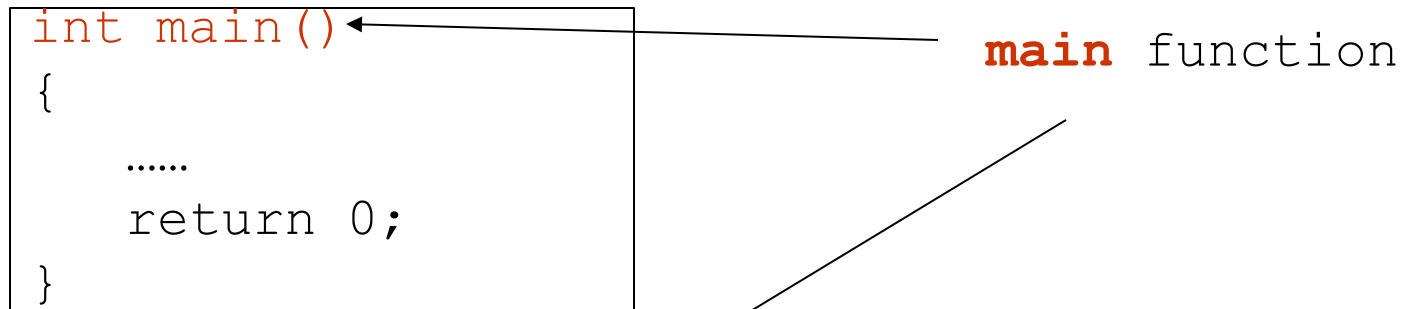# Function

Outline

- Library functions
- Function declaration and definition
- Parameters Passing
- Function Return
- Function Overloading
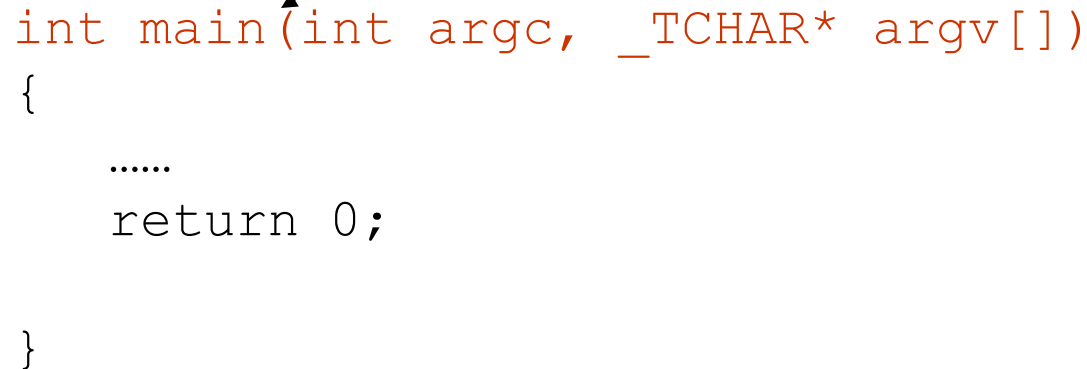- Local and Global Variable

Remember These？？？

```
int main()
{
    ……
    return 0;
}
```
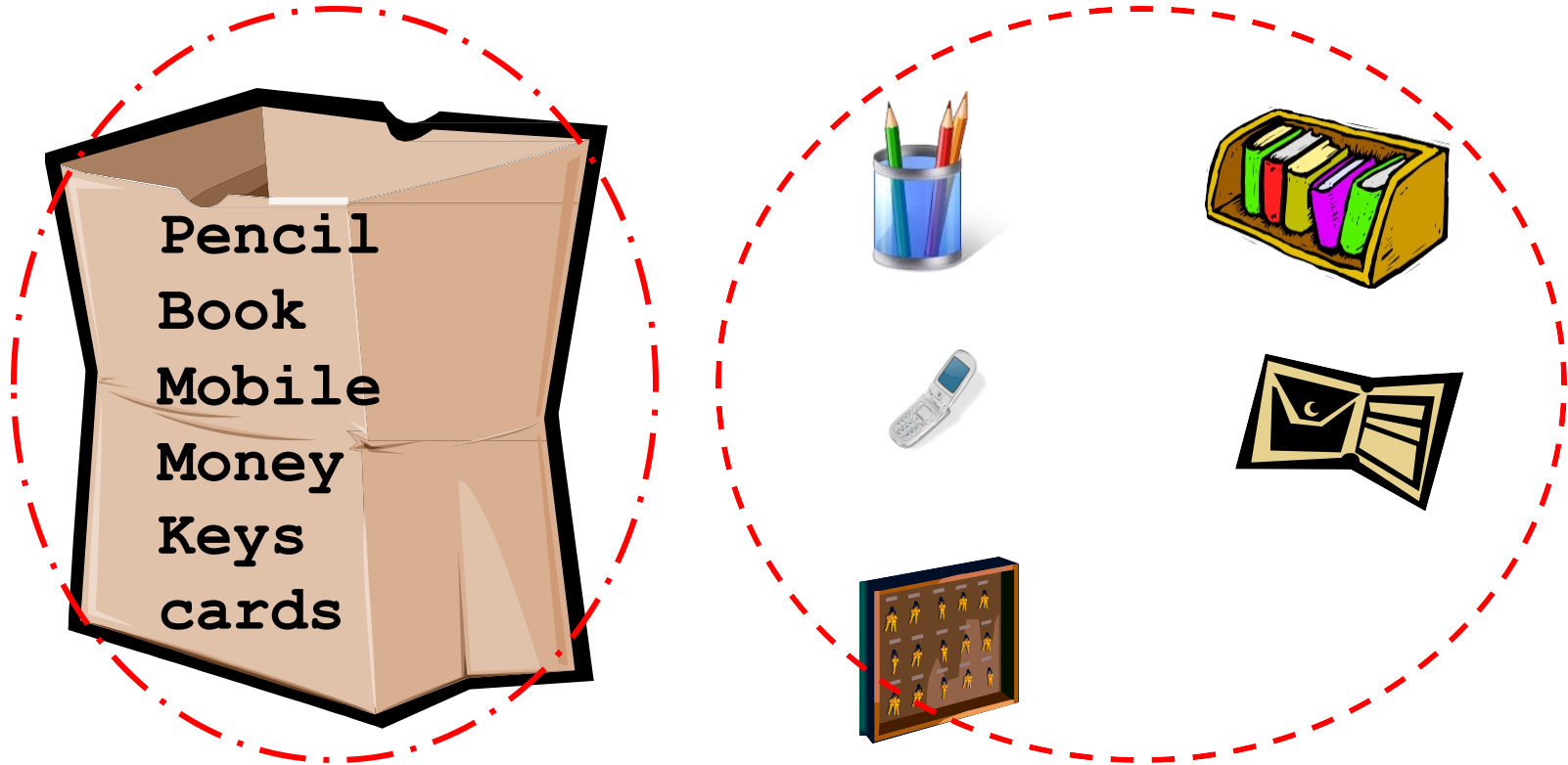
main function

```
int main(int argc, _TCHAR* argv[])
{
    ……
    return 0;

}
```

Function

Pencil
Book
Mobile
Money
Keys
cards

Which is better?
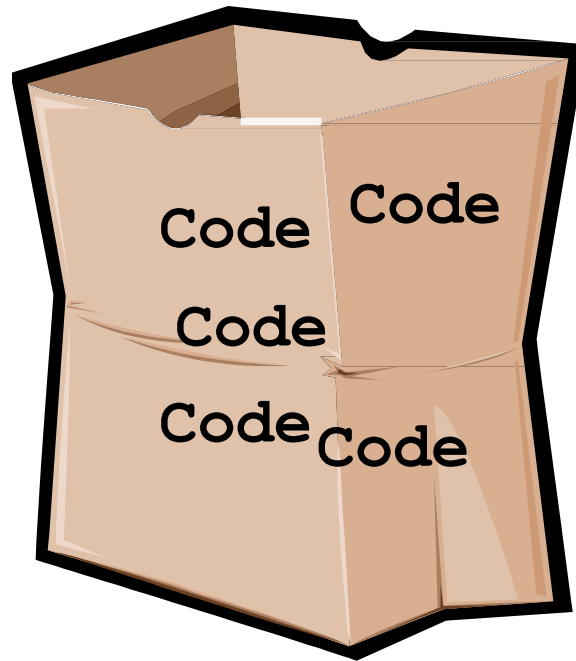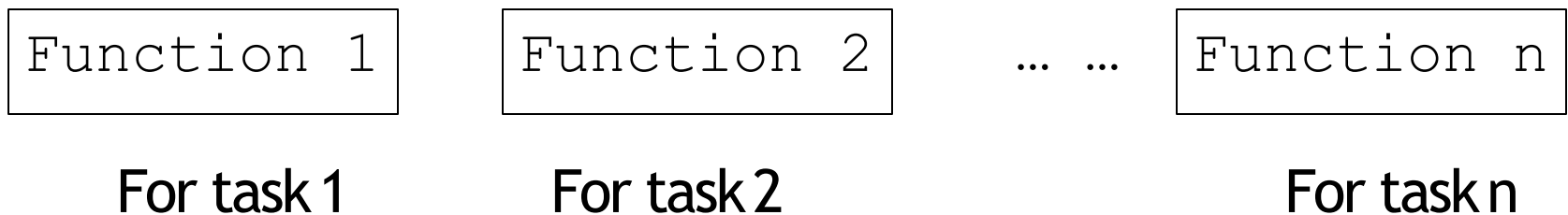
Function

Can you imagine a function with over ten thousands of lines

Functions

- A complex problem is often easier to solve by dividing it into several smaller sub-problems (tasks)

| Function 1 | Function 2 | … … | Function n |

For task 1   For task 2   For task n

Function

```
int main()
{
  cout << "Hello World";
  int a=1, b=2;
  cout << a+b;
  return 0;
}
```

Function

- Two kinds of functions
  - Library functions
    - Come with standard library
    - Programmer can use them directly
  - Defined functions
    - Written by programmers
- Any program function has both
  - function declaration (prototype)
    - How to use (call) this function
  - function definition
    - How this function is implemented

Functions

- Function declaration (prototype)
    - How to use (call) this function
- Function definition
    - How this function is implemented

Library functions
- Give the declaration
- Hide the definition

Defined functions
- Give both declaration and the definition explicitly.

Function Declaration

A function is a block of program code which deals with a particular task.

Function declaration syntax:

```
<type> <function name>(<type list>);
```

Input variables

Function
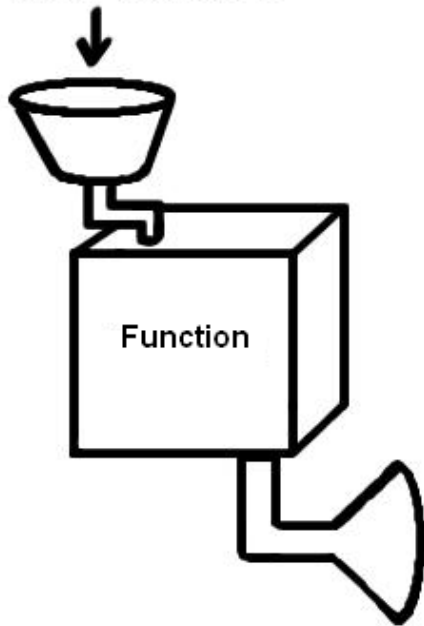
Return value

Type of
return value
(can be "void" if no
value is returned)

List of types of input variables
(can be empty if there is no input)

A function declaration is also called a function prototype.

An Example of Function Prototype

- `int print(char c, int i);`

  - Function name?
  - How many input variables?
  - What are the types of input variables?
  - What is the type of the return value (output value)

Function Name

- A function name must be meaningful
  - If we want to write a function to calculate the average of grades, which of the following function names is better?
    - averageGrade
    - Abc
    - aG
    - avaragegrade
    - ag

Function Definition

```
type functionName (formal_parameter_list){
      local_declarations
      sequence_of_statements
}
```

Examples of Function Definition

```cpp
int sum(int oprand1, int oprand2)
{
  int s;
  s = operand1 + operand2;
  return s;
}
```

```cpp
int absolute(int x){
  if (x >= 0)
    return x;
  else
    return -x;
}
```

```cpp
void printASCIICode(char code)
{
  cout << code << "\'s ASCII code is " << (int)code << endl;
  return;
}
```

Function Call

- To use a function, we can call this function by following its format.

- Syntax

```
function_name(<actual_parameter_list>)
```
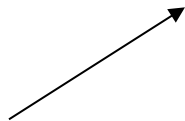
The name of the function that has been defined

The list of the input values in the same sequence as defined
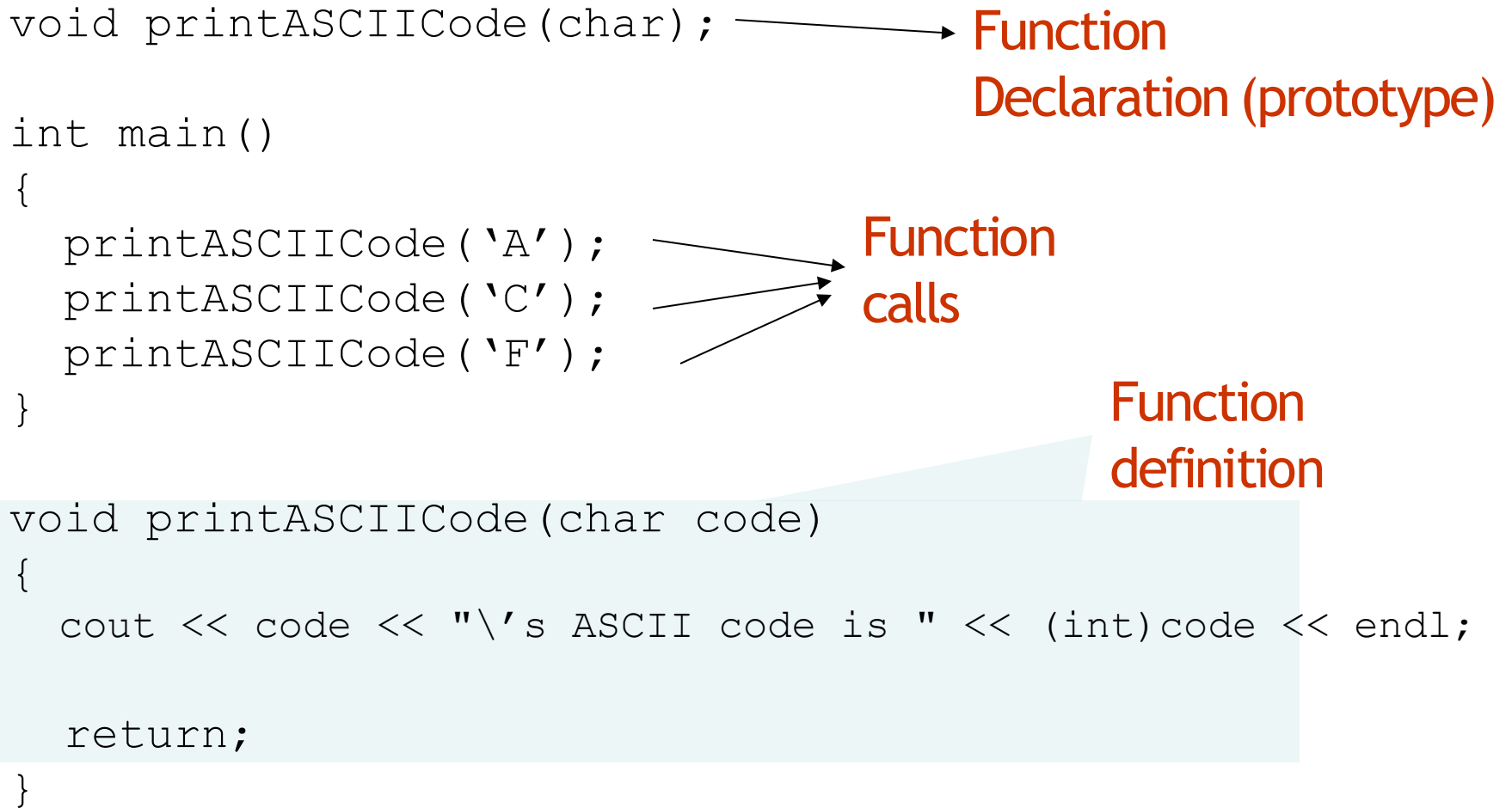
An Example of Function Call

```
void printASCIICode(char);          ──────►  Function
                                             Declaration (prototype)

int main()
{
  printASCIICode('A');     ──────►  Function
  printASCIICode('C');     ──────►  calls
  printASCIICode('F');     ──────►
}
                                    Function
                                    definition
void printASCIICode(char code)
{
  cout << code << "\'s ASCII code is " << (int)code << endl;

  return;
}
```
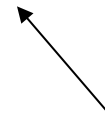
Three Steps
in Constructing and Using aFunction

1. First, declare a function using a function prototype (p1, p2, ..., pn) can be omitted sometimes)

   ```
   type function_name(type1 p1, type2 p2,…, typen pn);
   ```
2. Second, define the function

   ```
   type function_name(type1 p1, type2 p2,…, typen pn)
   {
      ………….
      return value;   //or return;
   }
   ```
3. Third, call the function

   ```
   variable = function_name(v1, v2,…, vn);
   // or function_name(v1, v2,…, vn)  if there is no
   ```
   return value;

Another Example of FunctionCall

```cpp
#include <iostream>
using namespace std;
int absolute(int);


int main()
{
    int value, answer;
    cin >> value;
    answer = absolute(value);
    cout << "The absolute value is " << answer << endl;
      return 0;
}


int absolute(int x){
    if (x >= 0)
      return x;
    else
      return -x;
```

Function declaration

Function calls

Function definition

18

Library Functions

- The standard C++ library contains a large collections of functions that can be called from a C++ program.
- The declarations of these functions are included in `.h` files.
  - Iostream
  - string
  - cstdlib
  - fstream
  - ……

Library Functions

- Three steps
  - find the header file which includes the library function to call
  - Use include directive `#include` to encompass the header file
  - Call the function in the program

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
    double x = sqrt(2.0);
    cout << "x = " << setw(10) << x << endl;
    return 0;
```

Structure in A Function

- A structure parameter value can be passed to a function
- A structure can be returned

```
date nextDay(date today)
{
    date tomorrow;
    // code to calculate the date of tomorrow
        ......
    return tomorrow;
}
```

(Assume date is a structure type)

Actual and Formal Parameters

```cpp
#include <iostream>
using namespace std;
int sum(int, int);
int main()
{
    int o1, o2;
    cin >> o1 >> o2;
    cout <<"The sum is " << sum(o1, o2) << endl;
    return 0;
}
int sum(int oprand1, int oprand2)
{
    return (operand1 + operand2);
}
```

Actual parameters

Formal parameters

Function Call – Value Passing
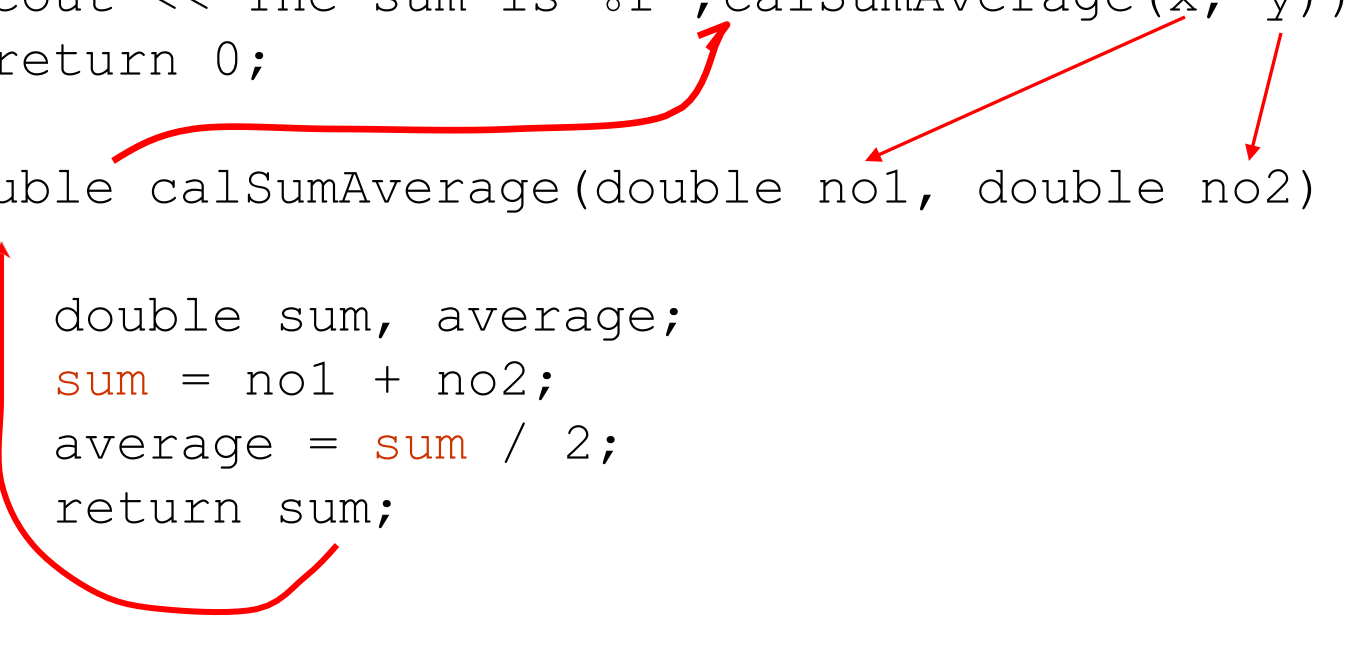
- A function call can return only one single result

```cpp
#include <iostream>
using namespace std;
int sum(int, int);
int main()
{
    int o1, o2;
    cin >> o1 >> o2;
    cout <<"The sum is " << sum(o1, o2);
    cout << endl;
    cout <<"2 + 3 = " << sum(2, 3) << endl;
    return 0;
}
int sum(int oprand1, int oprand2)
{
    return (operand1 + operand2);
}
```

How about if we want multiple values to be returned to the main function?

Value Passing

```
double calSumAverage(double, double );
int main( )
{
   double x = 1.0, y = 2.0;
   cout <<"The sum is %f",calSumAverage(x, y));
   return 0;
}
double calSumAverage(double no1, double no2)
{
    double sum, average;
    sum = no1 + no2;
    average = sum / 2;
    return sum;
}
```

We can return either sum or average, but not both

Passing Data to Function

- Pass Data by Value
- Pass Data by Reference
- Pass Data by Address

Pass by Value

Passing Data by Value

- Pass by value: when argument is passed to a function, a copy of its value is placed in the parameter

- Function cannot access the original argument

- Changes to the parameter in the function do not affect the value of the argument in the calling function

Passing Data to Parameters by Value

- Example: `int val = 5;`
                      `evenOrOdd(val);`

val

5

**argument in
calling function**

num

5

**parameter in
`evenOrOdd` function**

- **`evenOrOdd`** can change variable **`num`**, but it will have no effect on variable **`val`**

Pass by Reference

Passing Data by Reference

- While pass by value is suitable in many cases, it has a couple of limitations.
    - when passing a large array or struct to a function, pass by value will make a copy of the argument into the function parameter, which may cost **huge memory**.
    - Call by value only returns single value, while in some cases we may want to **modify the argument(s)** passed in.

- To pass a variable by reference, we simply declare the function parameters as references rather than as normal variables:

```
1   void addOne(int &y) // y is a reference variable
2   {
3       y = y + 1;
4   }
```

- When the function is called, y will become a reference to the argument. Any changes made to the reference are passed through to the argument!

Passing Data by Reference

```
1   void addOne(int &y) // y is a reference variable
2   {
3       y = y + 1;
4   } // y is destroyed here
5
6   int main()
7   {
8       int x = 5;
9       std::cout << "x = " << x << '\n';
10      addOne(x);
11      std::cout << "x = " << x << '\n';
12      return 0;
13  }
```

This produces the output:

```
x = 5
x = 6
```

- the value of argument x was changed by the function.

Returning multiple values via out parameters

- Sometimes we need a function to return multiple values. However, functions can only have one return value. One way to return multiple values is using reference parameters:

```cpp
1   #include <iostream>
2   #include <cmath>      // for std::sin() and std::cos()
3
4   void getSinCos(double degrees, double &sinOut, double &cosOut)
5   {
6       // sin() and cos() take radians, not degrees, so we need to convert
7       const double pi = 3.14159265358979323846; // the value of pi
8       double radians = degrees * pi / 180.0;
9       sinOut = std::sin(radians);
10      cosOut = std::cos(radians);
11  }
12
13  int main()
14  {
15      double sin(0.0);
16      double cos(0.0);
17
18      // getSinCos will return the sin and cos in variables sin and cos
19      getSinCos(30.0, sin, cos);
20
21      std::cout << "The sin is " << sin << '\n';
22      std::cout << "The cos is " << cos << '\n';
23      return 0;
24  }
```

Limitations of pass by reference

- Non-const references can only reference non-const l-values (e.g. non-const variables), so a reference parameter cannot accept an argument that is a **const l-value** or an **r-value** (e.g. literals and the results of expressions).

```cpp
 1  #include <iostream>
 2  #include <cmath>     // for std::sin() and std::cos()
 3
 4  void getSinCos(double degrees, double &sinOut, double &cosOut)
 5  {
 6      // sin() and cos() take radians, not degrees, so we need to convert
 7      const double pi = 3.14159265358979323846; // the value of pi
 8      double radians = degrees * pi / 180.0;
 9      sinOut = std::sin(radians);
10      cosOut = std::cos(radians);
11  }
12
13  int main()
14  {
15      double sin(0.0);
16      double cos(0.0);
17
18      // getSinCos will return the sin and cos in variables sin and cos
19      getSinCos(30.0, 0.0, cos);
20
21      std::cout << "The sin is " << sin << '\n';
22      std::cout << "The cos is " << cos << '\n';
23      return 0;
24  }
```

cannot be rvalue

| options | compilation | execution |

```
In function 'int main()':
19:29: error: invalid initialization of non-const reference of type 'double&' from an rvalue of type 'double'
4:6: note: in passing argument 2 of 'void getSinCos(double, double&, double&)'
```

33

Pass by const reference

- Although "pass by reference" has its advantages, this also opens us up to potential trouble. References allow the function to change the value of the argument, which is undesirable when we want an argument be read-only.

- If we know that a function **should not change** the value of an argument, but don't want to pass by value, the best solution is to pass by const reference.

```
void foo(const int &x) // x is a const reference
{
    x = 6;  // compile error: a const reference cannot have its value ch
anged!
}
```

Pass by const reference

- Using const is useful for several reasons:
  - It enlists the compilers help in ensuring values that shouldn't be changed aren't changed (the compiler will throw an error if you try, like in the above example).
  - It tells the programmer that the function won't change the value of the argument. This can help with debugging.
  - You can't pass a const argument to a non-const reference parameter. Using const parameters ensures you can pass both non-const and const arguments to the function.
  - Const references can accept any type of argument, including l-values, const l-values, and r-values.

- *Rule: When passing an argument by reference, always use a const references unless you need to change the value of the argument*

When to use pass by reference:

- When to **use** pass by reference:
    - When passing structs or classes (use const if read-only).
    - When you need the function to modify an argument.
    - When you need access to the type information of a fixed array.

- When **not to use** pass by reference:
    - When passing fundamental types that don't need to be modified (use pass by value).

Pass by Pointer/Address

Passing arguments by address

- **Passing an argument by address** involves passing the address of the argument variable rather than the argument variable itself. he function can then dereference the pointer to access or change the value being pointed to.

```cpp
#include <iostream>

void foo(int *ptr)
{
    *ptr = 6;
}

int main()
{
    int value = 5;

    std::cout << "value = " << value << '\n';
    foo(&value);
    std::cout << "value = " << value << '\n';
    return 0;
}
```

The above snippet prints:

```
value = 5
value = 6
```

Program 9-11

```
 1   // This program uses two functions that accept addresses of
 2   // variables as arguments.
 3   #include <iostream>
 4   using namespace std;
 5
 6   // Function prototypes
 7   void getNumber(int *);
 8   void doubleValue(int *);
 9
10   int main()
11   {
12      int number;
13
14      // Call getNumber and pass the address of number.
15      getNumber(&number);
16
17      // Call doubleValue and pass the address of number.
18      doubleValue(&number);
19
20      // Display the value in number.
21      cout << "That value doubled is " << number << endl;
22      return 0;
23   }
24
```

When the **getnumber** function is called in line 15, the address of the **number** variable is passed as the argument. After the function executes, the value entered by the user is stored in **number**.

(Program Continues)

```
25   //************************************************************
26   // Definition of getNumber. The parameter, input, is a pointer. *
27   // This function asks the user for a number. The value entered  *
28   // is stored in the variable pointed to by input.               *
29   //************************************************************
30
31   void getNumber(int *input)
32   {
33      cout << "Enter an integer number: ";
34      cin >> *input;
35   }
36
37   //************************************************************
38   // Definition of doubleValue. The parameter, val, is a pointer. *
39   // This function multiplies the variable pointed to by val by    *
40   // two.                                                          *
41   //************************************************************
42
43   void doubleValue(int *val)
44   {
45      *val *= 2;
46   }
```

cin>>input would store the value entered by the user in **input** as if the value were an address . Thus, **input** would no longer point to the **number** variable in function **main**

Program Output with Example Input Shown in Bold
```
Enter an integer number: 10 [Enter]
That value doubled is 20
```

Example

```cpp
#include <iostream>
using namespace std;
void swap(int*, int*);

int main()
{
 int x=5, y=10;
 cout<<"Before swap: x "<<x<<" y "<<y<<endl;
 swap(&x,&y);
 cout<<"After swap: x "<<x<<" y "<<y<<endl;
 return 0;
}


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

```
OUTPUT:

Before swap: x 5 y 10
After swap: x 10 y 5
```

Pointers Variables to Arrays

- To pass an array **sales** to a function using a pointer, include the following

```
FUNCTION PROTOTYPE

void getSales(double *, int);

double totalSales(double *, int);


FUNCTION CALL IN MAIN

getSales(sales,4);

double totsale=totalSales(sales,4);
```

Pointers Variables to Arrays

```
FUNCTIONS
void getSales(double* array, int size)
{
  for (int count=0; count<size; count++)
  { cout<<"Enter the sales figure for each quarter";
    cout<<(count+1)<<": ";
    cin>>array[count];
  }
}


double totalSales(double* array, int size)
{ double sum=0.0;
  for (int count=0; count<size; count++)
  { sum += *array;
    array++;
  }
  return sum;
}
```

Passing arguments by address

- Pass by address is typically used with pointers, which most often are used to point to built-in arrays.

```cpp
void printArray(int *array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        return;

    for (int index=0; index < length; ++index)
        cout << array[index] << ' ';
}

int main()
{
    int array[6] = { 6, 5, 4, 3, 2, 1 };
    printArray(array, 6);
}
```

Passing by const address

- Because printArray() doesn't modify any of its arguments, it's good form to make the array parameter const:

```cpp
void printArray(const int *array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        return;

    for (int index=0; index < length; ++index)
        std::cout << array[index] << ' ';
}

int main()
{
    int array[6] = { 6, 5, 4, 3, 2, 1 };
    printArray(array, 6);
}
```

Addresses are actually passed by value

- When you pass a pointer to a function by address, the pointer's value (the address it points to) is **copied** from the argument to the function's parameter.

```cpp
#include <iostream>

void setToNull(int *tempPtr)
{
    // we're making tempPtr point at something else, not changing the value that tempPtr points to.
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main()
{
    // First we set ptr to the address of five, which means *ptr = 5
    int five = 5;
    int *ptr = &five;

    // This will print 5
    std::cout << *ptr;

    // tempPtr will receive a copy of ptr
    setToNull(ptr);

    // ptr is still set to the address of five!

    // This will print 5
    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

Addresses are actually passed by value

- When passing an argument by address, the function parameter variable receives a copy of the address from the argument. At this point, the function parameter and the argument both point to the same value.

- If the function parameter is then *dereferenced* to change the value being pointed to, that *will* impact the value the argument is pointing to, since both the function parameter and argument are pointing to the same value!

- If the function parameter is *assigned* a different address, that **will not** impact the argument, since the function parameter is a copy, and changing the copy won't impact the original. After changing the function parameter's address, the function parameter and argument will point to different values, so dereferencing the parameter and changing the value will no longer affect the value pointed to by the argument.

Passing addresses by reference

- What if we want to change the address an argument points to from within the function?

- You can simply pass the address by reference.

```cpp
#include <iostream>

// tempPtr is now a reference to a pointer, so any changes made to tempPtr will change the argument as well!
void setToNull(int *&tempPtr)
{
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main()
{
    // First we set ptr to the address of five, which means *ptr = 5
    int five = 5;
    int *ptr = &five;

    // This will print 5
    std::cout << *ptr;

    // tempPtr is set as a reference to ptr
    setToNull(ptr);

    // ptr has now been changed to nullptr!

    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

Result: `5 ptr is null`

There is only pass by value

- Now that you understand the basic differences between passing by reference, address, and value, let's get reductionist for a moment.

- References are typically implemented by the compiler as pointers. This means that behind the scenes, pass by reference is essentially just a pass by address (with access to the reference doing an implicit dereference).

- And just above, we showed that pass by address is actually just passing an address by value!

- Therefore, we can conclude that **C++ really passes everything by value!**

49

Function arguments use cases

- *Call by value* is appropriate for small objects that should not be changed by the function

- *Call by constant reference* is appropriate for large objects that should not be changed by the function

- *Call by reference* is appropriate for all objects that may be changed by the function

- *Call by address* is always used when passing built-in arrays

- *Call by constant address* is used if array elements are not expected to be modified

Arrays as Function Arguments

Arrays as Function Arguments

- You can pass array to a function as a pointer
  ```
  void myFunction(int *param) {

      . . .
  }
  ```

- You can also pass entire array. Array parameters in functions are similar to reference variable (No copy, share array memory location).

  ```
  void myFunction(int param[]) {

                  . . .
  }
  ```

- Changes made to array in a function are made to the actual array in calling function.

Arrays as Function Arguments

Pass array as reference

```cpp
1   #include <iostream>
2   using namespace std;
3
4   // function declaration:
5   double getAverage(int [], int );
6
7 ▾ int main () {
8      // an int array with 5 elements.
9      int balance[5] = {1000, 2, 3, 17, 50};
10     double avg;
11
12     // pass pointer to the array as an argument.
13     avg = getAverage( balance, 5 ) ;
14
15     // output the returned value
16     cout << "Average value is: " << avg << endl;
17
18     return 0;
19  }
20
21 ▾ double getAverage(int arr[], int size) {
22     int i, sum = 0;
23     double avg;
24
25 ▾   for (i = 0; i < size; ++i) {
26        sum += arr[i];
27     }
28     avg = double(sum) / size;
29
30     return avg;
31  }
```

Pass array as pointer

```cpp
1   #include <iostream>
2   using namespace std;
3
4   // function declaration:
5   double getAverage(int* arr, int size);
6
7 ▾ int main () {
8      // an int array with 5 elements.
9      int balance[5] = {1000, 2, 3, 17, 50};
10     double avg;
11
12     // pass pointer to the array as an argument.
13     avg = getAverage( balance, 5 ) ;
14
15     // output the returned value
16     cout << "Average value is: " << avg << endl;
17
18     return 0;
19  }
20
21 ▾ double getAverage(int* arr, int size) {
22     int i, sum = 0;
23     double avg;
24
25 ▾   for (i = 0; i < size; ++i) {
26        sum += *(arr+i);
27     }
28     avg = double(sum) / size;
29
30     return avg;
31  }
```

options	compilation	execution

```
Average value is: 214.4
```

Modifying Arrays in Functions

- Modifications about array within function will affect the array in main function.

```cpp
// Example program
#include <iostream>
#include <string>
using namespace std;

void changeArray(int[] , int );

int main()
{
    const int size = 5;
    int arr[size] = {1,2,3,4,5};
    changeArray(arr, size);
    cout << arr[0];
}

void changeArray(int array[], int size){
    for (int i=0; i<size; i++)
        array[i] = 0;
}
```

Function Call Stack

The stack and the heap

- The memory a program uses is typically divided into a few different areas, called segments:
  - The **code** segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
  - The **bss** segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
  - The **data** segment (also called the initialized data segment), where initialized global and static variables are stored.
  - The **heap**, where dynamically allocated variables are allocated from.
  - The **call stack**, where function parameters, local variables, and other function-related information are stored.

The heap segment

- The heap segment keeps track of memory used for dynamic memory allocation.

- When you use the new operator to allocate memory, this memory is allocated in the application's heap segment.

```
int *ptr = new int; // ptr is assigned 4 bytes in the heap
int *array = new int[10]; // array is assigned 40 bytes in the heap
```

- When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received.

- Remember that deleting a pointer does not delete the variable, it just returns the memory at the associated address back to the operating system.

The call stack

- A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off.

- For example,

```
Stack: empty
Push 1
Stack: 1
Push 2
Stack: 1 2
Push 3
Stack: 1 2 3
Pop
Stack: 1 2
Pop
Stack: 1
```

The call stack in action

- When a function call is encountered, the function is pushed onto the call stack. When the current function ends, that function is popped off the call stack.

- Let's examine in more detail how the call stack works. The sequence of steps that takes place when a function is **called**:

    1. The program encounters a function call.

    2. A stack frame is constructed and pushed on the stack. The stack frame consists of:

    - The address of the instruction beyond the function call (called the **return address**). This is how the CPU remembers where to return to after the called function exits.
    - All function arguments.
    - Memory for any local variables.
    - Saved copies of any registers modified by the function that need to be restored when the function returns

    3. The CPU jumps to the function's start point.

    4. The instructions inside of the function begin executing.

The call stack in action

- When the function **terminates**, the following steps happen:
    1. Registers are restored from the call stack
    2. The stack frame is popped off the stack. This frees the memory for all local variables and arguments.
    3. The return value is handled.
    4. The CPU resumes execution at the return address.

Call stack example

```
1    int foo(int x)
2    {
3        // b
4        return x;
5    } // foo is popped off the call stack here
6
7    int main()
8    {
9        // a
10       foo(5); // foo is pushed on the call stack here
11       // c
12
13       return 0;
14   }
```

The call stack looks like the following at the labeled points:

a:

main()

b:

foo() (including parameter x)
main()

c:

main()

Stack overflow

- The stack has a limited size, and consequently can only hold a limited amount of information. On
  - Windows, 1MB. On some
  - Unix machines, can be as large as 8MB.

- If the program tries to put too much information on the stack, stack overflow will result.

- **Stack overflow** happens when all the memory in the stack has been allocated -- in that case, further allocations begin overflowing into other sections of memory.

- Below example will likely cause a stack overflow

```
int main()
{
    int stack[100000000];
    return 0;
}
```

- Another program that will cause a stack overflow.

```
void foo()
{
    foo();
}

int main()
{
    foo();

    return 0;
}
```

Function Returning values

- returning values from a function to its caller by value, address, or reference works almost exactly the same way as passing parameters to a function does.

- However, there is one more added bit of complexity -- because local variables in a function go out of scope and are destroyed when the function returns, we need to consider the effect of this on each return type.

Return value

- Return by value is the simplest and safest return type to use.

- When a value is returned by value, a copy of that value is returned to the caller.

```
1   int doubleValue(int x)
2   {
3       int value = x * 2;
4       return value; // A copy of value will be returned here
5   } // value goes out of scope here
```

- Return by value is the most appropriate when returning variables that were declared inside the function, or for returning function arguments that were passed by value.

- However, like pass by value, return by value is slow for structs and large classes.

Return value

- When to use return by value:
    - When returning variables that were declared inside the function
    - When returning function arguments that were passed by value

- When not to use return by value:
    - When returning a built-in array or pointer (use return by address)
    - When returning a large struct or class (use return by reference)

Return address

- Returning by address involves returning the address of a **variable** to the caller (not a literal or an expression, which don't have addresses).

- Return by address has one additional downside that return by value doesn't -- if you try to return the address of a variable local to the function, your program will exhibit undefined behavior.

```
1   int* doubleValue(int x)
2   {
3       int value = x * 2;
4       return &value; // return value by address here
5   } // value destroyed here
```

- value is destroyed just after its address is returned to the caller. The end result is that the caller ends up with the address of non-allocated memory (a **dangling pointer**), which will cause problems if used.

Return address

- Return by address is often used to return dynamically allocated memory to the caller:

```
1   int* allocateArray(int size)
2   {
3       return new int[size];
4   }
5
6   int main()
7   {
8       int *array = allocateArray(25);
9
10      // do stuff with array
11
12      delete[] array;
13      return 0;
14  }
```

- This works because dynamically allocated memory does not go out of scope at the end of the block in which it is declared, so that memory will still exist when the address is returned back to the caller.

Return address

- When to use return by address:
  - When returning dynamically allocated memory
  - When returning function arguments that were passed by address

- When not to use return by address:
  - When returning variables that were declared inside the function (use return by value)
  - When returning a large struct or class that was passed by reference (use return by reference)

Return address

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

char *findNull(char*);

int main(){
 char *s = {"Ira Rudowsky"};
 cout<<"String at null is ["<<(findNull(s))<<"]"<<endl;
 return 0;
}
// locate the null terminator in a string and returns a pointer to it
// The char * return type in the function header indicates that the function
// returns a pointer to a character

char *findNull(char *str)
{ int count=0;
  cout<<"count: "<<count<<" ";
  char *ptr = str;
  while (*ptr !='\0')
  { cout<<" *ptr: "<<*ptr<<" ptr: "<<ptr<<" &ptr: "<<&ptr<<endl;
    count++; ptr++;
    cout<<"count: "<<count<<" ";
  }
  cout<<endl;
  return ptr;
}
```

Return address

OUTPUT:

```
count: 0   *ptr: I   ptr: Ira Rudowsky    &ptr: 0x22ff40
count: 1   *ptr: r   ptr: ra Rudowsky     &ptr: 0x22ff40
count: 2   *ptr: a   ptr: a Rudowsky      &ptr: 0x22ff40
count: 3   *ptr:     ptr:  Rudowsky       &ptr: 0x22ff40
count: 4   *ptr: R   ptr: Rudowsky        &ptr: 0x22ff40
count: 5   *ptr: u   ptr: udowsky         &ptr: 0x22ff40
count: 6   *ptr: d   ptr: dowsky          &ptr: 0x22ff40
count: 7   *ptr: o   ptr: owsky           &ptr: 0x22ff40
count: 8   *ptr: w   ptr: wsky            &ptr: 0x22ff40
count: 9   *ptr: s   ptr: sky             &ptr: 0x22ff40
count: 10  *ptr: k   ptr: ky              &ptr: 0x22ff40
count: 11  *ptr: y   ptr: y               &ptr: 0x22ff40
count: 12
String at null is []
```

Return address

```cpp
#include <iostream>
#include <cstdlib>  // For rand and srand
#include <ctime>    // For the time func
using namespace std;


// Function prototype
int *getRandomNumbers(int);


int main()
{

   int *numbers;  // To point to the numbers


   // Get an array of five random numbers.
   numbers = getRandomNumbers(5);


   // Display the numbers.
   for (int count = 0; count < 5; count++)
      cout << numbers[count] << endl;


   // Free the memory.
   delete [] numbers;
   numbers = 0;
   return 0;

}
```

```cpp
/* The getRandomNumbers function returns a
   pointer to an array of random integers. The
   parameter indicates the number of numbers
   requested.
*/
int *getRandomNumbers(int num)
{
   int *array;    // Array to hold the numbers

   // Return null if num is zero or negative.
   if (num <= 0) return NULL;

   // Dynamically allocate the array.
   array = new int[num];

// Seed the random number generator by passing
//the return value of time(0) to srand.
   srand( time(0) );

   // Populate the array with random numbers.
   for (int count = 0; count < num; count++)
      array[count] = rand();

   // Return a pointer to the array.
   return array;
}
```

Returning a Boolean Value

- Function can return **true** or **false**

- Declare return type in function prototype and heading as **bool**

- Function body must contain **return** statement(s) that return **true** or **false**

- Calling function can use return value in a relational expression

return Boolean

```
bool isValid(int);          // prototype

bool isValid(int val)       // heading
{
    int min = 0, max = 100;
    if (val >= min && val <= max)
        return true;
    else
        return false;
}

if (isValid(score))         // call
    …
```

Returning multiple values

- C++ doesn't contain a direct method for passing multiple values back to the caller. Fortunately, there are several indirect methods that can be used.

- A tuple is a sequence of elements that may be different types, where the type of each element must be explicitly specified.

```cpp
1   #include <tuple>
2   #include <iostream>
3
4   std::tuple<int, double> returnTuple() // return a tuple that contains an int and a double
5   {
6       return std::make_tuple(5, 6.7); // use std::make_tuple() as shortcut to make a tuple to return
7   }
8
9   int main()
10  {
11      int a;
12      double b;
13      std::tie(a, b) = returnTuple(); // put elements of tuple in variables a and b
14      std::cout << a << ' ' << b << '\n';
15
16      return 0;
17  }
```

- You can also use std::tie to unpack the tuple into predefined variables.

Returning multiple values

- You can also choose to put multiple returns as referenced arguments.

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

void addOne(int &a, int &b){
    a += 1;
    b += 1;
    return;
}


int main(){
    int a=1, b=1;
    addOne(a, b);
    cout << a << "," << b;
    return 0;
}
```

Function Overloading

Overloading Functions

- **Function overloading** is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.

- Consider example below:

```
1   int add(int x, int y)
2   {
3       return x + y;
4   }
```

- Question: what if we also need to add two floating point numbers?

```
int addInteger(int x, int y)
{
    return x + y;
}
```

```
double addDouble(double x, double y)
{
    return x + y;
}
```

- However, for best effect, this requires that you define a consistent naming standard, remember the name of all the different flavors of the function, and call the correct one.

Overloading Functions

- Function overloading provides a better solution. Using function overloading, we can simply declare another add() function that takes double parameters.

- We now have two version of add():

```
int add(int x, int y); // integer version
double add(double x, double y); // floating point version
```

- it's also possible to define add() functions with a differing number of parameters:

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

- The compiler is able to determine which version of add() to call **based on the arguments** used in the function call.

Overloading Functions

- Function return types are not considered for uniqueness.

```
int getRandomValue();
double getRandomValue();
```

- This will generate error. These two functions have the same parameters (none), and consequently, the second getRandomValue() will be treated as an erroneous redeclaration of the first.

Overloaded Functions Example

If a program has these overloaded functions,
```
void getDimensions(int);            // 1
void getDimensions(int, int);       // 2
void getDimensions(int, float);     // 3
void getDimensions(double, double); // 4
```
then the compiler will use them as follows:
```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```

Default Arguments

- A **default parameter** (also called an **optional parameter** or a **default argument**) is a function parameter that has a default value provided to it.

- If the user does not supply a value for this parameter, the default value will be used.

- If the user does supply a value for the default parameter, the user-supplied value is used instead of the default value.

```cpp
void printValues(int x, int y=10)
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}

int main()
{
    printValues(1); // y will use default parameter of 10
    printValues(3, 4); // y will use user-supplied value 4
}
```

This program produces the following output:

```
x: 1
y: 10
x: 3
y: 4
```

Default Arguments

- All default parameters must be the rightmost parameters. The following is not allowed:

```
void printValue(int x=10, int y); // not allowed
```

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);      // OK
sum = getSum(num1, , num3);  // wrong!
```

Default Arguments

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int,int=0,int=0);// OK

int getSum(int,int=0,int);   // wrong!
```

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);      // OK

sum = getSum(num1, , num3);    // wrong!
```

Default parameters and function overloading

- Default parameters do **NOT** count towards the parameters that make the function unique.

- The following is not allowed:

```cpp
void printValues(int x);
void printValues(int x, int y=20);
```

Local and Global Variables

Local and Global Variables

- local variable: defined within a function or block; accessible only within the function or block

- Other functions and blocks can define variables with the same name

- When a function is called, local variables in the calling function are not accessible from within the called function

Local and Global Variables

- global variable: a variable defined outside all functions; it is accessible to all functions within its scope

- Easy way to share large amounts of data between functions

- Scope of a global variable is from its point of definition to the program end

- Use sparingly

Local Variable Lifetime

- A local variable only exists while its defining function is executing

- Local variables are destroyed when the function terminates

- Data cannot be retained in local variables between calls to the function in which they are defined

Initializing Local and Global Variables

- Local variables must be initialized by the programmer

- Global variables are initialized to `0` (numeric) or `NULL` (character) when the variable is defined

Global Variables – Why Use Sparingly?

Global variables make:

- Programs that are difficult to debug

- Functions that cannot easily be re-used in other programs

- Programs that are hard to understand

Local and Global Variable Names

- Local variables can have same names as global variables

- When a function contains a local variable that has the same name as a global variable, the global variable is unavailable from within the function. The local definition "hides" or "shadows" the global definition.

Example

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

// Variables declared outside of a function are global variables
int g_x; // global variable g_x
const int g_y(2); // global variable g_y

void doSomething()
{
    // global variables can be seen and used everywhere in program
    g_x = 3;
    std::cout << g_x << "\n";
}

int main()
{
    // global variables can be seen and used everywhere in program
    cout << g_x << "\n";
    doSomething();
    cout << g_x << "\n";

    std::cout << g_y << "\n";

    return 0;
}
```

Result: 0 3 3 2

Recursive Function

Recursion

- A **recursive function** is a function that calls itself.

```cpp
#include <iostream>

void countDown(int count)
{
    std::cout << "push " << count << '\n';
    countDown(count-1); // countDown() calls itself recursively
}

int main()
{
    countDown(5);

    return 0;
}
```

- The sequence of countDown(n) calling countDown(n-1) is repeated indefinitely.

- The computer will run out of stack memory, stack overflow will result, and the program will crash or terminate.

Recursive termination conditions

- Recursive function calls generally work just like normal function calls.

- You must include a recursive **termination condition**, or they will run "forever" (actually, until the call stack runs out of memory).

```cpp
#include <iostream>

void countDown(int count)
{
    std::cout << "push " << count << '\n';

    if (count > 1) // termination condition
        countDown(count-1);

    std::cout << "pop " << count << '\n';
}

int main()
{
    countDown(5);
    return 0;
}
```

```
push 5
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4
pop 5
```

Recursion Example 1: Summation

```
// return the sum of all the integers between 1 (inclusive) and sumto (inclusive)
// returns 0 for negative numbers
int sumTo(int sumto)
{
    if (sumto <= 0)
        return 0; // base case (termination condition) when user passed in an unexpected parameter (0 or negative)
    else if (sumto == 1)
        return 1; // normal base case (termination condition)
    else
        return sumTo(sumto - 1) + sumto; // recursive function call
}
```

- Recursive programs are often hard to figure out just by looking at them.
- It's often instructive to see what happens when we call a recursive function with a particular value. So let's see what happens when we call this function with parameter sum to = 5.

```
sumTo(5) called, 5 <= 1 is false, so we return sumTo(4) + 5.
sumTo(4) called, 4 <= 1 is false, so we return sumTo(3) + 4.
sumTo(3) called, 3 <= 1 is false, so we return sumTo(2) + 3.
sumTo(2) called, 2 <= 1 is false, so we return sumTo(1) + 2.
sumTo(1) called, 1 <= 1 is true, so we return 1.  This is the termination condition.
```

- Now unwind the call stack

```
sumTo(1) returns 1.
sumTo(2) returns sumTo(1) + 2, which is 1 + 2 = 3.
sumTo(3) returns sumTo(2) + 3, which is 3 + 3 = 6.
sumTo(4) returns sumTo(3) + 4, which is 6 + 4 = 10.
sumTo(5) returns sumTo(4) + 5, which is 10 + 5 = 15.
```

Recursion Example 2: Fibonacci number

$$F(n) = \begin{cases} 0 \text{ if } n = 0 \\ 1 \text{ if } n = 1 \\ f(n-1) + f(n-2) \text{ if } n > 1 \end{cases}$$

```cpp
#include <iostream>

int fibonacci(int number)
{
    if (number == 0)
        return 0; // base case (termination condition)
    if (number == 1)
        return 1; // base case (termination condition)
    return fibonacci(number-1) + fibonacci(number-2);
}

// And a main program to display the first 13 Fibonacci numbers
int main()
{
    for (int count=0; count < 13; ++count)
        std:: cout << fibonacci(count) << " ";

    return 0;
}
```

Recursion Example 3: Factorial

```cpp
#include <iostream>
using namespace std;
//Factorial function
int f(int n){
   /* This is called the base condition, it is
    * very important to specify the base condition
    * in recursion, otherwise your program will throw
    * stack overflow error.
    */
   if (n <= 1)
        return 1;
   else
       return n*f(n-1);
}
int main(){
   int num;
   cout<<"Enter a number: ";
   cin>>num;
   cout<<"Factorial of entered number: "<<f(num);
   return 0;
}
```

Recursive vs iterative

- You can always solve a recursive problem iteratively -- however, for non-trivial problems, the recursive version is often much simpler to write (and read).

- For example, while it's possible to write the Fibonacci function iteratively, it's a little more difficult! (Try it!)

- Iterative functions (those using a for-loop or while-loop) are almost always more efficient than their recursive counterparts. Overhead that takes place in pushing and popping stack frames.

- That's not to say iterative functions are always a better choice. Sometimes the recursive implementation of a function is so much cleaner and easier to follow that incurring a little extra overhead is more than worth it for the benefit in maintainability

# C++ Separate Header and Implementation Files

Header (.h) files

Consider the following program:

```cpp
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

• This program prints "Hello, world!" to the console using cout. However, this program never defines cout, so how does the compiler know what cout is? The answer is that cout has been declared in a header file (.h) called "iostream".

• When we use the line #include <iostream>, we're requesting that all of the content from the header file named "iostream" be **copied** into the including file.

• Keep in mind that header files typically only contain declarations. They do not define how something is *implemented*. So if cout is only *declared* in the "iostream" header file, where is it actually defined? It is implemented in the C++ runtime support library (compiled .cpp file), which is automatically linked into your program during the link phase.

Compile – Link – Execution

The whole process looks like below:
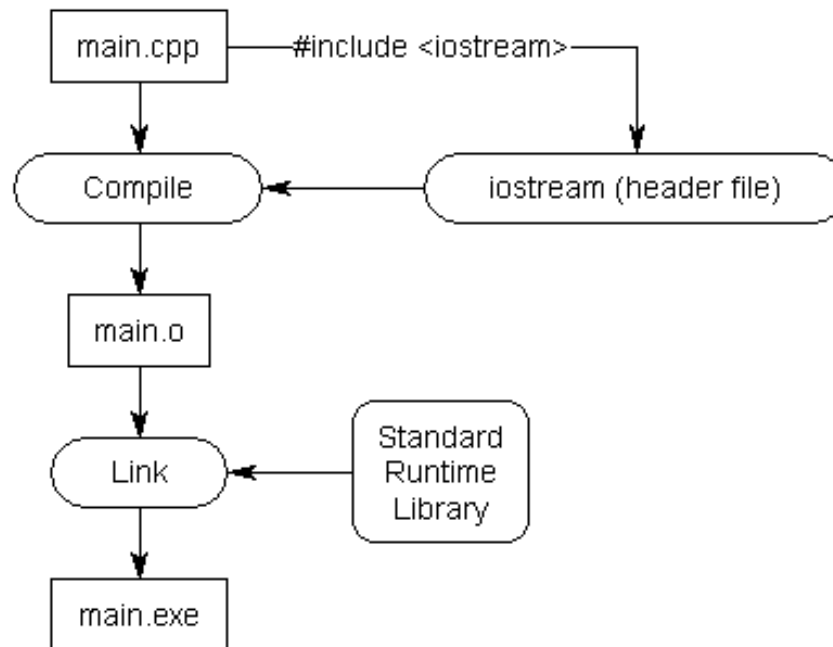
We write our own library

- add.h:

```
// This is start of the header guard.  ADD_H can be any unique name.  By convention, we use the name of the header file.
#ifndef ADD_H
#define ADD_H

// This is the content of the .h file, which is where the declarations go
int add(int x, int y); // function prototype for add.h -- don't forget the semicolon!

// This is the end of the header guard
#endif
```

- main.cpp

```
#include <iostream>
#include "add.h"

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << std::endl;
    return 0;
}
```

- add.cpp

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

Compile – Link – Execution