

C + + Programming

Week 5:

Functions

Dr. Owen Chen

Cary Chinese School

Director of Math and Computer Science

2023 Summer

Week 5: Agenda

- Review Arrays, Strings, Vectors
- Review Homework 4)
- New Topic: Functions

Two types of strings in C++

- In C++, we have two types of strings:

1) `std::string`

- The `std::string` class that's provided by the C++ Standard Library is the preferred method to use for strings.

2) C-style Strings

- These are strings derived from the C programming language and they continue to be supported in C++.
- These "collections of characters" are stored in the form of arrays of type `char` that are null-terminated (the `\0` null character).
- C-style strings are relatively unsafe and not recommended.

Defining and Initializing `string`s

Each class defines how objects of its type can be initialized. A class may define many different ways to initialize objects of its type. Each way must be distinguished from the others either by the number of initializers that we supply, or by the types of those initializers. Table 3.1 lists the most common ways to initialize `string`s. Some examples:

Table 3.1: Ways to Initialize a `string`

<code>string s1</code>	Default initialization; <code>s1</code> is the empty string.
<code>string s2(s1)</code>	<code>s2</code> is a copy of <code>s1</code> .
<code>string s2 = s1</code>	Equivalent to <code>s2(s1)</code> , <code>s2</code> is a copy of <code>s1</code> .
<code>string s3("value")</code>	<code>s3</code> is a copy of the string literal, not including the null.
<code>string s3 = "value"</code>	Equivalent to <code>s3("value")</code> , <code>s3</code> is a copy of the string literal.
<code>string s4(n, 'c')</code>	Initialize <code>s4</code> with <code>n</code> copies of the character <code>'c'</code> .

string Operations

Table 3.2: string Operations

<code>os << s</code>	Writes <code>s</code> onto output stream <code>os</code> . Returns <code>os</code> .
<code>is >> s</code>	Reads whitespace-separated string from <code>is</code> into <code>s</code> . Returns <code>is</code> .
<code>getline(is, s)</code>	Reads a line of input from <code>is</code> into <code>s</code> . Returns <code>is</code> .
<code>s.empty()</code>	Returns <code>true</code> if <code>s</code> is empty; otherwise returns <code>false</code> .
<code>s.size()</code>	Returns the number of characters in <code>s</code> .
<code>s[n]</code>	Returns a reference to the <code>char</code> at position <code>n</code> in <code>s</code> ; positions start at 0.
<code>s1 + s2</code>	Returns a string that is the concatenation of <code>s1</code> and <code>s2</code> .
<code>s1 = s2</code>	Replaces characters in <code>s1</code> with a copy of <code>s2</code> .
<code>s1 == s2</code>	The strings <code>s1</code> and <code>s2</code> are equal if they contain the same characters.
<code>s1 != s2</code>	Equality is case-sensitive.
<code><, <=, >, >=</code>	Comparisons are case-sensitive and use dictionary ordering.

string ctype functions

Table 3.3: ctype Functions

<code>isalnum(c)</code>	true if <code>c</code> is a letter or a digit.
<code>isalpha(c)</code>	true if <code>c</code> is a letter.
<code>iscntrl(c)</code>	true if <code>c</code> is a control character.
<code>isdigit(c)</code>	true if <code>c</code> is a digit.
<code>isgraph(c)</code>	true if <code>c</code> is not a space but is printable.
<code>islower(c)</code>	true if <code>c</code> is a lowercase letter.
<code>isprint(c)</code>	true if <code>c</code> is a printable character (i.e., a space or a character that has a visible representation).
<code>ispunct(c)</code>	true if <code>c</code> is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
<code>isspace(c)</code>	true if <code>c</code> is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
<code>isupper(c)</code>	true if <code>c</code> is an uppercase letter.
<code>isxdigit(c)</code>	true if <code>c</code> is a hexadecimal digit.
<code>tolower(c)</code>	If <code>c</code> is an uppercase letter, returns its lowercase equivalent; otherwise returns <code>c</code> unchanged.
<code>toupper(c)</code>	If <code>c</code> is a lowercase letter, returns its uppercase equivalent; otherwise returns <code>c</code> unchanged.

Arrays - C-style arrays

- Syntax:
- Type `array name[size] {...};`

```
#include <iostream>
using namespace std;
int main()
{
    int num_array1[5]; // int array
    int num_array2[5] = {0, 1, 2, 3, 4}; // int array
    float f_array1[5]{0.0, 1.0, 2.0, 3.0, 4.0}; // float array
    string s_array1[5>{"w1", "w2", "w3", "w4", "w5"}; // string array
}
```

Arrays - C++ STL arrays since C++11

- Syntax:
- `#include <array>`
- `Array<T, size> array_name {...};`

```
#include <iostream>
#include <array>
#include <string>
using namespace std;
int main()
{
    array<int, 5> num_array1{9}; //initialized int array
    array<int, 5> num_array2{0, 1, 2, 3, 4}; //initialized int array
    array<float, 5> float_array3{0.0, 1.0, 2.0, 3.0, 4.0}; //float array
    array<string, 5> s_array5{ "", "", "", "", "" }; //string array
}
```


Vector

- A **vector** is a collection of objects, all of which have the same type. Every object in the collection has an associated index, which gives access to that object. A `vector` is often referred to as a **container** because it “contains” other objects.
- To use a `vector`, we must include the appropriate header. In our examples, we also assume that an appropriate `using` declaration is made:

```
#include <vector>
using std::vector;
```

- A `vector` is a **class template**. C++ has both class and function templates.

Vector Initialization

Table 3.4: Ways to Initialize a vector

<code>vector<T> v1</code>	vector that holds objects of type T. Default initialization; v1 is empty.
<code>vector<T> v2 (v1)</code>	v2 has a copy of each element in v1.
<code>vector<T> v2 = v1</code>	Equivalent to <code>v2 (v1)</code> , v2 is a copy of the elements in v1.
<code>vector<T> v3 (n, val)</code>	v3 has n elements with value val.
<code>vector<T> v4 (n)</code>	v4 has n copies of a value-initialized object.
<code>vector<T> v5 {a,b,c ... }</code>	v5 has as many elements as there are initializers; elements are initialized by corresponding initializers.
<code>vector<T> v5 = {a,b,c ... }</code>	Equivalent to <code>v5 {a,b,c ... }</code> .

Vector Operations

Table 3.5: vector Operations

<code>v.empty()</code>	Returns <code>true</code> if <code>v</code> is empty; otherwise returns <code>false</code> .
<code>v.size()</code>	Returns the number of elements in <code>v</code> .
<code>v.push_back(t)</code>	Adds an element with value <code>t</code> to end of <code>v</code> .
<code>v[n]</code>	Returns a reference to the element at position <code>n</code> in <code>v</code> .
<code>v1 = v2</code>	Replaces the elements in <code>v1</code> with a copy of the elements in <code>v2</code> .
<code>v1 = {a, b, c ...}</code>	Replaces the elements in <code>v1</code> with a copy of the elements in the comma-separated list.
<code>v1 == v2</code>	<code>v1</code> and <code>v2</code> are equal if they have the same number of elements and each element in <code>v1</code> is equal to the corresponding element in <code>v2</code> .
<code>v1 != v2</code>	
<code><, <=, >, >=</code>	Have their normal meanings using dictionary ordering.

Range-based for Loop

C++11 introduces the **range-based for loop** to simplify the verbosity of traditional `for` loop constructs. They are equivalent to the `for` loop operating over a range of values, but **safer**

The range-based `for` loop avoids the user to specify start, end, and increment of the loop

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";    // print: 3 2 1
```

```
int values[] = { 3, 2, 1 };
for (int v : values)    // ARRAY OF VALUES
    cout << v << " ";  // print: 3 2 1
```

```
for (auto c : "abcd")    // RAW STRING
    cout << c << " ";    // print: a b c d
```

Homework 4

1) What does the following program do? Is it valid? If not, why not?

```
string s;  
cout << s[0] << endl;
```

Homework 4

1) What does the following program do? Is it valid? If not, why not?

```
string s;  
cout << s[0] << endl;
```

Solution:

The program declares a string variable `s` and outputs the first character of the string `s`. It is valid. However, since String `s` is not initialized, `s[0]` could be blank or any random value.

Homework 4

2) Which, if any, of the following vector definitions are in error? For those that are legal, explain what the definition does. For those that are not legal, explain why they are illegal.

(a) `vector<vector<int>> ivec;`

(b) `vector<string> svec = ivec;`

(c) `vector<string> svec(10, "null");`

Homework 4

2) Which, if any, of the following vector definitions are in error? For those that are legal, explain what the definition does. For those that are not legal, explain why they are illegal.

(a) `vector<vector<int>> ivec;`

(b) `vector<string> svec = ivec;`

(c) `vector<string> svec(10, "null");`

Solution:

(a) is legal. It declares ivec as a 2-dimension vector of integers (vector of vector of integers).

(b) It is invalid. Can not assign a 2-d vector to a string of vector.

(c) It is valid. svec is a vector of 10 string elements with each element being initialized as "null".

Homework 4

3) How many elements are there in each of the following vectors? What are the values of the elements?

(a) `vector<int> v1;`

(b) `vector<int> v2(10);`

(c) `vector<int> v3(10, 42);`

(d) `vector<int> v4{10};`

(e) `vector<int> v5{10, 42};`

(f) `vector<string> v6{10};`

(g) `vector<string> v7{10, "hi"};`

Homework 4

3) How many elements are there in each of the following vectors? What are the values of the elements?

(a) `vector<int> v1;`

(b) `vector<int> v2(10);`

(c) `vector<int> v3(10, 42);`

(d) `vector<int> v4{10};`

(e) `vector<int> v5{10, 42};`

(f) `vector<string> v6{10};`

(g) `vector<string> v7{10, "hi"};`

Solution:

v1: 0 elements

v2: 10 elements of 0

v3: 10 elements of 42

v4: 1 element of 10

v5: 2 elements of 10 and 42

v6: 10 elements of blank strings

v7: 10 elements of "hi"

Homework 4

4) Is there any error in the following code?

```
int array_size = 10;  
int ia[array_size];  
for (int ix = 1; ix <= array_size; ++ix)  
    ia[ix] = ix;
```

Homework 4

4) Is there any error in the following code?

```
int array_size = 10;  
int ia[array_size];  
for (int ix = 1; ix <= array_size; ++ix)  
    ia[ix] = ix;
```

Solution:

- The code will compile and run without any error.
- However, the last iteration of the for loop, ia[ix] with ix=10 is out of bound.
- ia[10] memory is not allocated - its value is unstable.
- It could override another variable's value and it could also be overridden by another variable.

Homework 4 - Programming

- 5) Write a program that reads a string of characters including punctuation and writes what was read but with the punctuation removed.
- 6) Write a program to read a sequence of *ints* from *cin* and store those values in a *vector*.
- 7) Modify your program in 6) above to read *strings* this time.
- 8) Write a program to copy a *vector* of *ints* into an *array* of *ints*.

Solutions

- homework5_p5.cpp
- homework5_p6.cpp
- homework5_p7.cpp
- homework5_p8.cpp

Functions

Function Overview

A **function** (**procedure** or **routine**) is a piece of code that performs a *specific task*. Function is a block of code which only runs when it is called.

Purpose:

- **Avoiding code duplication:** less code for the same functionality → less bugs
- **Readability:** better express what the code does
- **Organization:** break the code in separate modules

Function Overview

- Function is a block of code with a name.
- Declare a function with its name, parameters and return type
- Define a function with details
- Execute a function by calling the function
- A function takes zero or more arguments and usually returns a result.
- Functions can be overloaded, meaning that the same name may have different arguments and different return values

Declaration/Definition

Declaration/Prototype

A **declaration** (or *prototype*) of an entity is an identifier describing its type

A declaration is what the compiler and the linker needs to accept references (usage) to that identifier

C++ entities (class, functions, etc.) can be declared multiple times (with the same signature)

Definition/Implementation

An entity **definition** is the implementation of a declaration

For each entity, only a single *definition* is allowed

Declaration/Definition Function Example

```
void f(int a, string b); // function declaration -> put a function declaration in a .h file
```

```
void f(int a, string) { // function definition -> put a function definition in a .cpp file  
    ...                // "b" can be omitted if not used  
}
```

```
void f(int a, string b); // function declaration  
                        // multiple declarations is valid
```

```
f(3, "abc");           // usage
```

```
void g(); // function declaration
```

```
g(); // linking error "g" is not defined
```

Examples: Declare a function

```
#ifndef LOCALMATH_H
#define LOCALMATH_H

//definition in LocalMath.cc
int fact(int);          // iterative definition of factorial
#endif
```

Examples: define a function: LocalMath.cpp

```
#include "LocalMath.h"

// factorial of val is val*(val-1)*(val-2) . . . * ((val-(val-1)) * 1)
long fact(int val)
{
    long ret = 1; // local variable to hold the result as we calculate it
    while (val > 1)
        ret *= val--; // assign ret * val to ret and decrement val
    return ret;      // return the result
}
```

Examples: call a function from main(): fact.cpp

```
#include <iostream>
using namespace std;
#include "LocalMath.h"

int main()
{
    cout << fact(5) << endl;
    cout << fact(0) << endl;

    return 0;
}
```

Recursion

A function that calls itself, either directly or indirectly, is a *recursive function*.

Examples: Recursive Function

```
#include "LocalMath.h"

// recursive version of factorial:
// calculate val!, which is 1 * 2 * 3 . . . * val
long factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

Recursion

Trace of factorial(5)

Call

Returns

Value

factorial(5)	factorial(4) * 5	120
factorial(4)	factorial(3) * 4	24
factorial(3)	factorial(2) * 3	6
factorial(2)	factorial(1) * 2	2
factorial(1)	1	1

Examples: Compare both factorial functions

```
#include <iostream>
#include <array>
#include <chrono>
using namespace std;
using namespace std::chrono;
#include "LocalMath.h"

int main()
{
    array<int, 9> n{0, 5, 6, 7, 8, 9, 10, 20, 40};
    //Version 1 - while loop
    auto time_start = high_resolution_clock::now();
    cout << "Factorial function - Version 1: fact() - while loop" << endl;
    for(int i:n) cout << i << " ! = " << factorial(i) << endl;
    auto run_time = duration_cast<microseconds>(high_resolution_clock::now() - time_start);
    cout << "Elapsed Time = " << run_time.count() << " microseconds (1/million seconds)" << endl;

    //Version 2 - recursive function
    time_start = high_resolution_clock::now();
    cout << "Factorial function - Version 2: factorial() - recursive function" << endl;
    for(int i:n) cout << i << " ! = " << fact(i) << endl;
    run_time = duration_cast<microseconds>(high_resolution_clock::now() - time_start);
    cout << "Elapsed Time = " << run_time.count() << " microseconds (1/million seconds)" << endl;
    return 0;
}
```

Examples: Compile multiple C++ programs into one executable

```
# Step 1) Compile multiple C++ programs into objects first  
$ g++ -c LocalMath.cpp fact.cpp
```

```
# Step 2) Link object files into an executable  
$ g++ LocalMath.o fact.o -o fact.exe
```

```
# Combined one step  
$ g++ LocalMath.cpp fact.cpp -o fact.exe
```

Examples: Compare both factorial functions

```
~/cpp/week5$ fact.exe
Factorial function - Version 1: fact() - while loop
0! = 1
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
20! = 2432902008176640000
40! = -70609262346240000
Elapsed Time = 71 microseconds (1/million seconds)
Factorial function - Version 2: factorial() - recursive function
0! = 1
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
20! = 2432902008176640000
40! = -70609262346240000
Elapsed Time = 8 microseconds (1/million seconds)
```

Exercise: Generate Fibonacci numbers using a recursive function

```
// Week 3
int main()
{
    int a = 0, b = 1;
    int temp = 0;
    cout << "First 20 Fibonacci Numbers:" << endl;
    for(int i = 0; i < 20; i++){
        temp = b;
        b = a + b;
        a = temp;
        cout << a << " ";
    }
    cout << endl;

    return 0;
}
```

Exercise: Generate Fibonacci numbers using a recursive function

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

int fibonacci(int n)
{
    if (n < 3) {
        return 1;
    } else {
        return fibonacci(n-2) + fibonacci(n-1);
    }
}

void fibonacci_sequence(int n, vector<int> & fib_sequence)
{
    if (n == 1) {
        fib_sequence = {1};
    } else if (n == 2) {
        fib_sequence = {1, 1};
    } else {
        for(int i = 2; i <= n; i++){
            fib_sequence.push_back(fibonacci(i));
        }
    }
}
```

```
int main()
{
    vector<int> fib_seq;
    cout << "First 20 Fibonacci Numbers:" << endl;
    fibonacci_sequence(20, fib_seq);
    for (int fib: fib_seq) {
        cout << fib << " ";
    }
    cout << endl;
    return 0;
}
```

Function Parameter and Argument

Function Parameter [formal]

A **parameter** is the variable which is part of the method signature

Function Argument [actual]

An **argument** is the actual value (instance) of the variable that gets passed to the function

```
void f(int a, string b); // parameters: int a, string b
                        // return type: void

f(3, "abc");           // arguments: 3, "abc"
```

Where should a function be? Option 1

```
// draw.cpp
// The function must be defined before it was called
bool drawLine(int x1, int y1, int x2, int y2)
{
    // Source code here
    return true;
}
bool drawRectangle(int x1, int y1, int x2, int y2)
{
    // some calculation here
    drawLine(...);
    drawLine(...);
    drawLine(...);
    drawLine(...);
    return true;
}
```

Where should a function be? Option 2

```
// draw.cpp
// declared first, parameter names can be omitted
bool drawLine(int x1, int y1, int x2, int y2);
bool drawRectangle(int x1, int y1, int x2, int y2)
{
    // some calculation here
    drawLine(...);
    drawLine(...);
    drawLine(...);
    drawLine(...);
    return true;
}
// define it later
bool drawLine(int x1, int y1, int x2, int y2)
{
    // Source code here
    return true;
}
```


Where should a function be? Option 3

```
// draw.h
#ifndef __DRAW_H__
#define __DRAW_H__
bool drawLine(int x1, int y1, int x2, int y2);
bool drawRectangle(int x1, int y1, int x2, int y2);
#endif
```

```
// draw.cpp
#include <draw.h>
bool drawRectangle(int x1, int y1, int x2, int y2)
{
    // some calculation here
    drawLine(...);
    drawLine(...);
    drawLine(...);
    drawLine(...);
    return true;
}
// define it later
bool drawLine(int x1, int y1, int x2, int y2)
{
    // Source code here
    return true;
}
```

```
// main.cpp
#include <draw.h>

int main()
{
    // ...
    drawRectangle(10, 20, 50, 100);
    // ...
}
```

How are functions called?

A call stack can store information about the active functions of a program

Store the address the program returns after the function call

Store the registers

Store the local variables

//do some work of the called function

Restore the registers

Restore the local variables

Store the function returned result

Jump to the return address

The cost to call a function!

Function Parameters

The symbolic name for "data" that passes into a function.

Three ways to pass into a function:

- **Pass by value**
- **Pass by pointer**
- **Pass by reference**

Call-by-value

The object is copied and assigned to input arguments of the method $f(T\ x)$

Advantages:

- Changes made to the parameter inside the function have no effect on the argument

Disadvantages:

- Performance penalty if the copied arguments are large (e.g. a structure with a large array)

When to use:

- Built-in data type and small objects (≤ 8 bytes)

When not to use:

- Fixed size arrays which decay into pointers
- Large objects

Pass by-Pointer

Call-by-pointer

The address of a variable is copied and assigned to input arguments of the method
 $f(T^* \ x)$

Advantages:

- Allows a function to change the value of the argument
- Copy of the argument is not made (fast)

Disadvantages:

- The argument may be null pointer
- Dereferencing a pointer is slower than accessing a value directly

When to use:

- Raw arrays (use `const T*` if read-only)

When not to use:

- All other cases

Call-by-reference

The reference of a variable is copied and assigned to input arguments of the method
`f(T& x)`

Advantages:

- Allows a function to change the value of the argument (better readability compared with pointers)
- Copy of the argument is not made (fast)
- References must be initialized (no null pointer)
- Avoid implicit conversion (without `const T&`)

When to use:

- All cases except raw pointers

When not to use:

- Pass by-value *could* give performance advantages and improve the readability with built-in data type and small objects

Examples

```
struct MyStruct;

void f1(int a);           // pass by-value
void f2(int& a);          // pass by-reference
void f3(const int& a);    // pass by-const reference
void f4(MyStruct& a);     // pass by-reference

void f5(int* a);          // pass by-pointer
void f6(const int* a);    // pass by-const pointer
void f7(MyStruct* a);     // pass by-pointer

void f8(int*& a);          // pass a pointer by-reference
//-----
char c = 'a';
f1(c);    // ok, pass by-value (implicit conversion)
// f2(c); // compile error different types
f3(c);    // ok, pass by-value (implicit conversion)
```

Pass by value: fundamental type

The parameter is a copy of the original variable

```
int foo(int x)
{ // x is a copy
  x += 10;
  return x;
}
```

```
int main()
{
  int num1 = 20;
  int num2 = foo( num1 );

  return 0;
}
```

Will `num1` be changed in `foo()`?



Pass by value: pointer

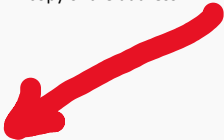
What's the difference?

```
int foo(int * p)
{
    (*p) += 10;
    return *p;
}
```

It still is passing by value (the address!)

A copy of the address

```
int main()
{
    int num1 = 20;
    int * p = &num1;
    int num2 = foo( p );
    return 0;
}
```



param-pointer.cpp

Pass by value: structure

How about structure parameter?

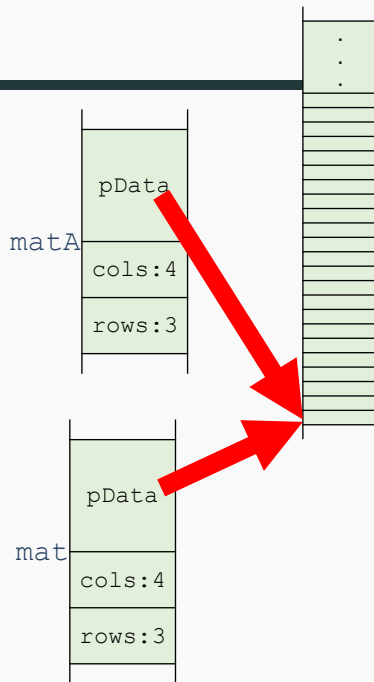
```
struct Matrix
{
    int rows;
    int cols;
    float * pData;
};

float matrix_max(struct Matrix mat)
{
    float max = FLT_MIN;
    for(int r = 0; r < mat.rows; r++)
        for (int c = 0; c < mat.cols; c++)
        {
            float val = mat.pData[ r * mat.cols + c];
            max = ( max > val ? max : val);
        }
    return max;
}
```

Pass by value: structure

```
Matrix matA = {3,4};  
matrix_max(matA);
```

```
float matrix_max(struct Matrix mat)  
{  
    float max = FLT_MIN;  
    for(int r = 0; r < mat.rows; r++)  
        for (int c = 0; c < mat.cols; c++)  
        {  
            float val = mat.pData[ r * mat.cols + c];  
            max = ( max > val ? max : val);  
        }  
    return max;  
}
```



Pass by value: structure

If the structure is a huge one, such as 1K bytes.

A copy will cost 1KB memory, and time consuming to copy it.



References in C++

References are in C++, not in C.

A reference is an alias to an already-existing variable/object.

```
int n = 0;  
int &r = n;    //r is a reference to n  
r = 10;       // what is the value of n?  
int i = 42;  
r = i;        //What is the value of n?
```

reference.cpp

References in C++

A reference to an object

```
struct Matrix
{
    int rows;
    int cols;
    float * pData;
};
```

```
Matrix matA = {3,4};
matA.pData = new float[matA.rows * matA.cols]{};
```

```
Matrix& matA_ref = matA;
```

```
Matrix * pMatA = &matA;
```

References in C++

A reference must be initialized after its declaration.

Reference VS Pointer: References are much safer

```
int& num_ref; // error  
Matrix& mat_ref; // error
```

structure

If the huge struct is passed as a function parameter

```
struct PersonInfo
{
    char firstname[256];
    char middlename[256];
    char lastname[256];
    char address[256];
    char nationalID[16];
    // and more
};

char * fullname(struct PersonInfo pi)
{
    // ...
}
```

- The data will be copied. Not a good choice!

The problem

One solution is to use a pointer

```
struct PersonInfo
{
    char firstname[256];
    char middlename[256];
    char lastname[256];
    char address[256];
    char nationalID[16];
    // and more
};

char * fullname(struct PersonInfo * ppi)
{
    if (ppi == NULL)
    {
        cerr << "Invalid pointer" << endl;
        return NULL;
    }
    // ...
}
```

References as function parameters


No data copying in the reference version; Better efficiency
The modification to a reference will affect the original object

```
struct Matrix
{
    int rows;
    int cols;
    float * pData;
};

float matrix_max(struct Matrix mat)
{
    float max = FLT_MIN;
    //find max value of mat
    for(int r = 0; r < mat.rows; r++)
        for (int c = 0; c < mat.cols; c++)
        {
            float val = mat.pData[ r * mat.cols + c];
            max = ( max > val ? max : val);
        }
    return max;
}
```

```
struct Matrix
{
    int rows;
    int cols;
    float * pData;
};


float matrix_max(struct Matrix &mat)
{
    float max = FLT_MIN;
    //find max value of mat
    for(int r = 0; r < mat.rows; r++)
        for (int c = 0; c < mat.cols; c++)
        {
            float val = mat.pData[ r * mat.cols + c];
            max = ( max > val ? max : val);
        }
    return max;
}
```



References as function parameters

To avoid the data is modified by mistakes,

```
float matrix_max(const struct Matrix& mat)
{
    float max = FLT_MIN;
    // ...
    return max;
}
```



Return statement

Statement `return;` is only valid if the function return type is `void`.
Just finish the execution of the function, no value returned.

```
void print_gender(bool isMale)
{
    if(isMale)
        cout << "Male" << endl;
    else
        cout << "Female" << endl;

    return;
}
```

```
void print_gender(bool isMale)
{
    if(isMale)
        cout << "Male" << endl;
    else
        cout << "Female" << endl;
}
```

Return statement

The return type can be a fundamental type or a compound type.

Pass by value:

Fundamental types: the value of a constant/variable is copied

Pointers: the address is copied

Structures: the whole structure is copied

```
float maxa = matrix_max(matA);
```

```
Matrix * pMat = create_matrix(4,5);
```

```
Matrix * create_matrix(int rows, int cols)
{
    Matrix * p = new Matrix{rows, cols};
    p->pData = new float[p->rows * p->cols]{1.f, 2.f, 3.f};
    // you should check if the memory is allocated successfully
    // and don't forget to release the memory
    return p;
}
```

If we have a lot to return

Such as a matrix addition function ($A+B \rightarrow C$)

A suggested prototype:

- To use references to avoid data copying

- To use const parameters to avoid the input data is modified

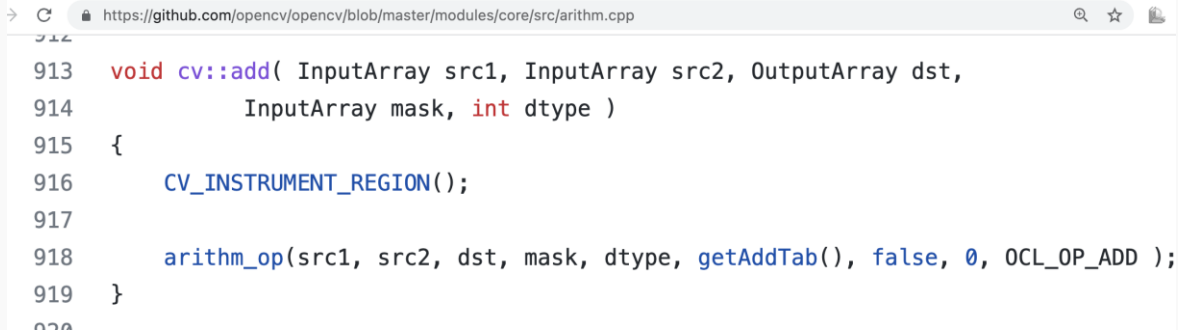
- To use non-const reference parameters to receive the output

```
bool matrix_add(const Matrix &matA, const Matrix &matB, Matrix &matC)
{
    // check the dimensions of the three matrices
    // re-create matC if needed
    // do: matC = matA + matB
    // return true if everything is right
}
```

Similar

Matrix add in OpenCV

<https://github.com/opencv/opencv/blob/master/modules/core/src/arithm.cpp>



The image shows a web browser window with the address bar displaying the URL `https://github.com/opencv/opencv/blob/master/modules/core/src/arithm.cpp`. The browser interface includes a search icon, a star icon, and a tab icon. The main content area displays a snippet of C++ code from the file `arithm.cpp`. The code is as follows:

```
912  
913 void cv::add( InputArray src1, InputArray src2, OutputArray dst,  
914              InputArray mask, int dtype )  
915 {  
916     CV_INSTRUMENT_REGION();  
917  
918     arithm_op(src1, src2, dst, mask, dtype, getAddTab(), false, 0, OCL_OP_ADD );  
919 }  
920
```

inline Functions Avoid Function Call Overhead

A function specified as **inline** (usually) is expanded “in line” at each call.

inline functions

Stack operations and jumps are needed for a function call.
It is a heavy cost for some frequently called **tiny** functions.

```
int main()
{
    int num1 = 20;
    int num2 = 30;
    int maxv = max_function(num1, num2);

    maxv = max_function(numn, maxv);
}
```

```
float max_function(float a, float b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
graph LR
    subgraph main [main]
        direction TB
        L1["int main()"]
        L2["{"]
        L3["int num1 = 20;"]
        L4["int num2 = 30;"]
        L5["int maxv = max_function(num1, num2);"]
        L6["maxv = max_function(numn, maxv);"]
        L7["}"]
    end
    subgraph max_function [max_function]
        direction TB
        F1["float max_function(float a, float b)"]
        F2["{"]
        F3["if (a > b)"]
        F4["return a;"]
        F5["else"]
        F6["return b;"]
        F7["}"]
    end
    L5 -- yellow arrow --> F1
    L6 -- yellow arrow --> F1
    F7 -- red arrow --> L5
    F7 -- red arrow --> L6
```

Inline functions

The generated instructions by a compiler can be as follows to improve efficiency

```
int main()
{
    int num1 = 20;
    int num2 = 30;
    int maxv = {if (num1 > num2)
                return num1;
                else
                return num2;}

    maxv = {if (numn > maxv)
            return numn;
            else
            return maxv;}
}
```

Inline functions

`inline` **suggests** the compiler to perform that kind of optimizations.

The compiler may not follow your suggestion if the function is too complex or contains some constraints. Some functions without `inline` may be optimized as an inline one.

```
inline float max_function(float a, float b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Preprocessor macro

```
#define MAX_MACRO(a, b) (a)>(b) ? (a) : (b)
```

The source code will be replaced by a preprocessor.
Surely no cost of a function call,
And `a`, `b` can be any types which can compare.

inline.cpp

Signature

Function signature defines the *input types* for a (specialized) function and the *inputs + outputs types* for a template function

A function signature includes the number of arguments, the types of arguments, and the order of the arguments

- The C++ standard prohibits a function declaration that only differs in the return type
- Function declarations with different signatures can have distinct return types

Overloading

Function overloading allows to have distinct functions with the same name but with different *signatures*

Overloaded Functions

Functions that have the same name but different parameter lists and that appear in the same scope are **overloaded**. For example, functions named `print`:

```
void print(const char *cp);  
void print(string s);  
void print(const int *beg, const int *end);  
void print(const int ia[], size_t size);
```

Function Signature and Overloading

```
void f(int a, char* b);           // signature: (int, char*)

// char f(int a, char* b);       // compile error same signature
// but different return types

void f(const int a, char* b);     // same signature, ok
// const int == int

void f(int a, const char* b);     // overloading with signature: (int, const char*)

int f(float);                    // overloading with signature: (float)
// the return type is different
```

Overloading Resolution Rules

- An exact match
- A promotion (e.g. char to int)
- A standard type conversion (e.g. float and int)
- A constructor or user-defined type conversion

```
void f(int a);  
void f(float b);           // overload  
void f(float b, char c);  // overload  
//-----  
    f(0);                 // ok  
// f('a');               // compile error ambiguous match  
    f(2.3f);              // ok  
// f(2.3);               // compile error ambiguous match  
    f(2.3, 'a');          // ok, standard type conversion
```


Function Default Parameters

Default/Optional parameter

A **default parameter** is a function parameter that has a default value

- If the user does not supply a value for this parameter, the default value will be used
- All default parameters must be the rightmost parameters
- Default parameters must be declared only once
- Default parameters can improve compile time and avoid redundant code because they avoid defining other overloaded functions

```
void f(int a, int b = 20);           // declaration

//void f(int a, int b = 10) { ... } // compile error, already set in the declaration

void f(int a, int b) { ... }        // definition, default value of "b" is already set

f(5); // b is 20
```

Function Default Parameters

```
typedef string::size_type sz; // typedef
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
string window;
window = screen(); // equivalent to screen(24,80,' ')
window = screen(66); // equivalent to screen(66,80,' ')
window = screen(66, 256); // screen(66,256,' ')
window = screen(66, 256, '#'); // screen(66,256,'#')
```