
{
…
Grade = 'B';

}
```



When we return to the main program the value of NewGrade has been modified and can be used in the rest of the program

REFERENCE PARAMETERS

```
// Function to read student information from user
void GetInfo(string & FirstName, string & LastName, int & Age)
{
   cout << "Enter first name: ";
   cin >> FirstName;
   cout << "Enter last name: ";
   cin >> LastName;
   cout << "Enter age: ";
   cin >> Age;
}
```



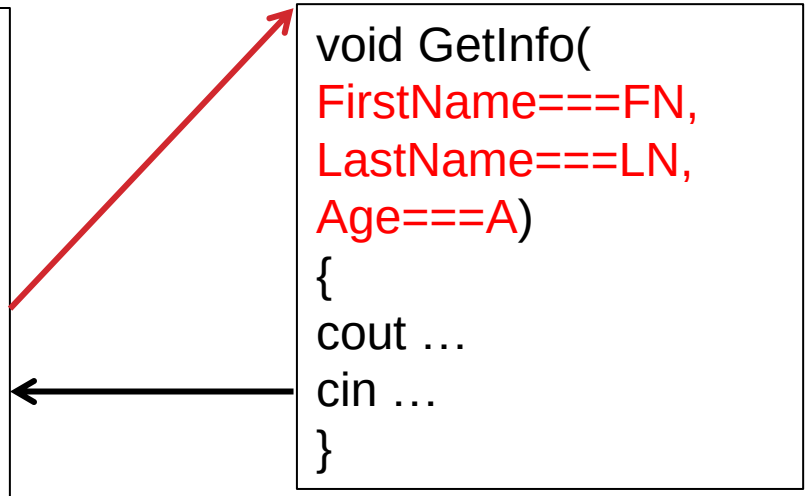
REFERENCE PARAMETERS

```
int main()
{
…
// Calling function with reference parameters
string FN, LN;
int A = 0;
GetInfo(FN, LN, A);
cout << FN << " age is " << A<< endl;
…
}
```

Here we call the GetInfo function with three reference parameters

REFERENCE PARAMETERS

```
int main()
{
…

string FN, LN;
int A = 0;
GetInfo(FN, LN, A);

…
}
```



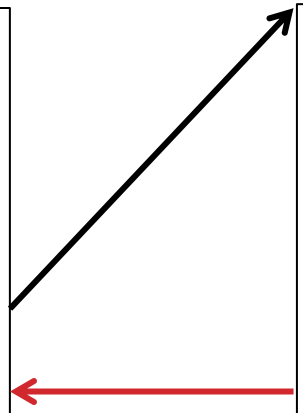
```
void GetInfo(
FirstName===FN,
LastName===LN,
Age===A)
{
cout …
cin …
}
```

When we call GetInfo the three parameters are aliases for three variables in the main program

REFERENCE PARAMETERS

```
int main()
{
…

string FN, LN;
int A = 0;
GetInfo(FN, LN, A);

…
}
```

```
void GetInfo(
FirstName===FN,
LastName===LN,
Age===A)
{
cout …
cin …
}
```



When we return to the main program the variables FN, LN and A have been modified

REFERENCE PARAMETERS

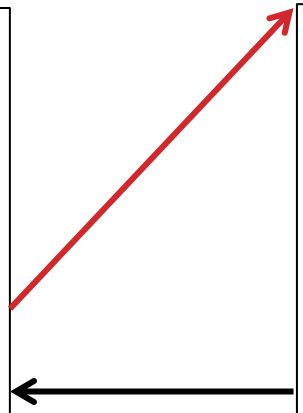
```
int main()
{
…
// Calling function with reference parameters
string FN, LN;
int A = 0;
GetInfo(LN, FN, A);
cout << FN << " age is " << A<< endl;
…
}
```

Calling function with the parameters in wrong order is a common problem

REFERENCE PARAMETERS

```
int main()
{
…

string FN, LN;
int A = 0;
GetInfo(LN, FN, A);

…
}
```

```
void GetInfo(
FirstName===LN,
LastName===FN,
Age===A)
{
cout …
cin …
}
```



Now the first name and last name will be stored in the wrong variables in the main program

REFERENCE PARAMETERS

- **Advantages:**
  - We can use reference parameters to modify multiple variables in the main program
  - No data is copied from the main program to the function when we make the function call (unlike value parameters)

- **Disadvantages**
  - We must provide a variable name when calling functions with reference parameters
  - Keeping track of the “aliases” created by reference parameters is more difficult than tracing value parameters

PRIME FACTORS EXAMPLE

- **How can we calculate and print the prime factors of an integer that is between 1..100?**
  - We need to print a message for each time one of the prime numbers 2,3,5,7 divides evenly into the input value
  - We also need to modify the input value by dividing it by this prime factor until it is no longer a factor

input = 50

2 is a factor, input = 50/2 = 25

5 is a factor, input = 25/5 = 5

5 is a factor, input = 5/5 = 1

PRIME FACTORS EXAMPLE

- **How can we implement this using functions?**
  - There is no wrong answer as long as we are breaking the larger problem down into smaller pieces that are easy to implement and debug
- **Option 1:**
  - Have the main program handle user input and have the function do all the work and print out the prime factors
- **Option 2:**
  - Have the main program handle user input and then call the function multiple times to check each of the prime factors one at a time (we illustrate this approach below)

PRIME FACTORS EXAMPLE

```
// Main body of program
int main()
{
   // Get user input
   int Num = 0;
   cout << "Enter number in [1..100] range: ";
   cin >> Num;

   // Check input value
   if ((Num < 1) || (Num > 100))
      cout << "Number must be in [1..100]" << endl;
```

PRIME FACTORS EXAMPLE

```
...
  // Calculate prime factors
  else
  {
    CheckFactors(Num, 2);
    CheckFactors(Num, 3);
    CheckFactors(Num, 5);
    CheckFactors(Num, 7);
    if (Num > 1)
       cout << Num << " is a factor\n";
  }
```

We call the CheckFactors function four times to check if Num is divisible by 2,3,5,7 and print a message if it is a factor

PRIME FACTORS EXAMPLE

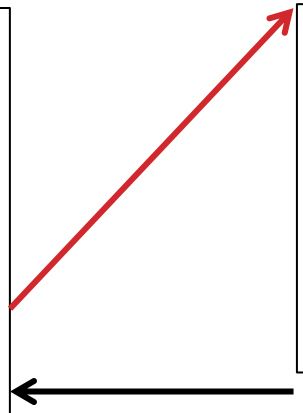
```
// Function to check one prime factor
void CheckFactors( int & Number, const int Factor )
{
   while (Number % Factor == 0)
   {
      cout << Factor << " is a factor\n";
      Number = Number / Factor;
   }
}
```

Number is a reference parameter and Factor is a value parameter

Here is where we modify the reference parameter which is an alias for a variable in main program

REFERENCE PARAMETERS

```
int main()
{
…

int Num = 0;
cin >> Num;
CheckFactors(Num, 2);

…
}
```

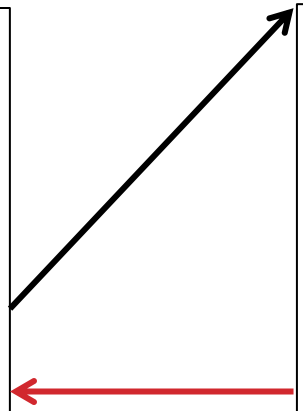


```
void CheckFactors(
Number===Num,
Factor=2)
{
…
}
```

Now Number will be an alias for Num in the main program

REFERENCE PARAMETERS

```
int main()
{
…

int Num = 0;
cin >> Num;
CheckFactors(Num, 2);

…
}
```

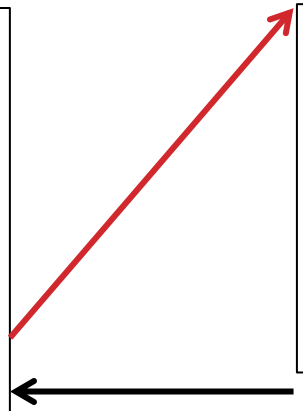


```
void CheckFactors(
Number===Num,
Factor=2)
{
…
Number = Number/Factor
…
}
```

All changes to Number will change the Num variable

REFERENCE PARAMETERS

```
int main()
{
…

int Num = 0;
cin >> Num;
CheckFactors(Num, 2);
CheckFactors(Num, 3);

…
}
```

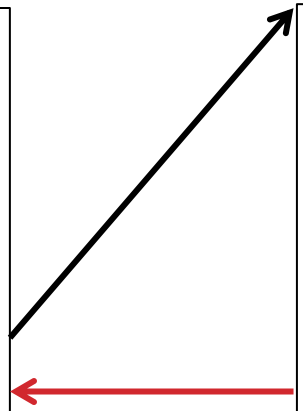


```
void CheckFactors(
Number===Num,
Factor=3)
{
…
}
```

Now Number will be an alias for Num in the main program

REFERENCE PARAMETERS

```
int main()
{
…

int Num = 0;
cin >> Num;
CheckFactors(Num, 2);
CheckFactors(Num, 3);

…
}
```



```
void CheckFactors(
Number===Num,
Factor=3)
{
…
Number = Number/Factor
…
}
```

All changes to Number will change the Num variable

PRIME FACTORS EXAMPLE

Sample program output:

```
Enter number in [1..100] range: 42
2 is a factor
3 is a factor
7 is a factor

Enter number in [1..100] range: 65
5 is a factor
13 is a factor
```

PRIME FACTORS EXAMPLE

- **How could we extend this program to handle values greater than 100?**
  - We could add a loop in the main program that calls the CheckFactors function with every value between 2 and the square root of Num to see if they are factors

```
for (int Factor=2; Factor<sqrt(Num); Factor++)
    CheckFactors(Num, Factor);
```

  - This would remove the need for input error checking and would also make the main program much shorter

SUMMARY

- **In this section we have described how value parameters and reference parameters can be used to get data in and out of functions**
  - Value parameters: “const int Number”
  - Reference parameters: “float & Value”

- **We have also shown two example programs that use functions with parameters to solve problems**
  - Sum of digits
  - Prime factors

FUNCTIONS

PART 3

RECURSION

RECURSION

- **Recursion is a very powerful problem solving method**

- **Recursive functions call themselves to solve problems**
  - We reduce "size" of problem each recursive function call
  - Need to have a terminating condition to stop recursion
  - Visualize using black box method with inputs and returns

- **Recursive solutions can also be implemented via iteration**
  - Sometimes the recursive function is smaller
  - Sometimes the iterative function is faster

RECURSION

- **Example: calculating N factorial**

- **Recursive solution solves problem in terms of itself**
  - Factorial(N) = N * Factorial(N-1)
  - This is the recurrence relationship

- **We must stop at some point**
  - Factorial(1) = 1
  - This is the terminating condition

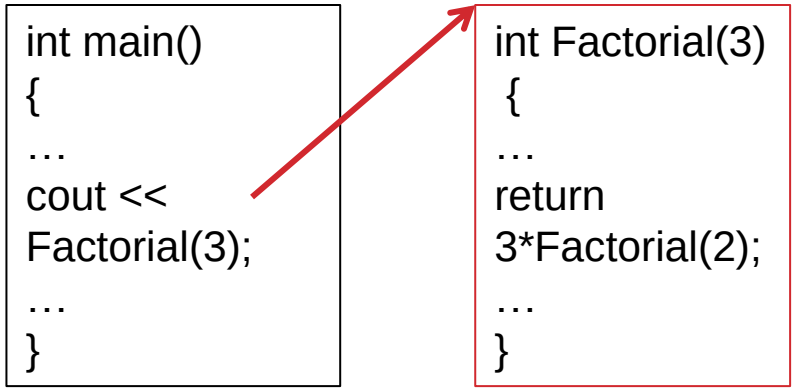
RECURSION

```
// Recursive Factorial function
int Factorial(const int Num)
{
   if (Num <= 1)
      return(1);
   else
      return( Num * Factorial(Num-1) );
}
```

This is the terminating condition of for the recursive function

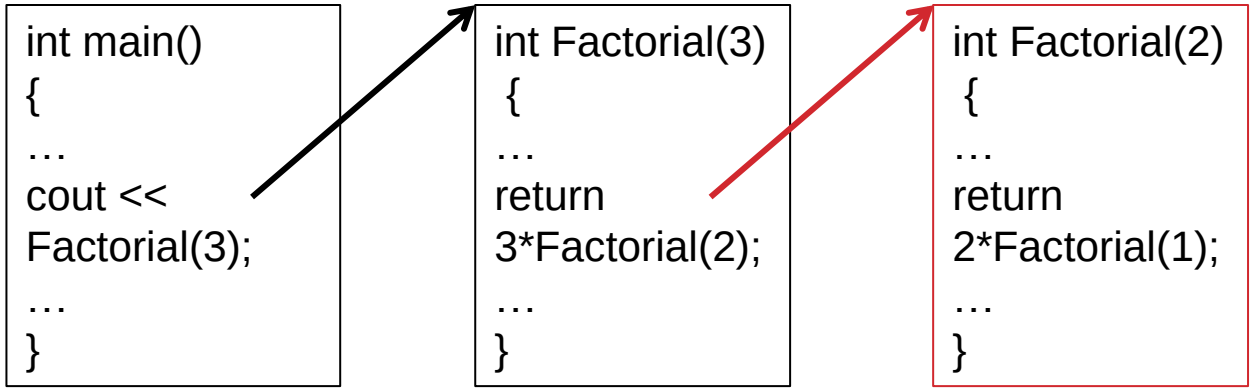
Notice that we are recursively calling the Factorial function with a smaller parameter value

RECURSION



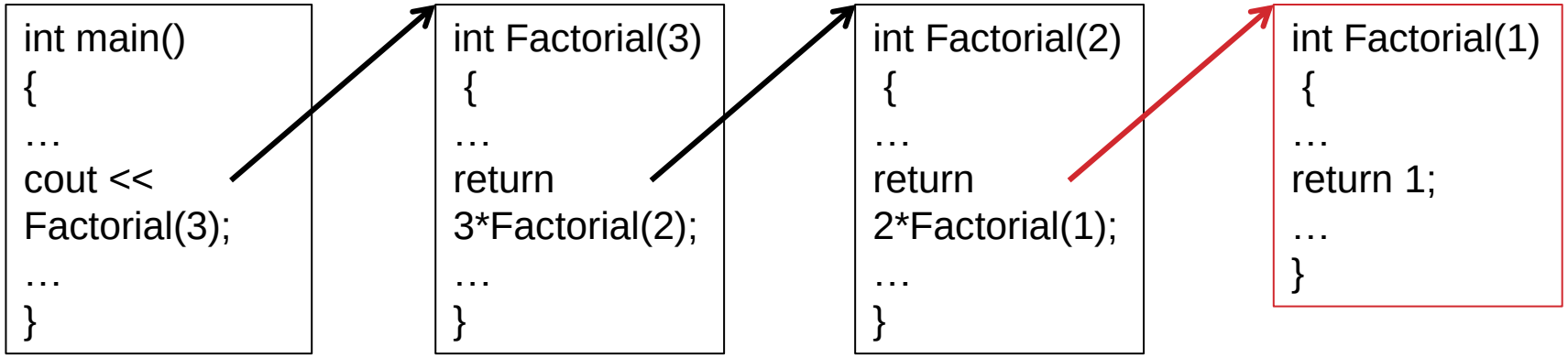
- First, the main program calls Factorial(3)

RECURSION



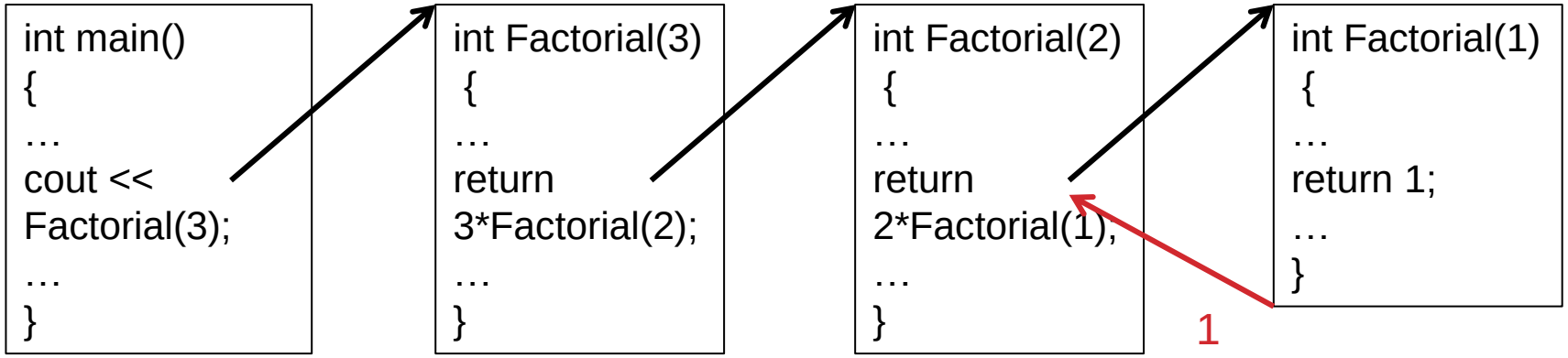
- First, the main program calls Factorial(3)
  - Then, Factorial(3) calls Factorial(2)

RECURSION



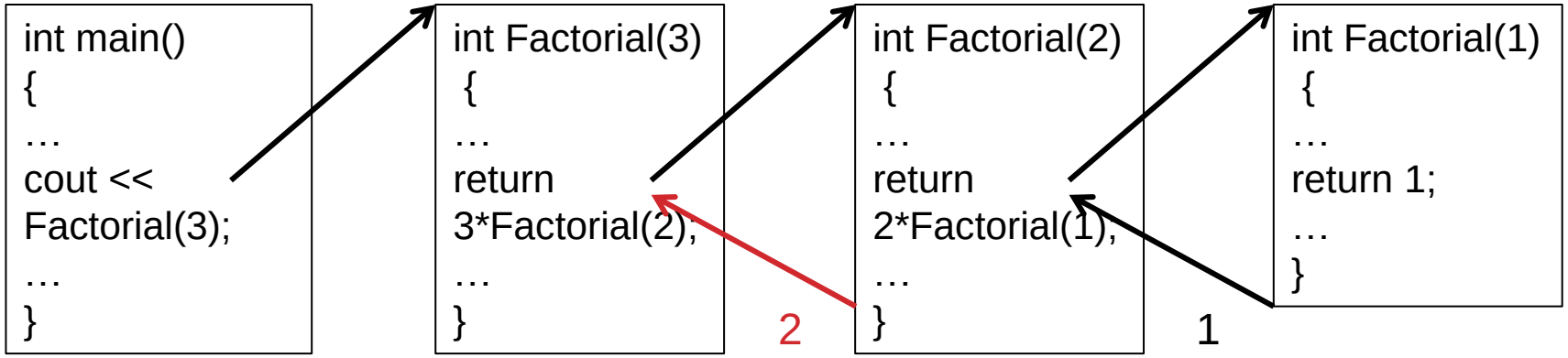
- First, the main program calls Factorial(3)
  - Then, Factorial(3) calls Factorial(2)
    - Then, Factorial(2) calls Factorial(1)

RECURSION



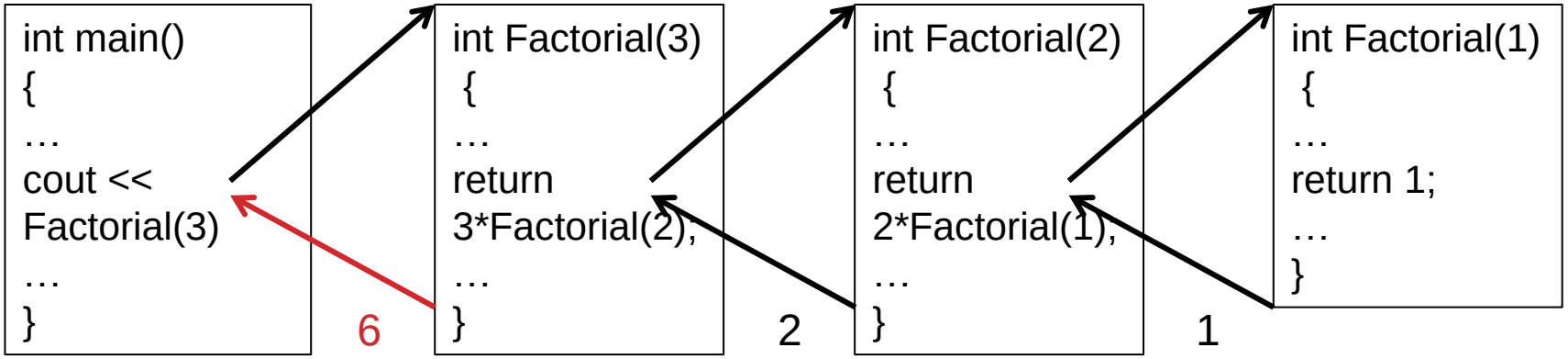
- First, the main program calls Factorial(3)
  - Then, Factorial(3) calls Factorial(2)
    - Then, Factorial(2) calls Factorial(1)
      - Then, Factorial(1) returns 1

RECURSION



- First, the main program calls Factorial(3)
  - Then, Factorial(3) calls Factorial(2)
    - Then, Factorial(2) calls Factorial(1)
      - Then, Factorial(1) returns 1
        - Then, Factorial(2) returns 2*1=2

RECURSION



- First, the main program calls Factorial(3)
  - Then, Factorial(3) calls Factorial(2)
    - Then, Factorial(2) calls Factorial(1)
      - Then, Factorial(1) returns 1
        - Then, Factorial(2) returns 2*1=2
          - Then, Factorial(3) returns 3*2=6
            - Finally, the main program prints 6

RECURSION

- **In this case the iterative solution is the same size and will run slightly faster than the recursive solution**

```
// Iterative Factorial function
int Factorial(const int Num)
{
   int Product = 1;
   for (int Count = 1; Count <= Num; Count++)
      Product = Product * Count;
   return( Product );
}
```

RECURSION

- **Example: calculating sum of all numbers from 1..N**

- **Recursive solution solves problem in terms of itself**
  - Sum(N) = N + Sum(N-1)
  - This is called the recurrence relationship

- **We must stop at some point**
  - Sum(1) = 1
  - This is called the terminating condition

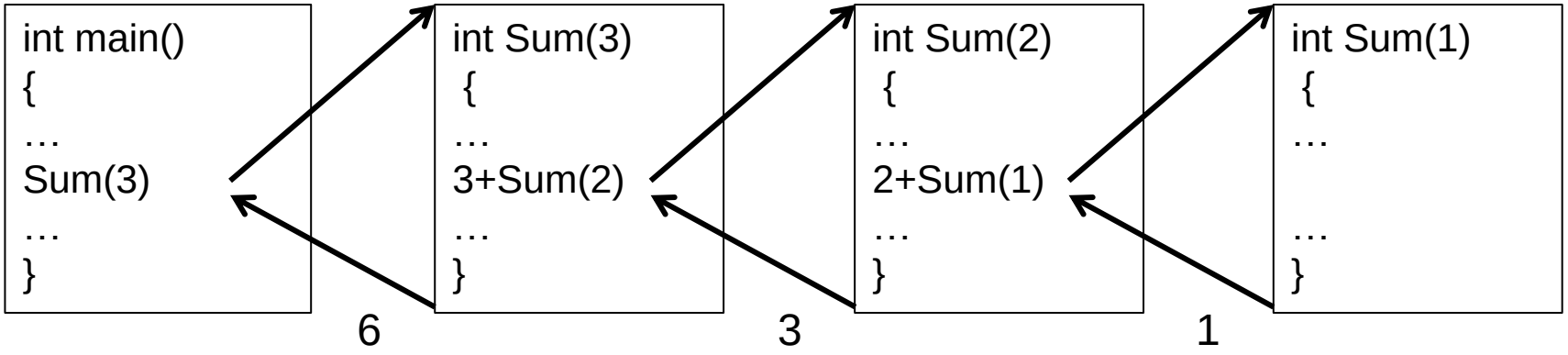
RECURSION

```
// Recursive summation function
int Sum(const int Num)
{
   if (Num <= 1)
      return(1);
   else
      return( Num + Sum(Num-1) );
}
```

This is the terminating condition of for the recursive function

Notice that we are recursively calling the Sum function with a smaller parameter value

RECURSION



- First, the main program calls Sum(3)
  - Then, Sum(3) calls Sum (2)
    - Then, Sum(2) calls Sum (1)
      - Then, Sum(1) returns 1
        - Then, Sum(2) returns 2+1=3
          - Finally, Sum(3) returns 3+3=6

RECURSION

- **In this case the iterative solution is the same size and will run slightly faster than the recursive solution**

```
// Iterative summation function
int Sum(const int Num)
{
   int Total = 0;
   for (int Count = 0; Count <= Num; Count++)
      Total = Total + Count;
   return(Total);
}
```

RECURSION

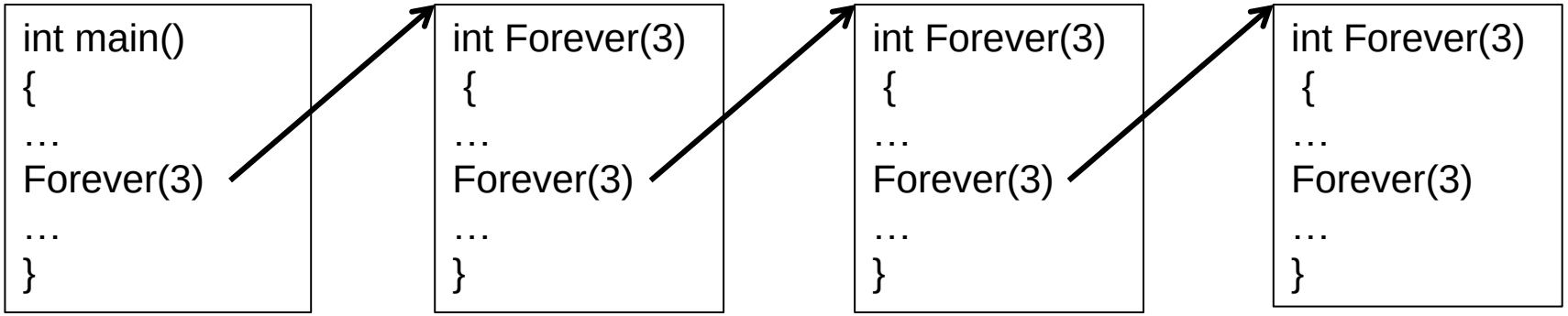
- **What happens if we make a mistake implementing the recurrence relationship?**
  - If the recursive function call has the same parameter value the function will call itself forever
  - This programming bug is called infinite recursion
  - Your program will crash with a message like “stack overflow” because it ran out of memory

RECURSION

```
// Infinite recursion function
int Forever(int Num)
{
   if (Num < 0)
      return(0);
   else
      return( Forever(Num) + 1 );
}
```

We are recursively calling the Forever function with the same parameter value

RECURSION



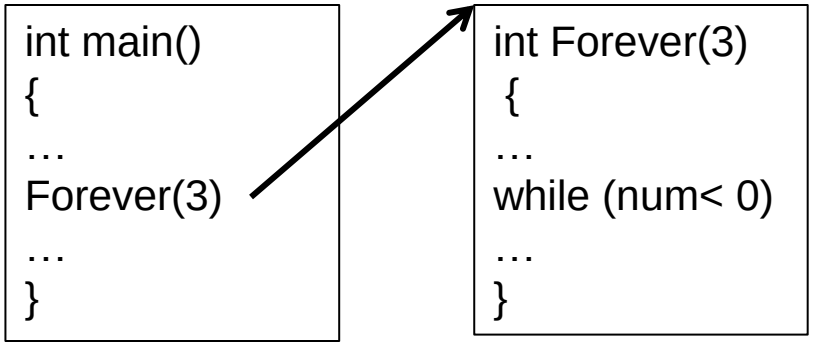
- First, the main program calls Forever(3)
  - Then, Forever(3) calls Forever(3)
    - Then, Forever(3) calls Forever(3)
      - This continues until stack overflow

RECURSION

- **The corresponding iterative program will not crash, but it will run forever adding one to the Total variable**

```
// Infinite iteration function
int Forever(int Num)
{
   int Total = 0;
   while (Num < 0)
      Total = Total + 1;
   return( Total );
}
```

RECURSION



- First, the main program calls Forever(3)
  - Then, Forever(3) loops forever
    - You must kill program yourself

RECURSION

- **When two functions call each other recursively it is called mutual recursion**
  - Function A calls function B, and function B calls function A

- **Normally functions are defined before they are used. How can we do this for mutual recursion?**
  - The C++ solution is to provide function headers
  - These give the “signature” of the function (return type, function name, and function parameters)
  - Function headers are placed at the top of the program

RECURSION

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

// Function headers
int Silly(int Num);
int Magic(int Val);
```

These function headers tell the compiler what these functions will look like, so it can properly handle calls to these functions in the code

RECURSION

```
// Function declaration
int Silly(int Num)
{
   cout << "calling silly " << Num << endl;
   if (Num < 42)
      return( Num + 13 );
   else
      return( Magic(Num*2) );
}
```

This debugging line will let us see what recursive calls occur when the program runs

Here is the recursion call

RECURSION

```
// Function declaration
int Magic(int Val)
{
   cout << "calling magic " << Val << endl;
   if (Val > 0)
      return( Val % 17 );
   else
      return( Silly(Val-8) );
}
```

This debugging line will let us see what recursive calls occur when the program runs

Here is the recursion call

RECURSION

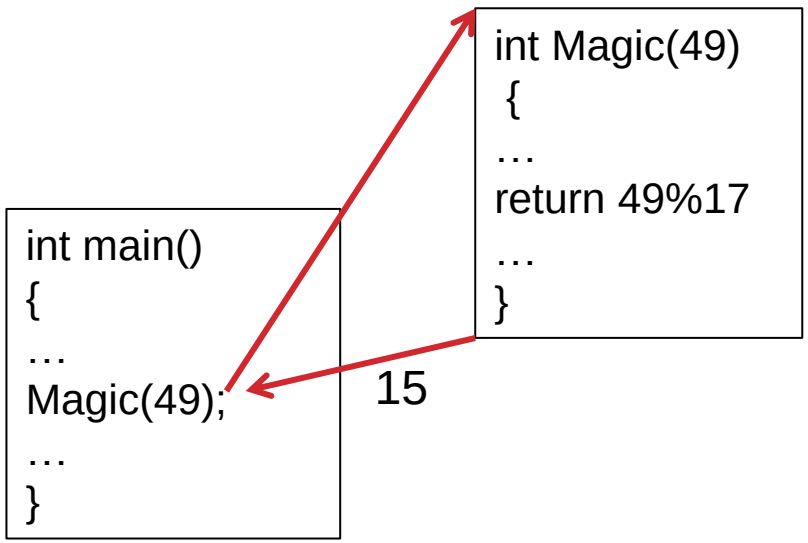
```
// Main program
int main()
{
   int Number;
   cout << "Enter number: ";
   cin >> Number;
   cout << "Magic(" << Number << ") = " << Magic(Number) << endl;
   cout << "Silly(" << Number << ") = " << Silly(Number) << endl;
   return(0);
}
```

RECURSION

Sample program output:

```
Enter number: 49

calling magic 49
Magic(49) = 15

calling silly 49
calling magic 98
Silly(49) = 13
```

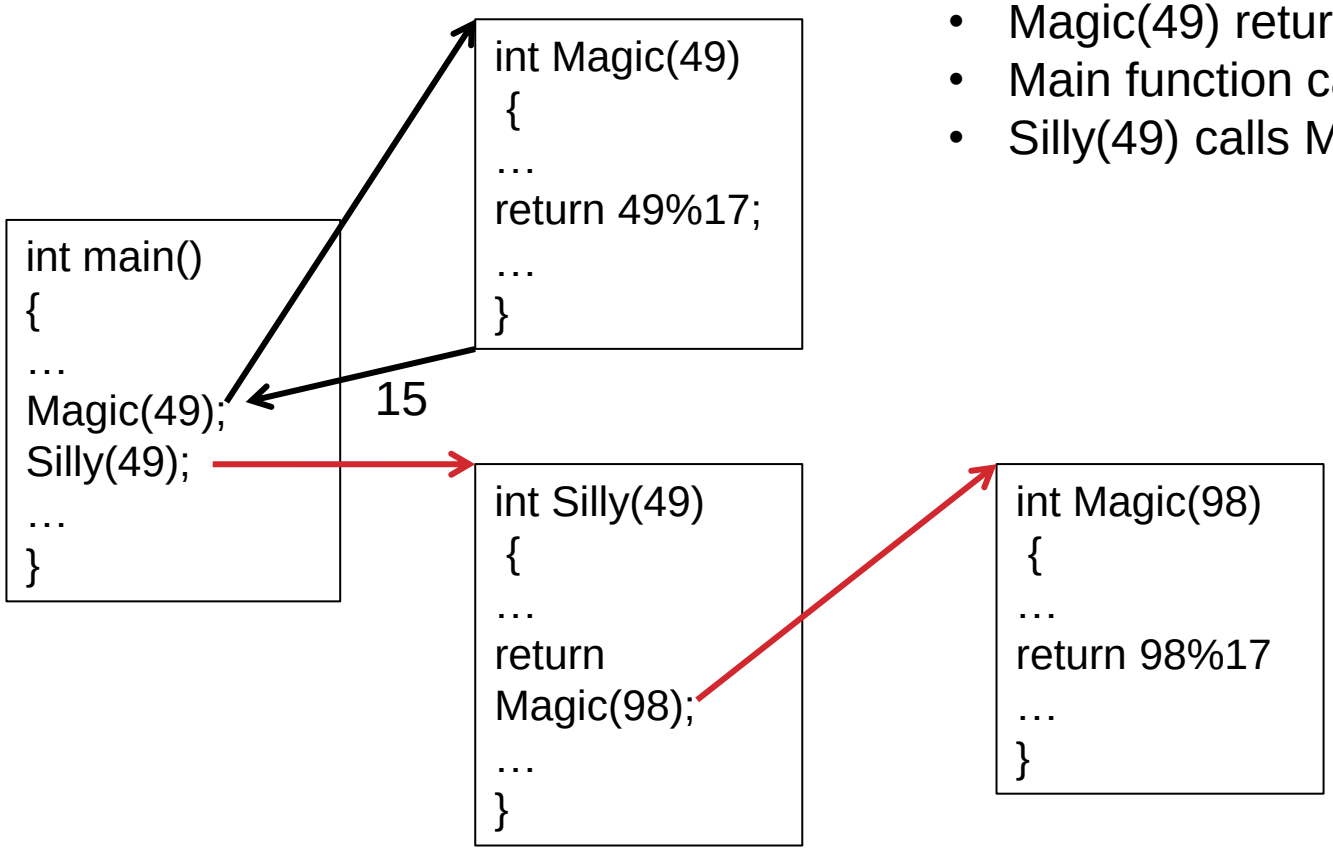
RECURSION



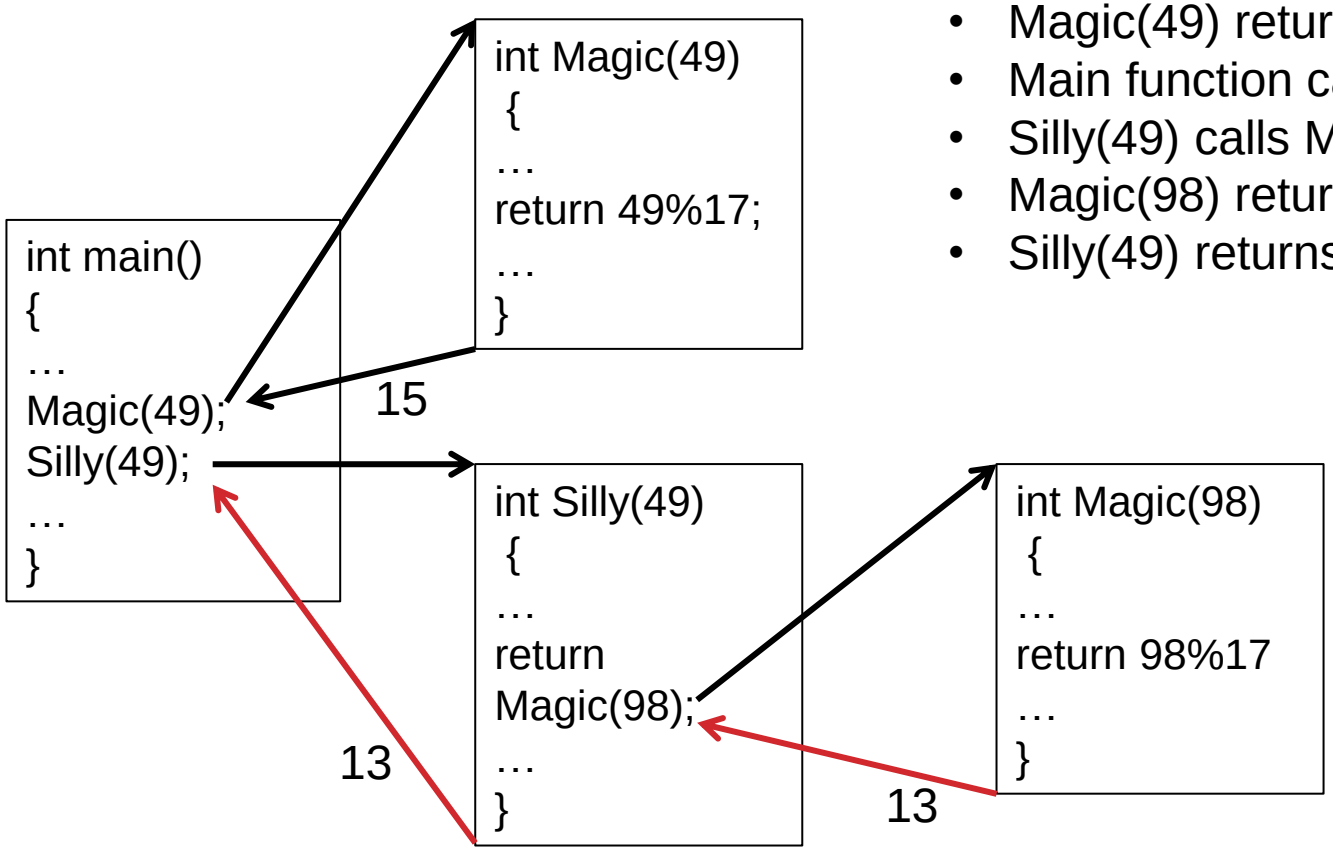
- Main function calls Magic(49)
- Magic(49) returns value 15

RECURSION

- Main function calls Magic(49)
- Magic(49) returns value 15
- Main function calls Silly(49)
- Silly(49) calls Magic(98)



RECURSION



- Main function calls Magic(49)
- Magic(49) returns value 15
- Main function calls Silly(49)
- Silly(49) calls Magic(98)
- Magic(98) returns 13
- Silly(49) returns 13

RECURSION

Sample program output:

```
Enter number: -10

calling magic -10
calling silly -18
Magic(-10) = -5

calling silly -10
Silly(-10) = 3
```

RECURSION

- **Example: calculating N raised to integer power P**

- **Recursive solution solves problem in terms of itself**
  - Pow(N,P) = N * Pow(N, P-1)
  - This is the recurrence relationship

- **We must stop at some point**
  - Pow(N,0) = 1
  - Pow(N,1) = N
  - These are the terminating conditions

RECURSION

```
// Recursive Pow function
int Pow(const int N, const int P)
{
   cout << “calling pow “ << N << “ “ << P << endl;
   if (P == 0)
      return 1;
    else if (P == 1)
      return N;
    else
      return N * Pow(N,P-1);
}
```

We are calling the Pow function with a smaller parameter value

RECURSION

```
// Main program
int main()
{
   int Number, Power;
   cout << "Enter number: ";
   cin >> Number;
   cout << "Enter power: ";
   cin >> Power;
   cout << Pow(Number, Power) << endl;
   return(0);
}
```

RECURSION

Sample program output when number = 2, power = 8:

```
calling pow 2 8
calling pow 2 7
calling pow 2 6
calling pow 2 5
calling pow 2 4
calling pow 2 3
calling pow 2 2
calling pow 2 1
256
```

Notice that the power is **decreasing by one** with each recursive function call and eventually reaches the terminating condition

RECURSION

Sample program output when number = 2, power = -5:

calling pow 2 -5

calling pow 2 -6

calling pow 2 -7

calling pow 2 -8

…

This recursion will go on forever because of a **logic error** in our program

RECURSION

```
// Improved recursive Pow function
double Pow(const double N, const int P)
{
   cout << “calling pow “ << N << “ “ << P << endl;
   if (P == 0)
      return 1;
    else if (P == 1)
      return N;
    else if (P < 0)
      return (1.0 / Pow(N, -P));
   else
      return N * Pow(N,P-1);
}
```

We add another recursive call to deal with negative exponent values

RECURSION

Sample program output when number = 2, power = -3:

Enter number: 2

Enter power: -3

calling pow 2 -3

calling pow 2 3

calling pow 2 2

calling pow 2 1

0.125

RECURSION

- **Can we come up with a faster solution?**
  - Divide the exponent in **half** instead of subtracting one

- **Recurrence relationships**
  - Pow(N,P) = Pow(N, P/2) * Pow(N, P/2) **when P is even**
  - Pow(N,P) = Pow(N, P/2) * Pow(N, P/2) * N **when P is odd**

- **Terminating conditions**
  - Pow(N,0) = 1
  - Pow(N,1) = N

RECURSION

```
double pow2(double N, int P)
{
   // cout << "calling pow2 " << N << " " << P << endl;
   if (P == 0)
      return 1;
   else if (P == 1)
      return N;
   else if (P < 0)
      return (1.0 / Pow(N, -P);
```

RECURSION

```
  else if (P % 2 == 0)
  {
     double temp = pow2(N, P/2);
     return temp * temp;
  }
  else // if (P % 2 == 1)
  {
     double temp = pow2(N, P/2);
     return temp * temp * N;
  }
}
```

Notice that the power is divided in **half** with each recursive call to pow2

RECURSION

Sample program output when number = 2, power = 25:

```
calling pow2 2 25
calling pow2 2 12
calling pow2 2 6
calling pow2 2 3
calling pow2 2 1
33554432
```

We obtain the correct solution after only 5 function calls instead of 25 function calls for the previous implementation

RECURSION

- **Example: calculating the Nth Fibonacci number from the sequence 1,1,2,3,5,8,13,21,34,55…**

- **Recursive solution solves problem in terms of itself**
  - Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
  - This is the recurrence relationship

- **We must stop at some point**
  - Fibonacci(1) = 1
  - Fibonacci(2) = 1
  - These are the terminating conditions

RECURSION

- **This recursive function calls itself twice each time it is called so there are a lot of function calls to calculate the answer**

```
// Recursive Fibonacci function
int Fibonacci(const int Num)
{
   cout << "calling Fibonacci " << Num << endl;
   if (Num <= 2)
      return(1);
   else
      return( Fibonacci(Num-1) + Fibonacci(Num-2) );
}
```

RECURSION

Sample program output when Num = 3:

calling Fibonacci 3

calling Fibonacci 2

calling Fibonacci 1

There are 2 Recursive calls from Fibonacci(3)

RECURSION

Sample program output when Num = 4:

calling Fibonacci 4

calling Fibonacci 3

calling Fibonacci 2

calling Fibonacci 1

calling Fibonacci 2

Recursive calls from Fibonacci(3)

Fibonacci(2) does not have recursive calls

RECURSION

Sample program output when Num = 5:

calling Fibonacci 5

calling Fibonacci 4
calling Fibonacci 3
calling Fibonacci 2
calling Fibonacci 1
calling Fibonacci 2

Recursive calls from Fibonacci(4)

calling Fibonacci 3
calling Fibonacci 2
calling Fibonacci 1

Recursive calls from Fibonacci(3)

RECURSION

- **How can we calculate the number of function calls needed to calculate the Nth Fibonacci number?**

- **The recurrence relationship:**
  - Calls(N) = Calls(N-1) + Calls(N-2) + 1

- **The terminating conditions:**
  - Calls(1) = 1
  - Calls(2) = 1

- **The number of calls is 1,1,3,5,9,15,25,41,67,109…**
  - This grows more quickly than the Fibonacci sequence!

RECURSION

```
// Iterative Fibonacci function
int Fibonacci(const int Num)
{
   int Num1 = 1;
   int Num2 = 1;
   for (int Count = 1; Count < Num; Count++)
   {
      int Num3 = Num1 + Num2;
      Num1 = Num2;
      Num2 = Num3;
   }
   return (Num1);
}
```

- This iterative function is slightly longer than the recursive solution but it is much faster at run time
- Hence choosing the right algorithm can have a huge impact on run time!

SUMMARY

- **In this section we have focused on recursive functions**
  - We need to implement the recurrence relationship
  - We need to implement the terminating conditions

- **Recursion has several pros and cons**
  - Sometimes the recursive solution is smaller and easier to understand than the iterative solution
  - Sometimes recursion is faster when a “divide and conquer” approach is used
  - Sometimes recursion is slower when there are a large number of “redundant” function calls

FUNCTIONS

PART 4

FUNCTION LIBRARIES

FUNCTION LIBRARIES

- **Function libraries are collections of similar functions grouped together in one file for ease of use**
  - By making and using libraries we can increase code reuse and decrease development and debugging time

- **To make use of the functions in a library we need to add #include<library> at the top of our program**
  - This include file contains the function prototypes for all of the functions in the library
  - The compiler will automatically link in code for the functions you have used with your program

FUNCTION LIBRARIES

- **Here are some of the most commonly used libraries**

```
#include <iostream>      // I/O stream functions
#include <iomanip>       // I/O formatting functions
#include <fstream>       // File I/O stream functions
#include <cmath>         // C based math functions
#include <cstdio>        // C based  I/O functions
#include <cstdlib>       // C based library functions
#include <cctype>        // C based character functions
#include <cstring>       // C based string functions
#include <ctime>         // C based timing functions
```

- **See cplusplus.com for full documentation**

CMATH FUNCTIONS

- **The <cmath> library includes a wide range of trigonometric functions and other functions you remember fondly from math class**
  - double sin (double x);
  - double cos (double x);
  - double sqrt (double x);
  - double log (double x);
  - double pow (double base, double exponent);
  - Please use num * num instead of pow(num, 2) !!

  …

- **All of these functions have been implemented very cleverly so they are fast and accurate**

CMATH FUNCTIONS

```
#include <iostream>
#include <cmath>  // old way <math.h>
using namespace std;
int main()
{
   // Calculate and print sin and cos table
   for (int Degrees = 0; Degrees <= 360; Degrees += 10)
   {
      double Radians = Degrees * M_PI / 180.0;
      cout << Degrees << " " << cos(Radians) << "  " << sin(Radians)<< endl;
   }
   return 0;
}
```

IOMANIP FUNCTIONS

- **The <iomanip> library includes functions to help you create well formatted program output**
  - setw(int n);
  - setprecision(int n);
  - setfill(char c);

  …

- **These functions are used together with “cout” to control how the next value will be printed on the screen**

IOMANIP FUNCTIONS

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
int main()
{
   // Print header for table
   cout << setw(8) << "Angle”
            << setw(8) << "Cos”
            << setw(8) << "Sin”;<< endl;
```

IOMANIP FUNCTIONS

```
   // Calculate and print sin and cos table
   for (int Degrees = 0; Degrees <= 360; Degrees += 10)
   {
      double Radians = Degrees * M_PI / 180.0;
      cout << setw(8) << Degrees;
      cout << setw(8) << setprecision(3) << cos(Radians);
      cout << setw(8) << setprecision(3) << sin(Radians) << endl;
   }
   return 0;
}
```

SOFTWARE ENGINEERING TIPS

- **Start your program with "empty" function bodies**
  - Helps debug the main program and parameter passing
- **Implement and test bodies of functions one at a time**
  - This way you are never far from a running program
- **Print debugging messages inside each function**
  - To see which function is being called and its parameters
- **Give functions and parameters meaningful names**
  - Make your code easier to read and understand

SOFTWARE ENGINEERING TIPS

- **Common mistakes when creating functions**
  - Function definition does not match function prototype
  - Semicolon at end of function definition is not needed
  - Infinite loop of recursive function calls

- **Common mistakes when calling functions:**
  - Incorrect number or types of parameters
  - Incorrect ordering of parameters
  - Function return value not used properly

SUMMARY

- **In this section we have introduced C++ function libraries and some example programs that use these libraries**
  - Learning how to use the built in libraries is an important skill for all programmers to develop
  - You do not want to waste your time to reinvent a function that has already been written and debugged

- **We have also discussed several software engineering tips for creating and debugging programs using functions**