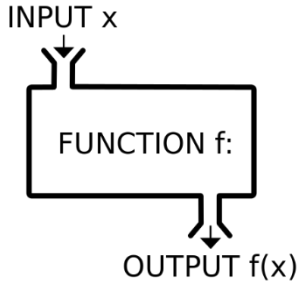
FUNCTION BASICS

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)

Demo Code

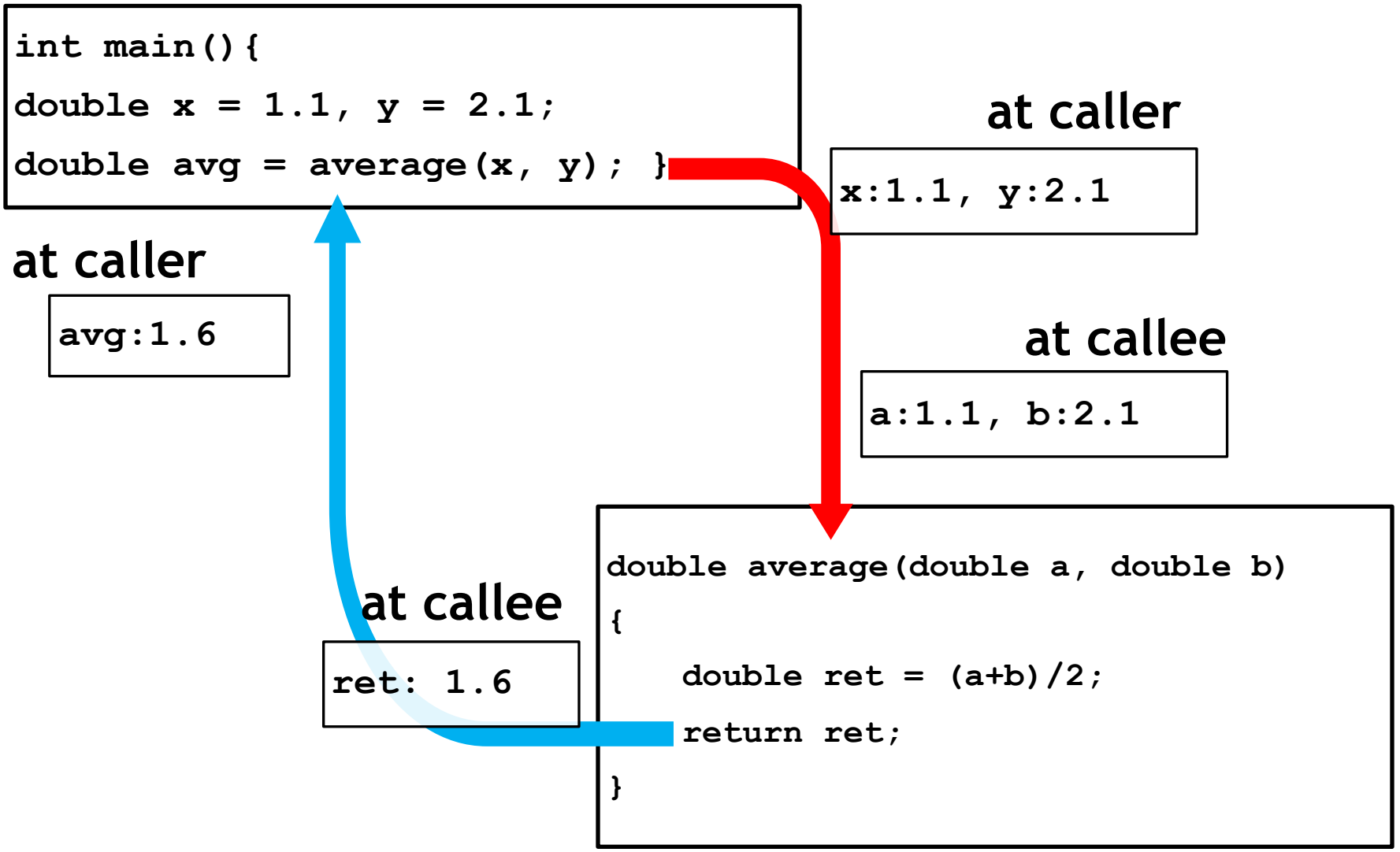
```
1  #include <iostream>
2  using namespace std;
3  void range_summation (int min, int max);

4  void range_summation(int min, int max)
5  {
6     int sum = 0;
7     for (int i = min; i <= max; i++)
8        sum += i;
9
10    cout << "Sum is " << sum << endl;
11 }
12
13 int main()
14 {
15    int userInput;
16
17    while (1) {
18      cin >> userInput;
19
20      switch(userInput){
21        case 1:
22          range_summation(0,10);
23          break;
24
25        case 2:
26          range_summation(20,30);
27          break;
28
29        case 3:
30          range_summation(40,50);
31          break;

            …
```

Function Declaration

Function Definition

Function Call

Parameter Passing Example



Scope for Parameters

```
int main (){
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
```

The term ***locally*** usually and roughly means ***inside of a block.***

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

Summary

- **Function Basics**
  - Function Declaration, Definitions, Calls
- **Parameter Passing Example**
- **Scope of Parameters**

PARAMETERS IN FUNCTION CALL - CALL-BY-VALUE

Call-by-Value Parameters

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

- **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"

PARAMETERS IN FUNCTION CALL - CALL-BY-REFERENCE

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 );
```

Function arguments must be variables

- No constant or other expression

Call-by-reference


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

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

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

  cout << "before: " <<  i << endl;
  //cout << AddOne(i) << endl;
  cout << AddOneRef(i) << endl;
  cout << "after : " << i << endl;
#if 0
  cout << "before: " << d << endl;
  cout << Half(d) << endl;
  cout << "after : " << d << endl;
#endif
}
```

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

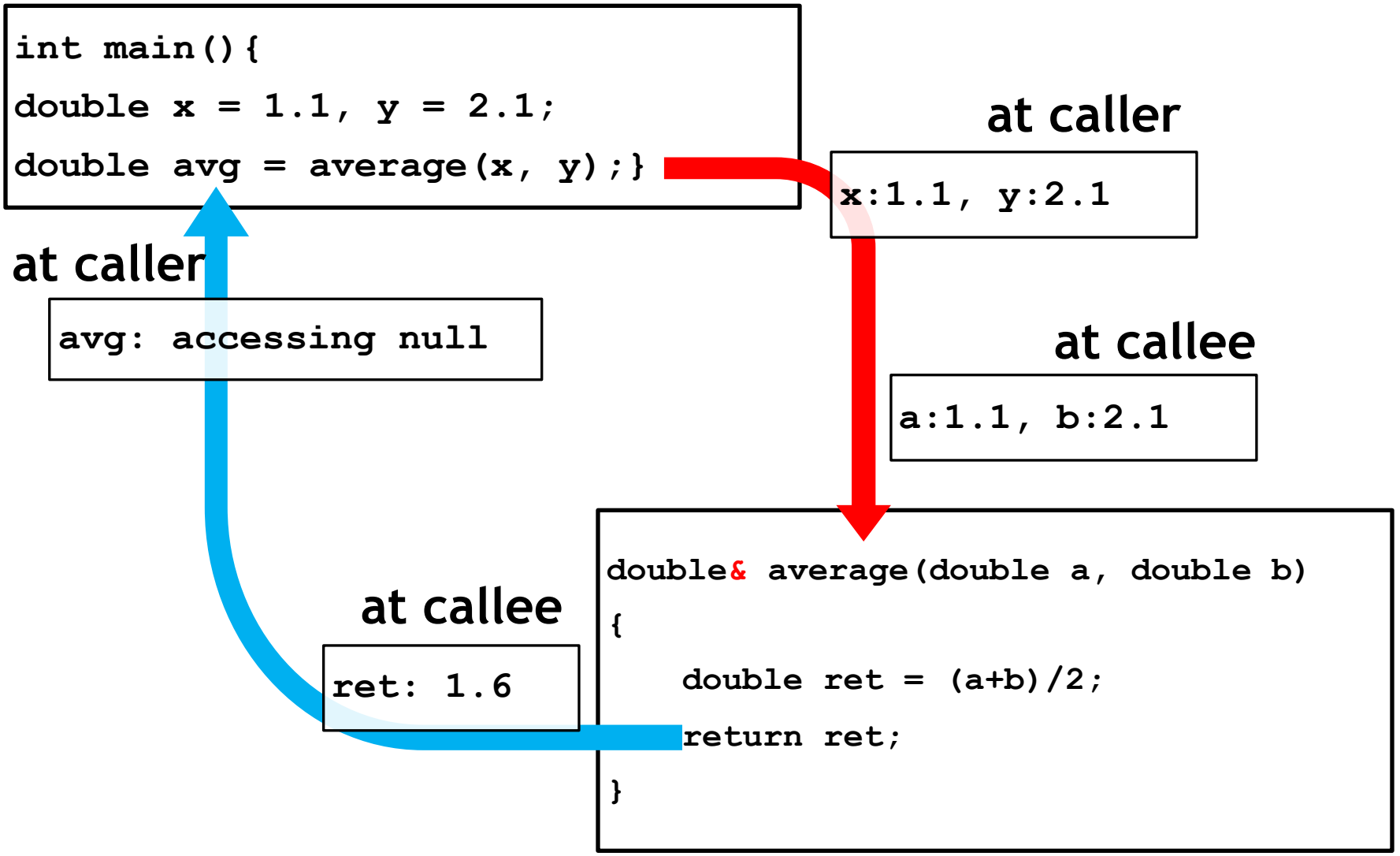
Function on the left side of an assignment

```
func(a) = 2;
func(b) = 3;
```

Returning Reference

- **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



Returning reference


```
#include <iostream>

using namespace std;

double& func( double& 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 = func(dVar);
  // See the difference
  //double& dVar2 = func(dVar);
  cout << "&dVar:" << &dVar <<
endl;
  cout << "&dVar2:" << &dVar2 <<
endl;
}
```

Example: Display 4.2 (1/2)


```
// 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
```

Summary

- **Call-By Value**
  - Value of argument is copied
- **Call-By Reference**
  - Memory of argument is passed
- **Returning Reference**
  - Should not return local variables

PARAMETERS AND ARGUMENTS

Parameters and Arguments

- **Confusing terms, often used interchangeably**
- **True meanings:**
  - Parameters or Formal parameters
    - In function declaration and function definition
  - Arguments
    - Used to "fill-in" a formal parameter
    - In function call (argument list)
  - Call-by-value & Call-by-reference
    - Simply the "mechanism" used in plug-in process

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

average(var1, var2);
```

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 triangle_area( int height, int base);
void func1( int a, int b);

int main()
{
    // Variables with different scopes
    int height = 2, base = 5;

    // calling with variables
    triangle_area( height, base);
}
```

Summary

- Argument vs parameter
- Const reference parameter
- Mixed parameter lists
- Parameter naming

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`

```
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**.

```
int avg(int a, int b);
int avg(int x, int s);
double avg(int a, int b);
int avg(int a, double b);
int avg(int a, int b, int c);
```

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


```
#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;
}
```

Ambiguity

- Avoid confusing overloading

```
#1 void f(int n, double m);
#2 void f(double n, int m);
#3 void f(int n, int m);

f(98, 99);   -> Calls #3
f(5.3, 4);   -> Calls #2
f(4.3, 5.2); -> Calls ?
```

Summary

- Function Overloading in C++
- Overloading Resolution Rules
- Ambiguity in Overloading

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


```
#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; }
```

Inline Functions

Another usage of inline functions

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

- You can define a function in a header, not causing error when included multiple times.
- Common in recent header-only style libraries
  - You do not have to define functions in an additional *.cpp.

Example: yourmath.h

```
#pragma once
#ifndef __YOURMATH_H__
#define __YOURMATH_H__

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

#endif
```

Summary

- Default Argument
- Inline Function
- #ifndef, #define, #endif