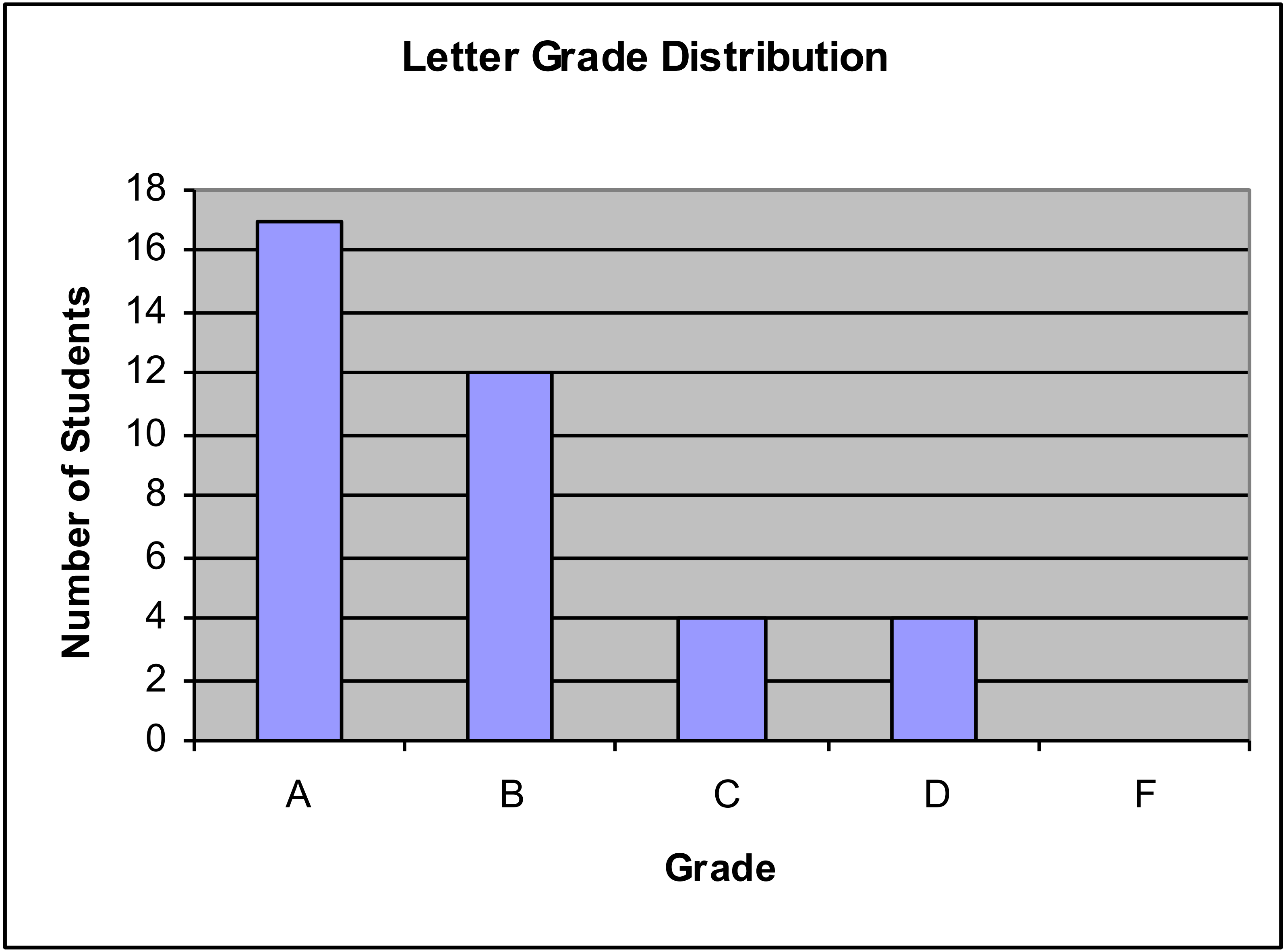
Current Assignments

- Homework 3 is available and is due on Thursday. Iteration and basic functions.

- Exam 1 on Monday. Review on Thursday.

Homework 1


Letter Grade Distribution
Number of Students
18
16
14
12
10
8
6
4
2
0
A
B
C
D
F
Grade

This Time

Introduction to Functions
Math Library Functions
Function Definitions
Function Prototypes
Header Files
Random Number Generation
Example: A Game of Chance and Introducing `enum`
Recursion
Example Using Recursion: The Fibonacci Series
Recursion vs. Iteration
Functions with Empty Parameter Lists
Inline Functions
References and Reference Parameters
Default Arguments
Unary Scope Resolution Operator
Function Overloading
Function Templates

Introduction to Functions

- In mathematics a “function” is said to map a value in its domain to a value in its range
- Implicit in this definition is that someone has to actually perform the algorithm that transforms the value from the domain into the value in the range

Introduction to Functions

- A function then can be thought of as an instruction to perform some algorithm and then plug the result in where the function was
- Functions in older programming languages are often called “subroutines” or “procedures” a terms which capture the idea of a function containing an algorithm.

Introduction to Functions

- Using functions to execute algorithms has several advantages:
  - Instead of having to rewrite an entire algorithm every time you need it you can just call a function you (or someone else) defined earlier
  - Functions make your code much easier to read because they hide its complexity

Introduction to Functions

- "Functional languages" like Scheme or Lisp maintain the mathematical definition so that a function operates on the values it is given and returns a value
- A function in C++ is more like a sub-program. Functions often perform all sorts of operations without taking any values or returning any values

Functions, Side effects

- Functions which take arguments and return a value based only on those arguments are called "functional"
- But functions in C++ can modify all sorts of variables and parameters beyond the variables they were given as arguments
- When a function uses or modifies a variable that was not in its list of arguments it is called "side-effecting"
- Side-effecting is generally discouraged

Writing Functions

When you write a function you have to do two things:

1) Write the function prototype.

   The prototype appears before the function is called and outside the main function

   The prototype tells the compiler how the function can be called

2) Write the function definition.

   This is where the actual code for your function goes. It appears after the main function.

Introduction to Functions

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program

Introduction to Functions

- Programs can use functions that were defined in them or functions that were written by someone else
- There are many, many prepackaged functions for you to use. Prepackaged functions are typically called “libraries”
- Libraries only let you see the function prototypes not the function definitions.

Program Components in C++

- Boss to worker analogy
  - A boss (the calling function or caller) asks a worker (the called function) to perform a task and return (i.e., report back) the results when the task is done.

Math Library Functions

- To perform common mathematical calculations
  - Include the header file **`<cmath>`**
- Functions called by writing
  - functionName(argument1, argument2, …);
- Example

```
cout << sqrt( 900.0 );
```

  - sqrt (square root) function The preceding statement would print 30
  - All functions in cmath return a **`double`**

Math Library Functions

- Function arguments can be
  - Constants
    - `sqrt( 4 );`
  - Variables
    - `sqrt( x );`
  - Expressions
    - `sqrt( sqrt( x ) ) ;`
    - `sqrt( 3 - 6x );`

Method	Description	Example

`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 )` is `1.0`

`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.71828`
		`exp( 2.0 )` is `7.38906`

`fabs( x )`	absolute value of x	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`fmod( x, y )`	remainder of x/y as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`

`log( x )`	natural logarithm of x (base e)	`log( 2.718282 )` is `1.0`
		`log( 7.389056 )` is `2.0`

`log10( x )`	logarithm of x (base 10)	`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`pow( x, y )`	x raised to power y (xy)	`pow( 2, 7 )` is `128`
		`pow( 9, .5 )` is `3`

`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is `0`

`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0`
		`sqrt( 9.0 )` is `3.0`

`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is `0`

Fig. 3.2 Math library functions.

Anatomy of a Function

- Function prototype
  - Tells compiler argument type and return type of function
  - **`int square( int );`**
    - Function takes an **`int`** and returns an **`int`**
  - Explained in more detail later

- Calling/invoking a function
  - **`square(x);`**
  - Parentheses are an operator used to call function
    - Pass argument x
    - Function gets its own copy of arguments
  - After finished, passes back result

Anatomy of a Function

- Format for function definition

  return-value-type function-name **(** *parameter-list* **)**

  {

  declarations and statements

  }

  - Parameter list
    - Comma separated list of arguments
      - Data type needed for each argument
    - If no arguments, use **`void`** or leave blank
  - Return-value-type
    - Data type of result returned (use **`void`** if nothing returned)

Anatomy of a Function

- Example function

```
int square( int y )
{
   return y * y;
}
```

- **`return`** keyword
  - Returns data, and control goes to function's caller
    - If no data to return, use **`return;`**
  - Function ends when reaches right brace
    - Control goes to caller

Function Prototypes

- **Function prototypes contain**
  - Function name
  - Parameters (number and data type)
  - Return type (`void` if returns nothing)
  - Only needed if function definition after function call
- **Prototype must match function definition**
  - Function prototype

```
double maximum( double, double, double );
```

  - Function Definition

```
double maximum( double x, double y, double
  z )
{
 …
}
```

```
 // Writing a function example

 #include <iostream>

int square( int );  // functi

int main()
{
      cout << square( x ) << "   “ <<

   return 0;  // indicates successful termination

} // end main

// square function definition returns squ
int square( int y )  // y is a copy of a
{
   return y * y;    // returns square of

} // end function square
```

Function prototype: specifies data types of arguments and return values. **square** expects an **int**, and returns an **int**.

Parentheses **()** cause function to be called. When done, it returns the result.

The function definition contains the actual code to run when the square is called

```
// Finding the maximum of three floating-point numbers.
   #include <iostream>

  // Function prototype
  double maximum( double x, double y, double

  int main()
  {
     double number1, number2

     cout << "Enter three fl                        s: ";
     cin >> number1 >> numbe

     // number1, number2 and number3 are arguments to
     // the maximum function call
    cout << "Maximum is: "
         << maximum( number1, number2, number3 ) << endl;

     return 0;  // indicates successful termination
```

Comma separated list for multiple parameters.

Function **`maximum`** takes 3 arguments (all **`double`**) and returns a **`double`**.

```
  } // end main

// function maximum definition;
// x, y and z are parameters
double maximum( double x, double y, double
{
   double max = x;      // assume x is largest

   if ( y > max )       // if y is larger,
      max = y;          // assign y to max

   if ( z > max )       // if z is larger,
      max = z;          // assign z to max

   return max;          // max is largest value

} // end function maximum
```

The Function definition should look exactly like the function prototype

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32
```

Function Signatures

- Function signature
  - Part of prototype with name and parameters
    - `double maximum( double, double, double );`

      Function signature
- The function signature is how the compiler figures out what function you are trying to call and whether you are calling it correctly
- You can give different functions the same name
- You cannot create two functions with the same signature
- Writing two or more functions with the same name but different signatures is called "function overloading"

Argument Coercion

- Remember our discussion of how chars can be treated as ints

- Argument Coercion
  - Forces arguments to be of the type specified on the prototype
    - Converting **`int`** (4) to **`double`** (4.0)

```
cout << sqrt(4)
```

Function Overloading

- Function overloading is used frequently and can be very useful.
- If your function needs to do slightly different things based on the type of arguments it received then function overloading simplifies things
- Instead of the user having to remember three different user defined functions print_int(int), print_float(float), print_char(char), with function overloading they can just remember one function name print() and let the system decide which version to call based on the argument type.

Function Overloading

- The operators we have seen like + and / are special versions of functions that take two arguments.
- These functions are overloaded so that you don’t have to use float/ when dividing floating point numbers or int/ with integers.
- This is also how the stream insertion operator is able to print any basic type you give it. You are actually calling a different function when you write

`cout << x` than when you write `cout << y;`

if x is an int and y is a float

Argument Coercion

- Conversion rules
  - Arguments are usually **`cast`** automatically
  - Changing from **`double`** to **`int`** can truncate data -3.4 to 3
- Most compilers will warn you if a truncation occurs
- e.g. This is what MSVC6 tells you:

```
warning C4244: '=' : conversion
from 'double' to 'float',
possible loss of data
```

Function Argument Coercion

Data types	
long double	
double	
float	
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char	
bool	(false becomes 0, true becomes 1)

Fig. 3.5 Promotion hierarchy for built-in data types.

Writing Functions, example

```
float power( float base, float x );  // Function prototype

int main()  // main function, called by operating system
{
        float n = 10.0, x = 2.0, result = 0.0;
        result = power( x, n );
        return 0;
}

float power( float base, float x )
{
        float answer = 0.0;
        for( int i  = 0; i < x; i++ )
        {
            answer = answer * base;
        }
        return answer;
}
```

Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Header files ending with .h
  - Programmer-defined header files

```
#include “myheader.h”
```

- Library header files

```
#include <cmath>
```

Random Number Generation

- `rand` function (`<cstdlib>`)
  - `i = rand();`
  - Generates unsigned integer between 0 and RAND_MAX (usually 32767)
- Scaling and shifting
  - Modulus (remainder) operator: `%`
    - `10 % 3` is `1`
    - `x % y` is between `0` and `y - 1`
  - Example

    ```
    i = rand() % 6 + 1;
    ```

    - "`Rand() % 6`" generates a number between `0` and `5` (scaling)
    - "`+ 1`" makes the range 1 to 6 (shift)
  - Next: program to roll dice

Random Number Generation

- Calling rand() repeatedly
  - Gives the same sequence of numbers
- Pseudorandom numbers
  - Preset sequence of "random" numbers
  - Same sequence generated whenever program run
- To get different random sequences
  - Provide a seed value
    - Like a random starting point in the sequence
    - The same seed will give the same sequence
  - **`srand(seed);`**
    - **`<cstdlib>`**
    - Used before **`rand()`** to set the seed

Random Number Generation

- If you call rand in two separate runs of your program you will get the same sequence of "random" numbers.
- To aviod this you have to set the "seed"
- Can use the current time to set the seed
  - No need to explicitly set seed every time
  - **`srand( time( 0 ) );`**
  - **`time( 0 );`**
    - **`<ctime>`**
    - Returns current time in seconds
- General shifting and scaling
  - *Number* **=** *shiftingValue* **+ `rand()` `%`** *scalingFactor*
  - shiftingValue = first number in desired range
  - scalingFactor = width of desired range

Example: Game of Chance and Introducing enum

- Enumeration
  - Set of integers with identifiers

  `enum` *typeName* **`{`***constant1*, *constant2…***`};`**
  - Constants start at 0 (default), incremented by 1
  - Constants need unique names
  - Cannot assign integer to enumeration variabl
  - Must use a previously defined enumeration type
- Example

```
enum Status {CONTINUE, WON, LOST};
Status enumVar;
enumVar = WON; // cannot do enumVar = 1
```

Example: Game of Chance and Introducing enum

- Enumeration constants can have preset values

```
enum Months { JAN = 1, FEB, MAR, APR,
  MAY, JUN, JUL, AUG, SEP, OCT, NOV,
  DEC};
```

  - Starts at 1, increments by 1
- Next: craps simulator
  - Roll two dice
  - 7 or 11 on first throw: player wins
  - 2, 3, or 12 on first throw: player loses
  - 4, 5, 6, 8, 9, 10
    - Value becomes player's "point"
    - Player must roll his point before rolling 7 to win

```
// Game of Craps.
#include <iostream>
using namespace std;

// contains function prototypes for functions srand and rand
#include <cstdlib>
// contains prototype for function time
#include <ctime>

int rollDice( void );                              type

int main()
{
   // enumeration constants represent game s
   enum Status { CONTINUE, WON, LOST };

   int sum, myPoint;

   Status gameStatus;  // can contain CONTINUE, WON or LOST
```

Function to roll 2 dice and return the result as an **`int`**.

Enumeration to keep track of the current game’s status.

```
// randomize random number generator using current time
srand( time( 0 ) );

sum = rollDice();   // first roll of the dice

// determine game sta                         on sum of dice
switch ( sum ) {

   // win on first roll
   case 7:
   case 11:
      gameStatus = WON;
      break;

   // lose on first roll
   case 2:
   case 3:
   case 12:
      gameStatus = LOST;
      break;
```

switch statement determines outcome based on die roll.

```
default: // remember point (the number to roll again)
      gameStatus = CONTINUE;
      myPoint = sum;
      cout << "Point is " << myPoint << endl;
      break;                   // optional

} // end switch

// while game not complete ...
while ( gameStatus == CONTINUE ) {
   sum = rollDice();          // roll dice again

   // determine game status
   if ( sum == myPoint )       // win by making point
      gameStatus = WON;
   else
      if ( sum == 7 )          // lose by rolling 7
         gameStatus = LOST;

} // end while
```

```
   if ( gameStatus == WON ) // display won or lost message
     {
       cout << "Player wins" << endl;
     }
   else
     {
      cout << "Player loses" << endl;
     }
   return 0;  // indicates successf
} // end main

// roll dice, calculate sum and display results
int rollDice( void )
{
   int die1 = 0, die2 = 0, workSum = 0;
   die1 = 1 + rand() % 6;  // pick random die1 value
   die2 = 1 + rand() % 6;  // pick random die2 value
   workSum = die1 + die2; // sum die1 and die2
   // display results of this roll
   cout << "Player rolled " << die1 << " + " << die2
        << " = " << workSum << endl;
   return workSum;          // return sum of dice
} // End function rollDice
```

Function **rollDice** takes no arguments, so has **void** in the parameter list.

```
Player rolled 2 + 5 = 7
Player wins
Player rolled 6 + 6 = 12
Player loses
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
    - Answer works way back up, solves entire problem

Recursion

- Example: factorial

  $n! = n * (n - 1) * (n - 2) * \ldots * 1$

  – Recursive relationship (n! = n * (n – 1)!)

  $5! = 5 * 4!$

  $4! = 4 * 3! \ldots$

  – Base case (1! = 0! = 1)

```
// Program to print 0!...10!
   #include <iostream>
   #include <iomanip>

  // Recursive factorial funct
  unsigned long factorial(unsigned long );

  int main()
  {
     // Loop 10 times. During each iteration, calculate
     // factorial( i ) and display result.
     for ( int i = 0; i <= 10; i++ )
        cout << i << "! = " << factorial( i ) << endl;

     return 0;  // indicates successful termination

} // end main
```

Data type **unsigned long** can hold an integer from 0 to 4 billion.

```
// recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
   // base case
   if ( number > 1 )
   {
    number *= factorial( number - 1 );
   }
   else
   {
    number = 1;
  }

   return number;
} // end function factorial
```

The base case occurs when we have 0! or 1!. All other cases must be split up (recursive step).

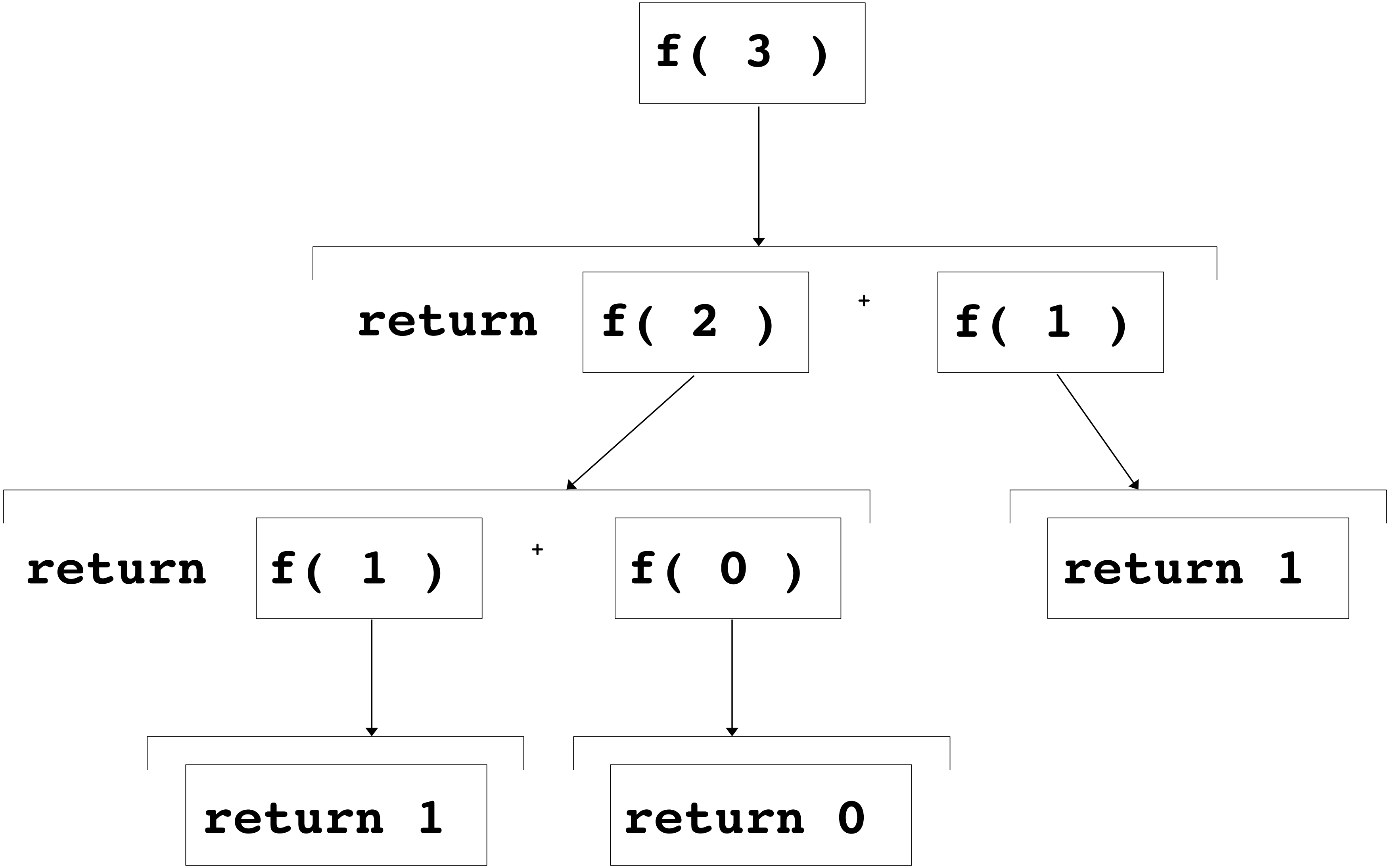
```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Example Using Recursion: Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of two previous ones
  - Example of a recursive formula:
    - *fib(n) = fib(n-1) + fib(n-2)*
- C++ code for Fibonacci function

```
long fibonacci( long n )
{
  if ( n == 0 || n == 1 )  // base case
      return n;
  else
   return fibonacci( n - 1 ) +
          fibonacci( n - 2 );
}
```

Example Using Recursion: Fibonacci Series



f(3)

return f(2) + f(1)

return f(1) + f(0)

return 1

return 1

return 0

Example Using Recursion: Fibonacci Series

- Order of operations
  - **`return fibonacci( n - 1 ) + fibonacci( n - 2 );`**
- Do not know which one executed first
  - C++ does not specify
  - Only **`&&, ||`** and **`?:`** guaranteed left-to-right evaluation
- Recursive function calls
  - Each level of recursion doubles the number of function calls
    - 30th number = 2^30 ~ 4 billion function calls
  - Exponential complexity

```
  // Recursive fibonacci function.
  #include <iostream>

  unsigned long fibonacci( unsigned long ); //

 int main()
  {
    unsigned long result, number;

    // obtain integer from user
    cout << "Enter an integer: ";
    cin >> number;

    // calculate fibonacci value for number input by user
    result = fibonacci( number );

    // display result
    cout << "Fibonacci(" << number << ") = " << result << endl;

    return 0;  // indicates successful termination
}
```

The Fibonacci numbers get large very quickly, and are all non-negative integers. Thus, we use the **`unsigned long`** data type.

```
// recursive definition of function fibonacci
 unsigned long fibonacci( unsigned long n )
 {
    // base case
    if ( n == 0 || n == 1 )
     {
       return n;
      }
    // recursive step
    else
     {
       return fibonacci( n - 1 ) + fibonacci( n - 2 );
     }
 } // end function fibonacci
```

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and elegance (recursion)
- Some languages, like Scheme, Prolog, and Lisp use recursion for almost everything.

Functions with Empty Parameter Lists

- Empty parameter lists
  - **`void`** or leave parameter list empty
  - Indicates function takes no arguments
  - Function **`print`** takes no arguments and returns no value
    - **`void print();`**
    - **`void print( void );`**

```cpp
// Functions that take no arguments.
#include <iostream>

 using std::cout;
 using std::endl;

 void function1();        // function prototype
 void function2( void );  // function prototype

int main()
{
   function1();  // call function1 with no arguments
   function2();  // call function2 with no arguments

   return 0;     // indicates successful termination

} // end main
```

```
// function1 uses an empty parameter list to specify that
// the function receives no arguments
void function1()
{
   cout << "function1 takes no arguments" << endl;

} // end function1

// function2 uses a void parameter list to specify that
// the function receives no arguments
void function2( void )
{
   cout << "function2 also takes no arguments" << endl;

} // end function2
```

Inline Functions

- Inline functions
  - Keyword **`inline`** before function
  - Asks the compiler to copy code into program instead of making function call
    - Reduce function-call overhead
    - Compiler can ignore **`inline`**
  - Good for small, often-used functions
- Example

```
inline double
 cube( double s )
   { return s * s * s; }
   // Using an inline function to calculate.
   // the volume of a cube.
  #include <iostream>

// Definition of inline function cube. Definition of
// function appears before function is called, so a
// function prototype is not required. First line of
// function definition acts as the prototype.

  inline double cube( const double side )
  {
     return side * side * side;  // calculate cube
  } // end function cube
```

```
int main()
{
   double side = -1.0;
   cout << "Enter the side length of your cube:  ";
   cin >> side;

   // calculate cube of sideValue and display result
   cout << "Volume of cube with side "
        << side << " is " << cube( side ) << endl;

   return 0;  // indicates successful termination

} // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

References and Reference Parameters

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original

References and Reference Parameters

- Reference parameter
  - Alias for argument in function call
    - Passes parameter by reference
  - Use **`&`** after data type in prototype
    - **`void myFunction( int &data )`**
    - **`data`** is a *reference* to an **`int`**
  - Function call format the same
    - However, the original variable can now be changed

```
// Comparing pass-by-value and pass-by-reference
// with references.
#include <iostream>

using std::cout;
using std::endl;

int squareByValue( int );           // func
void squareByReference( int & );  // function prototype

int main()
{
   int x = 2;
   int z = 4;

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
        << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;
```

Notice the **&** operator, indicating pass-by-reference.

```
   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;

   return 0;  // indicates successful termination
} // end main

// squareByValue multiplies number by it
// result in number and returns the new
int squareByValue( int number )
{
   return number *= number;  // caller's argument not modified

} // end function squareByValue

// squareByReference multiplies numberRef by
// stores the result in the variable to whic
// refers in function main
void squareByReference( int &numberRef )
{
   numberRef *= numberRef;   // caller's arg

} // end function squareByReference
```

Changes **number**, but original parameter (**x**) is not modified.

Changes **numberRef**, which is a reference to the variable being passed in. Thus, **z** is changed.

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

References and Reference Parameters

- Pointers (week 5)
  - Another way to pass-by-reference
- References as aliases to other variables
  - Refer to same variable
  - Can be used within a function

```
int count = 1; // declare integer
 variable count
Int &cRef = count; // create cRef as
 an alias for count
++cRef; // increment count (using
 its alias)
```

- References must be initialized when declared
  - Otherwise, compiler error
  - Dangling reference
    - Reference to undefined variable

```
// References must be initialized.
#include <iostream>
int main()
{
   int x = 3;
   // y refers to (is an alias for) x
   int &y = x;

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;  // indicates successful termination
} // end main
```

y declared as a reference to **x**.

```
x = 3
y = 3
x = 7
y = 7
```

Default Arguments

- Function call with omitted parameters
  - If not enough parameters passed in by the caller, the rightmost go to their defaults
  - Default values
    - Can be constants, global variables, or function calls
- Set defaults in function prototype

```
int myFunc(int x=1,int y=2,int z=3);
```

  - `myFunc(3)`
    - **`x = 3`**, **`y`** and **`z`** get defaults (rightmost)
  - `myFunc(3, 5)`
    - **`x = 3`**, **`y = 5`** and **`z`** gets default

```
// Using default arguments.
   #include <iostream>

   // function prototype that specifies de
   int boxVolume( int length = 1, int widt

  int main()
  {
     // no arguments--use default values for all dimensions
     cout << "The default box volume is: " << b

     // specify length; default width and heigh
     cout << "\n\nThe volume of a box with leng
          << "width 1 and height 1 is: " << box

     // specify length and width; default height
     cout << "\n\nThe volume of a box with length 10,\n"
          << "width 5 and height 1 is: " << boxVolume( 10, 5 );
```

Set defaults in function prototype.

Function call with some parameters missing – the rightmost parameters get their defaults.

```
   // specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
        << endl;

   return 0;  // indicates successful termination

} // end main

// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;

} // end function boxVolume
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Function Overloading

- Function overloading
  - Functions with same name and different parameters
  - Should perform similar tasks
    - i.e., function to square **`int`**s and function to square **`float`**s

      ```
      int square( int x) {return x * x;}
      float square(float x) { return x * x; }
      ```
- Overloaded functions distinguished by signature
  - Based on name and parameter types (order matters)
  - Name mangling
    - Encodes function identifier with parameters
  - Type-safe linkage
    - Ensures proper overloaded function called

```
// Using overloaded functions.
 #include <iostream>

 // function square for int
 int square( int x )
{
   cout << "Called square with int argument: " << x << endl;
   return x * x;

} // end int version of function square

// function square for double values
double square( double y )
{
   cout << "Called square with double argument: " << y << endl;
   return y * y;

} // end double version of function square
```

Overloaded functions have the same name, but the different parameters types.

```
int main()
{
   int intResult = square( 7 );         // calls int version
   double doubleResult = square( 7.5 ); // calls double version

   cout << "\nThe square of integer 7 is " << intResult
        << "\nThe square of double 7.5 is " << doubleResult
        << endl;

   return 0;  // indicates successful

} // end main
```

The argument type determines which function gets called (**`int`** or **`double`**).

```
Called square with int argument: 7
Called square with double argument: 7.5

The square of integer 7 is 49
The square of double 7.5 is 56.25
```

```
  // Name mangling.

  // function square for int values
 int square( int x )
  {
     return x * x;
  }

 // function square for double values
 double square( double y )
{
    return y * y;
 }

 // function that receives arguments of types
 // int, float, char and int *
void nothing1( int a, float b, char c, int *d )
 {
    // empty function body
 }
```

```
  // function that receives arguments of types
  // char, int, float * and double *
  char *nothing2( char a, int b, float *c, double *d )
  {
     return 0;
  }

  int main()
 {
     return 0;  // indicates successful termination

} // end main
```

```
_main
@nothing2$qcipfpd
@nothing1$qifcpi
@square$qd
@square$qi
```

Mangled names produced in assembly language.

$q separates the function name from its parameters. **c** is **char**, **d** is **double**, **i** is **int**, **pf** is a pointer to a **float**, etc.

LAB (45 min)

- Write three functions:

  minarg(char arg1, char arg2, char arg3);

  minarg(float arg1, float arg2, float arg3);

  minarg(int arg1, int arg2, int arg3);

That each return an integer indicating which of their arguments is the smallest i.e. if the function returns 1 then arg1 was smallest, if 3 then arg3 is the "least."

Then write a program to get three values of each type from the user, call the three functions, and print the results.

Function Templates

- Compact way to make overloaded functions
  - Generate separate function for different data types
- Format
  - Begin with keyword **`template`**
  - Formal type parameters in brackets **`<>`**
    - Every type parameter preceded by **`typename`** or **`class`** (synonyms)
    - Placeholders for built-in types (i.e., **`int`**) or user-defined types
    - Specify arguments types, return types, declare variables
  - Function definition like normal, except formal types used

Function Templates

- Example

  ```
  template < class T > // or template< typename T >
  T square( T value1 )
  {
     return value1 * value1;
  }
  ```

  - `T` is a formal type, used as parameter type
    - Above function returns variable of same type as parameter
  - In function call, T replaced by real type
    - If `int`, all `T`'s become `int`s

      ```
      int x;
      int y = square(x);
      ```

```
 // Using a function template.
 #include <iostream>

 // definition of function ter
template < class T >  // or te
T maximum( T value1, T value2, T value3 )
{
   T max = value1

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   return max;

} // end function template maximum
```

Formal type parameter **T** placeholder for type of data to be tested by **maximum**.

maximum expects all parameters to be of the same type.

```
int main()
{
   // demonstrate maximum with int values
   int int1, int2, int3;

   cout << "Input three integer values:
   cin >> int1 >> int2 >> int3;

   // invoke int version of maximum
   cout << "The maximum integer value is: "
        << maximum( int1, int2, int3 );

   // demonstrate maximum with double values
   double double1, double2, double3;

   cout << "\n\nInput three double values: ";
   cin >> double1 >> double2 >> double3;

   // invoke double version of maximum
   cout << "The maximum double value is: "
        << maximum( double1, double2, double3 );
```

`maximum` called with various data types.

```
      // demonstrate maximum with char values
      char char1, char2, char3;
     cout << "\n\nInput three characters: ";
      cin >> char1 >> char2 >> char3;

      // invoke char version of maximum
      cout << "The maximum character value is: "
           << maximum( char1, char2, char3 )
           << endl;

      return 0;  // indicates successful termination
   } // end main
```

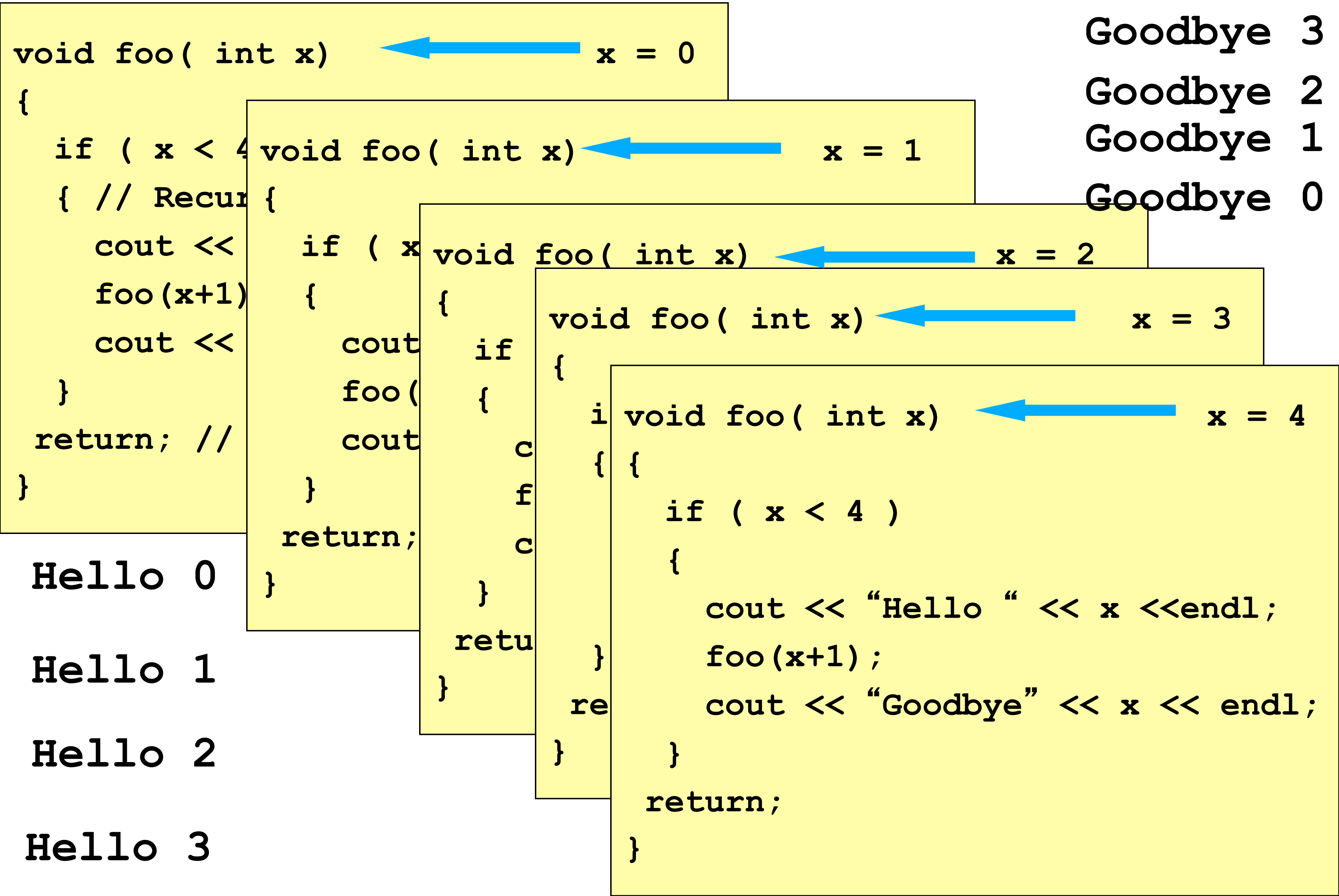
```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Recursion (Frames and The Stack)

```
foo(0);
```



```
void foo( int x)          x = 0
{
  if ( x < 4
  { // Recur
    cout <<
    foo(x+1)
    cout <<
  }
 return; //
}
```

```
void foo( int x)          x = 1
{
  if ( x
  {
    cout
    foo(
    cout
  }
 return;
}
```

```
void foo( int x)          x = 2
{
  if
  {
    c
    f
    c
  }
 retu
}
```

```
void foo( int x)          x = 3
{
  i
  {
  }
 re
}
```

```
void foo( int x)          x = 4
{
  if ( x < 4 )
  {
    cout << “Hello “ << x <<endl;
    foo(x+1);
    cout << “Goodbye” << x << endl;
  }
 return;
}
```

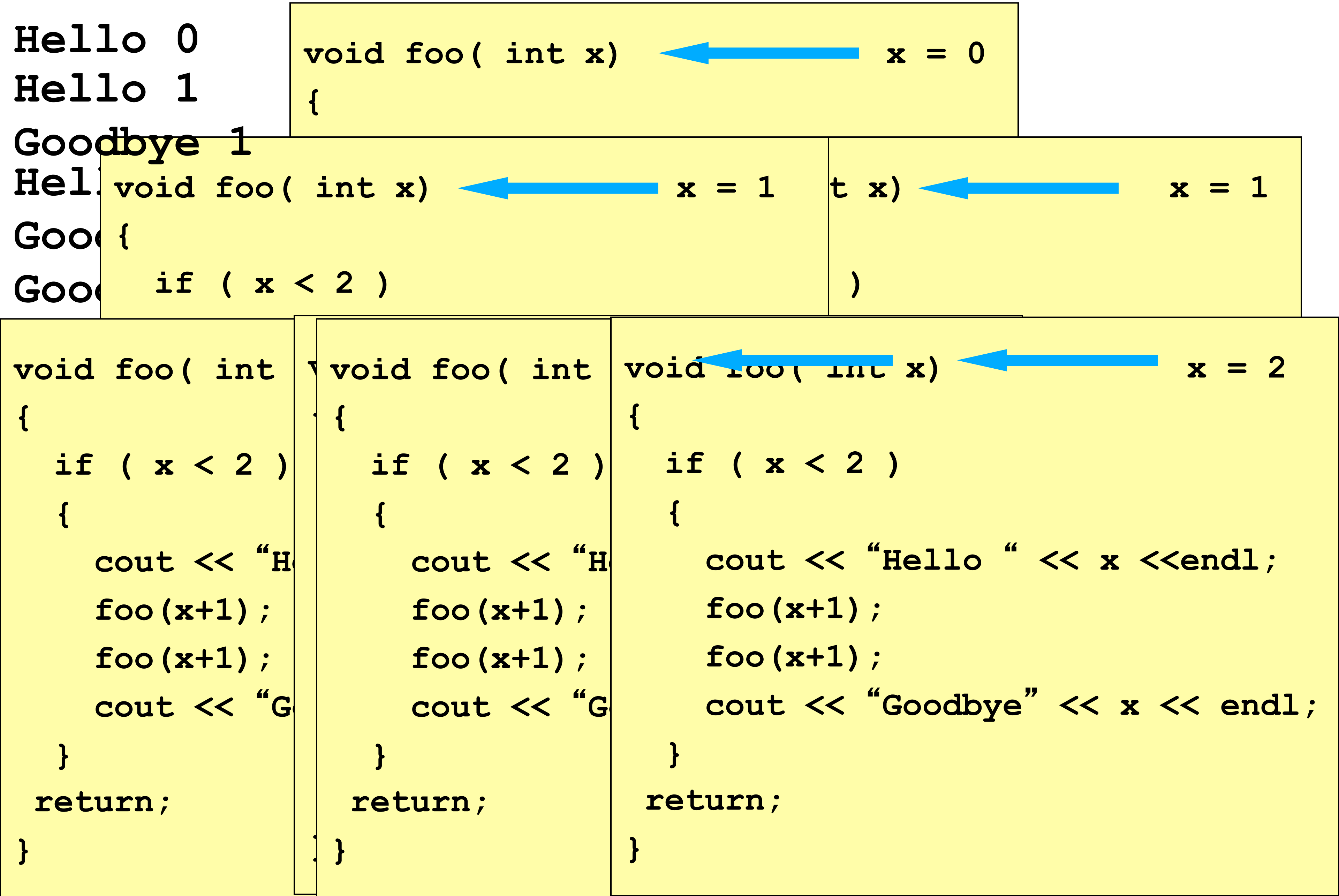
```
Hello 0
Hello 1
Hello 2
Hello 3
```

```
Goodbye 3
Goodbye 2
Goodbye 1
Goodbye 0
```

Recursion (Frames and The Stack)

```
foo(0)
```

```
Hello 0
Hello 1
Goodbye 1
Hel
Good
Good
```



```
void foo( int x)
{
```

x = 0

```
void foo( int x)
{
  if ( x < 2 )
```

x = 1

```
t x)

 )
```

x = 1

```
void foo( int
{
  if ( x < 2 )
  {
    cout << "H
    foo(x+1);
    foo(x+1);
    cout << "G
  }
 return;
}
```

```
void foo( int
{
  if ( x < 2 )
  {
    cout << "H
    foo(x+1);
    foo(x+1);
    cout << "G
  }
 return;
}
```

```
void foo( int x)
{
  if ( x < 2 )
  {
    cout << "Hello " << x <<endl;
    foo(x+1);
    foo(x+1);
    cout << "Goodbye" << x << endl;
  }
 return;
}
```

x = 2

Recursion

- Program Trace
- Fib
- Fact

- Visualization of Recursion with Java

  http://www.iol.ie/~jmchugh/csc302/