Functions

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

This week!

- Function Basics
- Call-by-value
- Call-by-reference in C++
- Parameters and Arguments
- Function overloading
- Default arguments
- Inline functions

FUNCTION BASICS

Sample Code

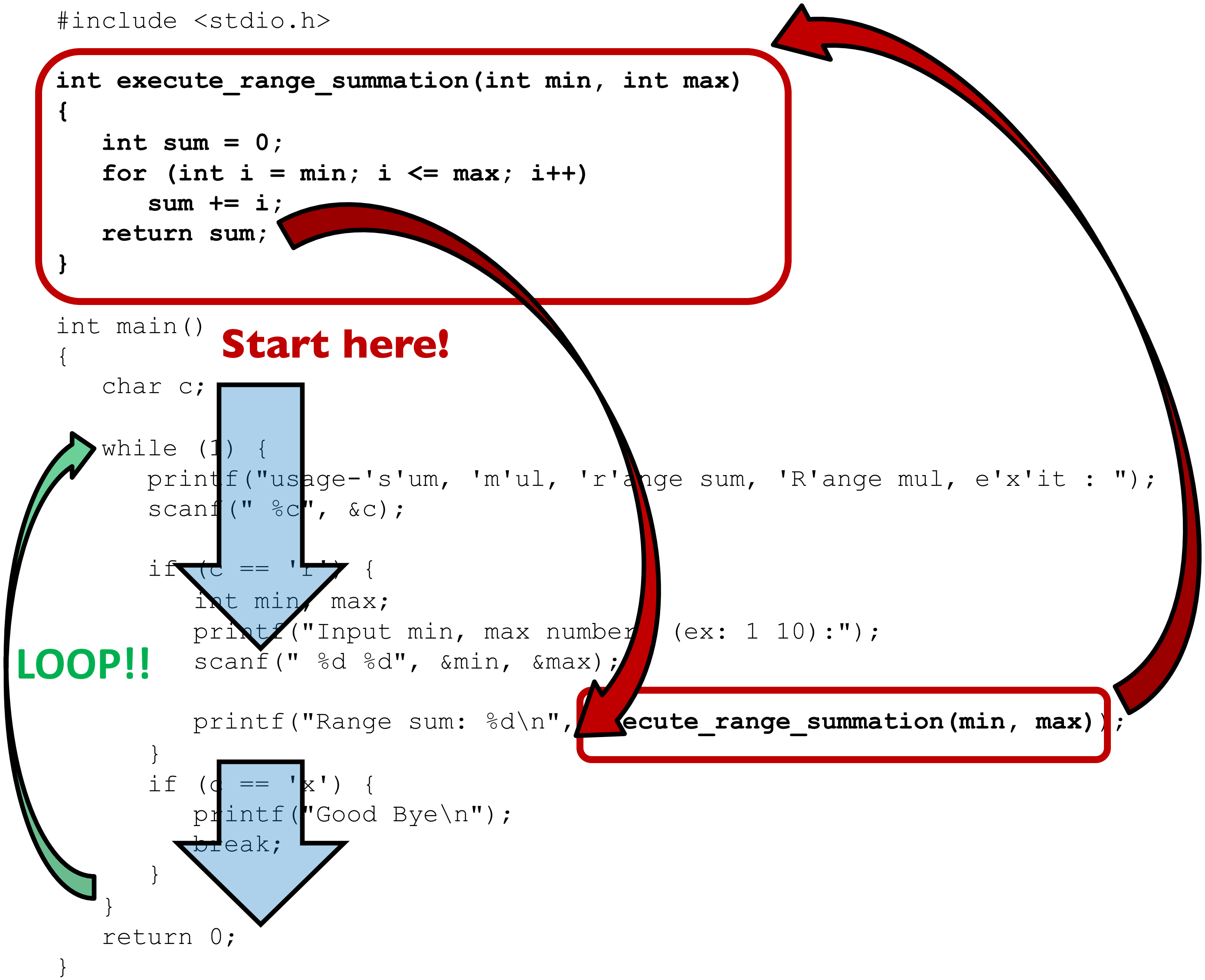
```
#include <stdio.h>

int execute_range_summation(int min, int max)
{
   int sum = 0;
   for (int i = min; i <= max; i++)
      sum += i;
   return sum;
}
```

Function Definition

```
int main()
{
   char c;

   while (1) {
      printf("usage-'s'um, 'm'ul, 'r'ange sum, 'R'ange mul, e'x'it : ");
      scanf(" %c", &c);

      if (c == 'r') {
         int min, max;
         printf("Input min, max numbers (ex: 1 10):");
         scanf(" %d %d", &min, &max);

         printf("Range sum: %d\n", execute_range_summation(min, max));
      }
      if (c == 'x') {
         printf("Good Bye\n");
         break;
      }
   }
   return 0;
}
```

Function Call

Control Flow for Functions



```
#include <stdio.h>

int execute_range_summation(int min, int max)
{
   int sum = 0;
   for (int i = min; i <= max; i++)
      sum += i;
   return sum;
}

int main()
{
   char c;

   while (1) {
      printf("usage-'s'um, 'm'ul, 'r'ange sum, 'R'ange mul, e'x'it : ");
      scanf(" %c", &c);

      if (c == 'r') {
         int min, max;
         printf("Input min, max numbers (ex: 1 10):");
         scanf(" %d %d", &min, &max);

         printf("Range sum: %d\n", execute_range_summation(min, max));
      }
      if (c == 'x') {
         printf("Good Bye\n");
         break;
      }
   }
   return 0;
}
```

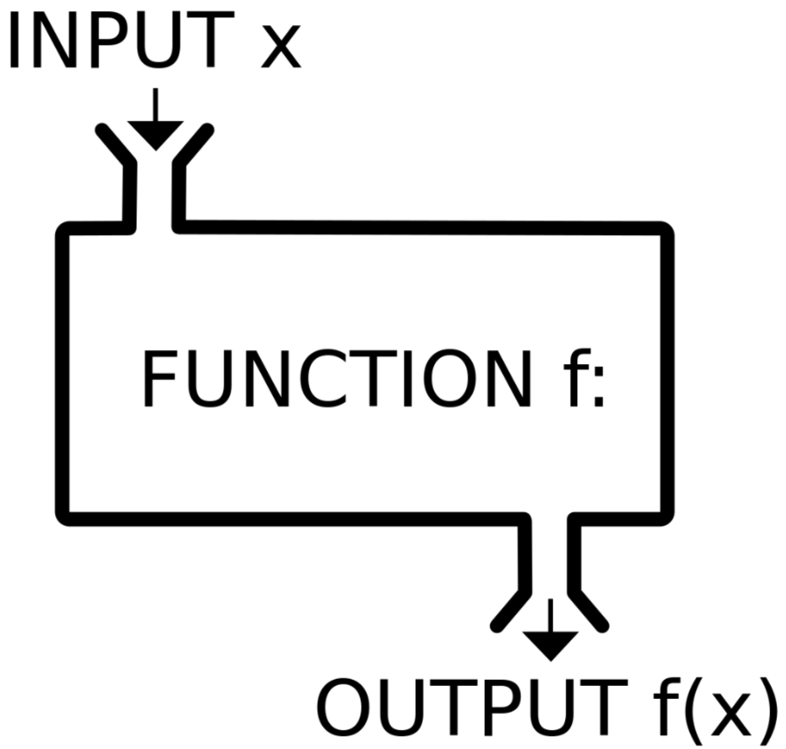
Start here!

LOOP!!

What is a function in programming?

- Inputs and Outputs

$$f(x) = ax^2+bx+c$$

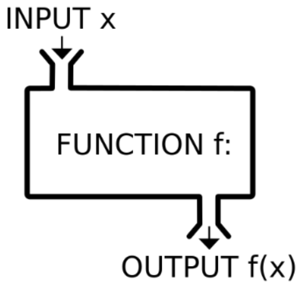


INPUT x

FUNCTION f:

OUTPUT f(x)

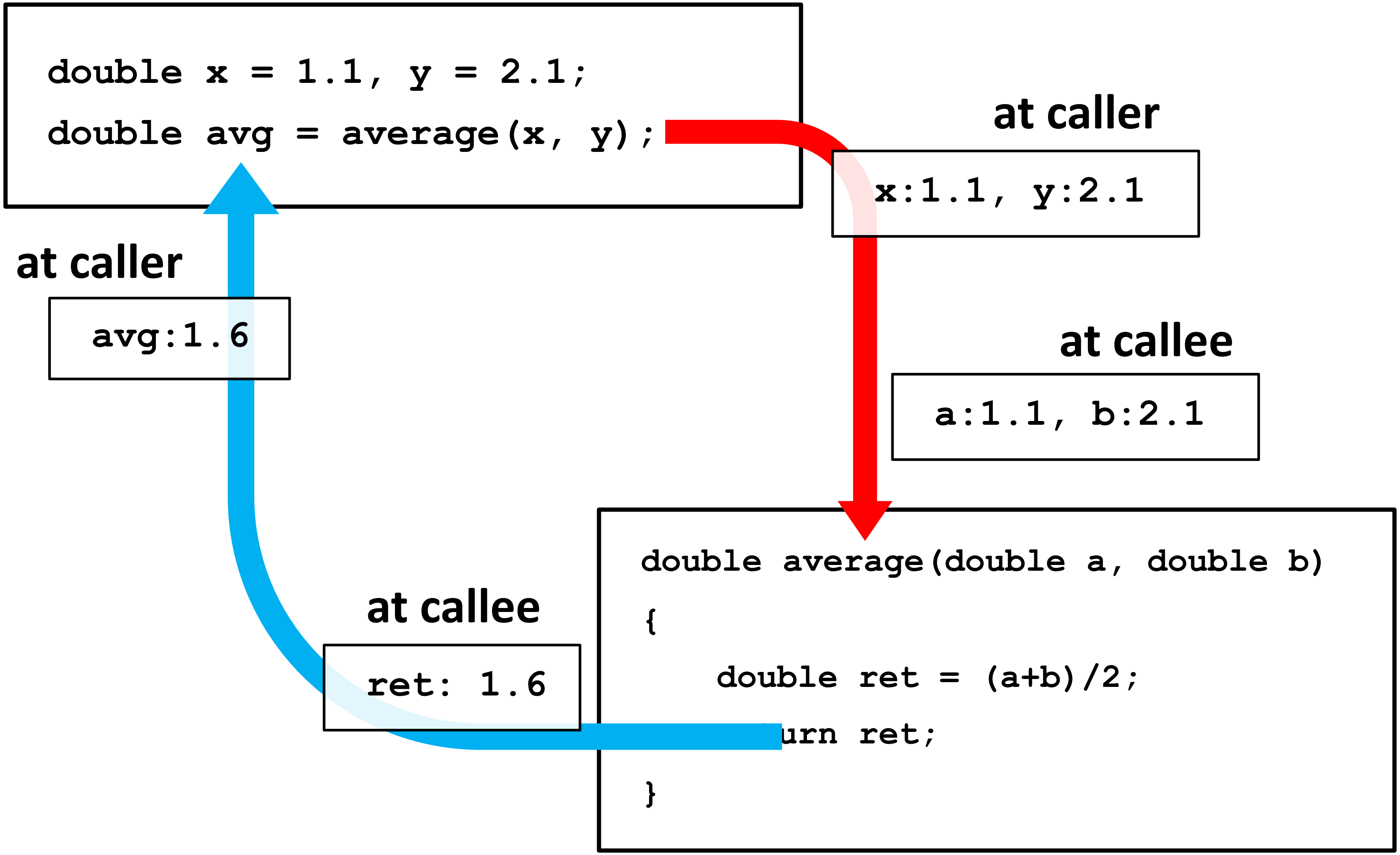
Functions (overview)



- A function
  - is a series of statements that have been grouped together and given a name.
  - divides a program into small pieces (easier to understand, ***readability***)
  - can be reused on the same (maybe similar) operations *(**reusability)***

- Inputs and outputs for functions
  - one or more inputs and outputs
  - There can be no inputs and outputs. (`void` type)

Parameter Passing Example



```
double x = 1.1, y = 2.1;
double avg = average(x, y);
```

at caller

```
x:1.1, y:2.1
```

at callee

```
a:1.1, b:2.1
```

```
double average(double a, double b)
{
    double ret = (a+b)/2;
    return ret;
}
```

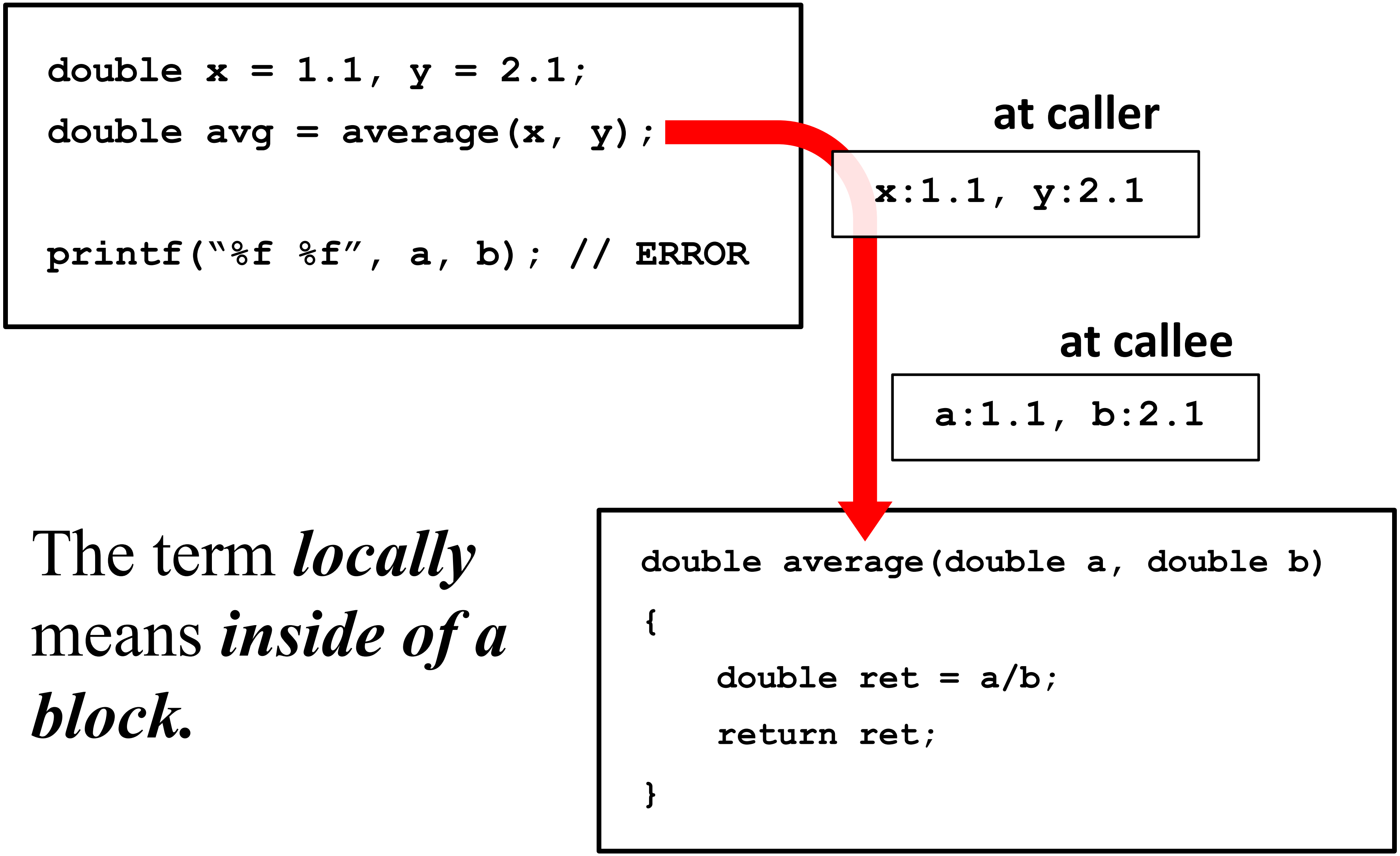
at callee

```
ret: 1.6
```

at caller

```
avg:1.6
```

Scope for Parameters



```
double x = 1.1, y = 2.1;
double avg = average(x, y);

printf(“%f %f”, a, b); // ERROR
```

at caller

```
x:1.1, y:2.1
```

at callee

```
a:1.1, b:2.1
```

```
double average(double a, double b)
{
    double ret = a/b;
    return ret;
}
```

The term ***locally*** means ***inside of a block.***

CALL-BY-VALUE

Call-by-Value Parameters

- Copy of actual argument passed
  - Considered "local variable" inside function
  - If modified, only "local copy" changes
  - Function has no access to "actual argument" from caller

- This is the default method in C/C++.
  - However, we often want to access the source of the variable.
    - This is called call-by-reference.
  - In C, it was possible using pointers, but C++, we can also use references.

Example

- Display 4.1 (in Textbook)
  - Formal Parameter Used as a Local Variable (1 of 3)

```
// Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00; // Dollars per quarter hour.

double fee( int hoursWorked, int minutesWorked );
// Returns the charges for hoursWorked hours and
// minutesWorked minutes of legal services.

int main( )
{
   int hours, minutes;
   double bill;
```

Example

- Display 4.1 (in Textbook)
  - Formal Parameter Used as a Local Variable (2 of 3)

```
    cout << "Welcome to the law office of\n"
     << "Dewey, Cheatham, and Howe.\n"
     << "The law office with a heart.\n"
     << "Enter the hours and minutes"
     << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}
```

The value of minutes is not changed by the call to fee.

Example

- Display 4.1 (in Textbook)
  - Formal Parameter Used as a Local Variable (3 of 3)

```
double fee( int hoursWorked, int minutesWorked )
{
   int quarterHours;

   minutesWorked = hoursWorked*60 + minutesWorked;
   quarterHours = minutesWorked/15;
   return (quarterHours*RATE);
}
```

```
Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00
```

Call-by-Value Pitfalls

- Common Mistake:
  - Declaring parameter "again" inside function:

```
double fee( int hoursWorked, int minutesWorked )
{
   int quarterHours;      // local variable
   int minutesWorked;     // NO!
}
```

  - Compiler error results
    - "Redefinition error…"
- Value arguments ARE like "local variables"
  - But function gets them "automatically"

■ cout.setf

DEMO

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

int main( )
{
  int integer = 511;
  double floating1 = 511;
  double floating2 = 11111.0/3.0;

  cout.setf(ios::showpoint);
  cout.setf(ios::fixed);
  cout.precision(3);
  cout << integer << endl;
  cout << floating1 << endl;
  cout << floating2 << endl;

  return 0;
}
```

CALL-BY-REFERENCE IN C++ FUNCTIONS

Call-By-Reference Parameters

- Used to provide access to caller’s actual argument
  - Caller’s data can be modified by called function!
  - Specified by reference in C++; you can still use pointers.

```
void func_with_ref( double& variable );
void func_with_ptr( double* p_variable );
```

Call-by-reference

DEMO

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

int AddOneRef(int& i) {
  return ++i;
}
int AddOne(int i) {
  return ++i;
}
double HalfRef(double& d){
  d /= 2;
  return d;
}
double Half(double d)
{
  d /= 2;
  return d;
}
```

```
int main()
{
  int i = 1;
  double d = 3.14;

  cout << "before (i): " <<  i << endl;
  cout << AddOne (i) << endl;
  cout << "after Add (i) : " << i << endl;
  cout << AddOneRef (i) << endl;
  cout << "after AddRef (i) : " << i << endl;

  cout << "before (d): " << d << endl;
  cout << Half(d) << endl;
  cout << "after Half(d): " << d << endl;
  cout << HalfRef(d) << endl;
  cout << "after HalfRef(d): " << d << endl;
}
```

Returning Reference

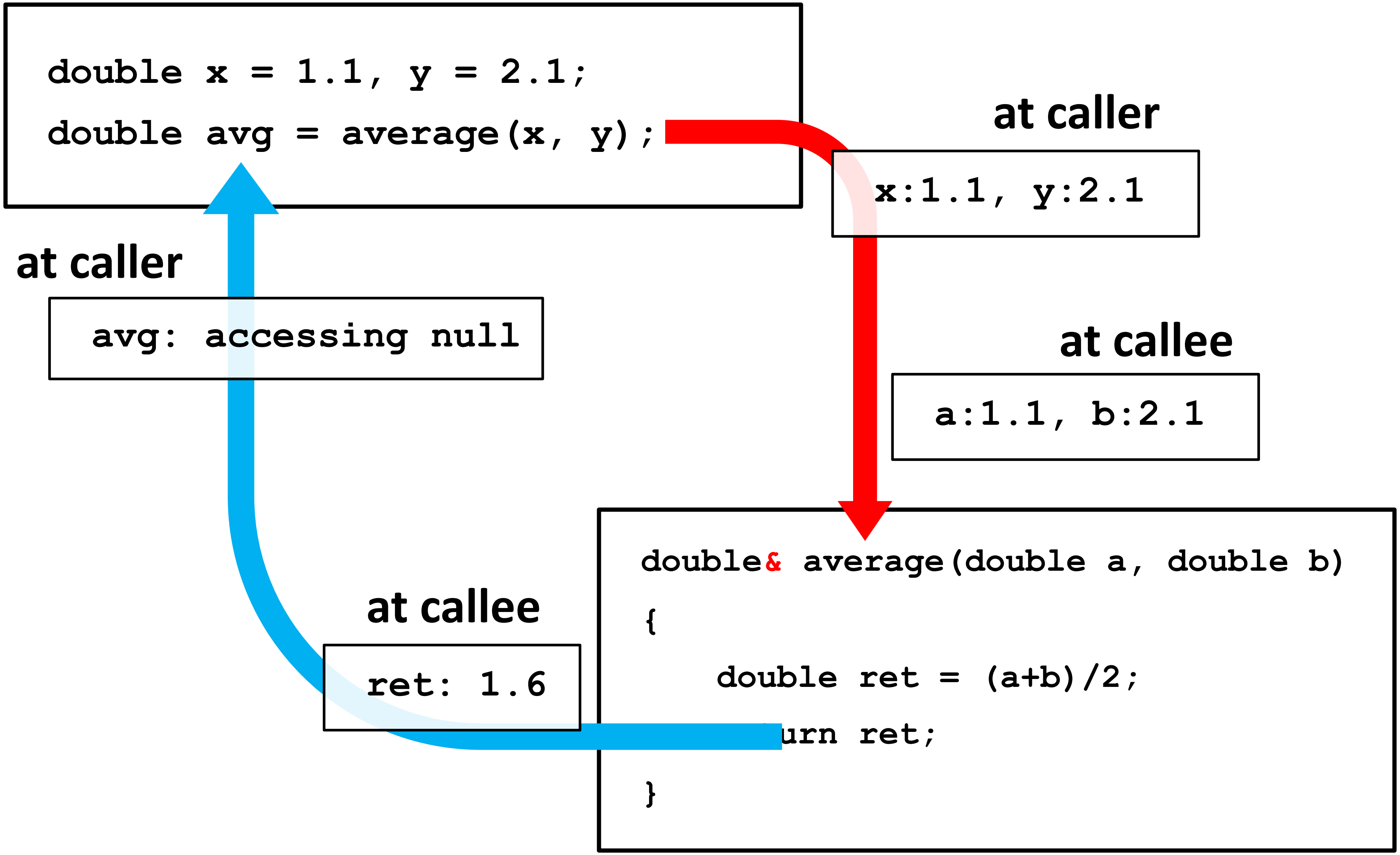
- Returning a reference

```
double& func( double& variable )
{
    return variable;
}
```

  - Think of as returning an "**alias**" to a variable
  - Must match in function declaration and heading

- A returned item must have a reference
  - Like a variable of that type
  - Local variable (allocated in function call stack) cannot be returned
  - Cannot be an expression like "x+5"
    - Has no place in memory to "refer to“

Returning Reference



```
double x = 1.1, y = 2.1;
double avg = average(x, y);
```

at caller

```
x:1.1, y:2.1
```

at callee

```
a:1.1, b:2.1
```

```
double& average(double a, double b)
{
    double ret = (a+b)/2;
    return ret;
}
```

at callee

```
ret: 1.6
```

at caller

```
avg: accessing null
```

Returning reference

DEMO

```
#include <iostream>

using namespace std;

double& func( double& variable )
{
  variable++;
  return variable;
}

double& mal_func( double variable)
{
  return variable;
}
```

```
int main() {
  double dVar = 3.14;

  cout << dVar << endl;
  cout << func(dVar) << endl;
  cout << dVar << endl;

  double& dVar2 = mal_func(dVar);
  cout << "&dVar:" << &dVar << endl;
  cout << "&dVar2:" << &dVar2 << endl;
}
```

Example: Display 4.2 (1/2)

DEMO

```cpp
// Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

// Reads two integers from the keyboard
void getNumbers( int& input1, int& input2 );
// Interchanges the values of variable1 and variable2
void swapValues( int& variable1, int& variable2 );
// Shows the values of output1 and output2, in that order
void showResults( int output1, int output2 );

int main( )
{
    int firstNum, secondNum;
    getNumbers( firstNum, secondNum );
    swapValues( firstNum, secondNum );
    showResults( firstNum, secondNum );
    return 0;
}
```

Example: Display 4.2 (2/2)

```
void getNumbers( int& input1, int& input2 )
{
    cout << "Enter two integers: ";
    cin >> input1 >> input2;
}
void swapValues( int& variable1, int& variable2 )
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
void showResults( int output1, int output2 )
{
    cout << "In reverse order the numbers are: "
     << output1 << " " << output2 << endl;
}
```

```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```

PARAMETERS AND ARGUMENTS

Parameters and Arguments

- Confusing terms, often used interchangeably
- True meanings:
  - Formal parameters
    - In function declaration and function definition
  - Arguments
    - Used to "fill-in" a formal parameter
    - In function call (argument list)

Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller’s data can be changed
  - Often this is desired, sometimes not

- To "protect" data, & still pass by reference:
  - Use **const** modifier

```
void send_const_ref( const int& par1, const int& par2);
```

  - Makes arguments "read-only" by function
  - No changes allowed inside function body.
  - This is a very common practice in C++ functions.

- Why and when do we use constant reference parameters?

Mixed Parameter Lists

- Can combine passing mechanisms
  - Parameter lists can include pass-by-value and pass-by-reference parameters

```
void mixed_call( int& par1, int par2, const double& par3 );
```

  - Function call:

```
mixed_call( arg1, arg2, arg3 );
```

    - arg1 must be an integer type, is passed by reference
    - arg2 must be an integer type, is passed by value
    - arg3 must be a double type, is passed by const reference

Choosing Formal Parameter Names

- Same rule as naming any identifier:
  - Meaningful names!

- Functions as "self-contained modules"
  - Designed separately from the rest of program
  - All must "understand" proper function use
  - OK if formal parameter names are same as argument names
    - See next example

- Choose function names with same rules

Parameter names
same as argument names

```
// function declaration
void mixed_call( int& par1, int par2, const double& par3 );

int main()
{
    // Variables with different scopes
    int par1 = 1, par2 = 2, par3 = 3;

    // calling with variables
    mixed_call( par1, par2, par3 );
}
```

FUNCTION OVERLOADING

Problems in C Functions

- C allows only a single function for a unique function name
  - This leads many different definitions for the same functionality but only with changes in function signatures.

- Typical workaround
  - append postfixes to indicate the function signature change.
    - Example: two average functions for `double` and `int`

```c
double averaged( double n1, double n2 )
{
    return ((n1 + n2) / 2.0);
}

int averagei( int n1, int n2 )
{
    return ((n1 + n2) / 2);
}
```

Function Overloading in C++

- Same function name, but different function signature
  - C++ significantly relaxes the constraint in C, by allowing multiple signatures for the same function name.
  - Allows same task performed on different data

- Function "signature"
  - Function name & parameter list
  - Must be "unique" for each function definition
  - **Return type is not included in the function signature**.

Overloading Example: Average

- Two functions of the same name (same task)
  - Function computes average of 2 numbers:

```
double average( double n1, double n2 ){
   return ((n1 + n2) / 2.0);
}
```

  - compute average of 3 numbers:

```
double average( double n1, double n2, double n3 ){
   return ((n1 + n2 + n3) / 3.0);
}
```

- Usage
  - You can call either of average(5.2, 6.7) or average(6.5, 8.5, 4.2)
  - Compiler resolves invocation based on signature of function call.
  - When there is an ambiguity, compiler says errors.

Overloading Resolution with Type Conversion

```
double mpg( double miles, double gallons )
{
    return (miles/gallons);
}
```

- 1st: looks for the exact match of signature
  - where no argument conversion required
  - e.g., `mpg_computed = mpg(5.8, 20.2);`
- 2nd: looks for a compatible match
  - where automatic type conversion is possible:
  - e.g., `mpg_computed = mpg(5, 20);`
    - Converts 5 & 20 to doubles, then passes
  - e.g., `mpg_computed = mpg(5, 2.4);`
    - Converts 5 to 5.0, then passes values to function

Overloading resolution

DEMO

```
#include <iostream>

using namespace std;

double mpg( double miles, double gallons ){
  cout << "dd" << endl;
  return (miles/gallons);
}

double mpg( int miles, int gallons ){
  cout << "ii" << endl;
  return (double(miles)/gallons);
}

double mpg( int miles, double gallons ){
  cout << "id" << endl;
  return (miles/gallons);
}
```

```
int main()
{
  cout << mpg(5, 20) << endl;
  cout << mpg(5, 24.9) << endl;
}
```

DEFAULT ARGUMENTS

Default Arguments

- Allows omitting some arguments
  - Very convenient in practice
  - Specified in function declaration/prototype

```
// last 2 arguments are defaulted
void show_volume( int length, int width=1, int height=1 );

// possible calls
show_volume(2, 4, 6);  // all arguments supplied
show_volume(3, 5);     // height defaulted to 1
show_volume(7);        // width & height defaulted to 1
```

- Usage
  - should start from the end of parameter list, and be consecutively defined.

```
// erroneous definition: compilation error!
void show_volume( int length, int width=1, int height );
```

Default Arguments

DEMO

```
#include <iostream>

using namespace std;

// The code has errors
void show_volume( int length, int width=1, int height );

int main()
{
  show_volume(2, 4, 6); // all arguments supplied
  show_volume(3, 5);    // height defaulted to 1
  show_volume(7);     // width & height defaulted to 1
}

void show_volume( int length, int width, int height )
{
  cout << length * width * height << endl;
}
```

INLINE FUNCTIONS

Inline Functions

- Compiler attempts to insert the code in place of call
  - Eliminates overhead of function call
  - More efficient, but not always guaranteed to be inserted.

- Usage:
  - Use keyword inline in function declaration and function heading
  - Use for short functions in general

```
inline int add( int a, int b ){ return a+b; }
```