C++ Program Design -- Functions

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Function parameters and arguments

Parameters vs Arguments

- A **function parameter** (sometimes called a **formal parameter**) is a variable declared in the function declaration:

```
void foo(int x); // declaration (function prototype) -- x is a parameter
void foo(int x) // definition (also a declaration) -- x is a parameter
{}
```

- An **argument** (sometimes called an **actual parameter**) is the value that is passed to the function by the caller:

```
foo(6); // 6 is the argument passed to parameter x
foo(y+1); // the value of y+1 is the argument passed to parameter x
```

- When a function is called, all of the parameters are created as variables, and the value of the arguments are copied into them.

Passing arguments by value

- By default, arguments in C++ are passed by value.

```
void foo(int y){
    std::cout << "y = " << y << '\n';
    y = 6;
    std::cout << "y = " << y << '\n';
} // y is destroyed here

int main(){
    int x = 5;
    std::cout << "x = " << x << '\n';
    foo(x);
    std::cout << "x = " << x << '\n';
    return 0;
}
```

Passing arguments by value

- Advantages of passing by value:
  - **Easy:** Arguments passed by value can be variables (e.g. x), literals (e.g. 6), expressions (e.g. x+1), structs & classes, and enumerators.
  - **Safe:** Arguments are never changed by the function being called, which prevents side effects.

- Disadvantages of passing by value:
  - **Low performance:** Copying structs and classes can incur a significant performance penalty, especially if the function is called many times.

Passing arguments by reference

```
void foo(int &value){
    value = 6;
}

int main(){
    int value = 5;

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

Returning multiple values via out parameters

```
void getSinCos(double degrees, double &sinOut, double &cosOut){
    static const double pi = 3.14159265358979323846; // the value of pi
    double radians = degrees * pi / 180.0;
    sinOut = sin(radians);    cosOut = cos(radians);
}

int main(){
    double sin(0.0);    double cos(0.0);
    // getSinCos will return the sin and cos in variables sin and cos
    getSinCos(30.0, sin, cos);
    std::cout << "The sin is " << sin << '\n';
    std::cout << "The cos is " << cos << '\n';
    return 0;
}
```

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 changed!
}
```

Why using const reference instead of references?

- It tells the **compilers** help in ensuring values that shouldn't be changed aren't changed
- It tells the **programmer** that the function won't change the value of the argument. This can help with debugging.

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

Passing arguments by reference

- Advantages of passing by reference:
  - It allows a function to change the value of the argument,
  - Because a copy of the argument is not made, it is fast,
  - References can be used to return multiple values from a function.
  - References must be initialized, so there’s no worry about null values.

- Disadvantages of passing by reference:
  - Because a non-const reference cannot be made to an rvalue (e.g. a literal or an expression), reference arguments must be normal variables.
  - the programmer may not realize a function will change the value of the argument.

Passing arguments by address

```
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;
}
```

Passing arguments by address

- typically used with arrays and dynamically allocated variables.

```
void printArray(int *array, int length){
    for (int index=0; index < length; ++index)
        std::cout << array[index] << ' ';
}


int main(){
    int array[6] = { 6, 5, 4, 3, 2, 1 }; // remember, arrays decay into pointers
    printArray(array, 6); // so array evaluates to a pointer to
the first element of the array here, no & needed
}
```

Passing arguments by address

- Dereferencing a null pointer will typically cause the program to crash

```
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] << ' ';
}
```

Passing by const address

printArray() doesn’t modify any of its arguments

```
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] << ’ ’;
}
```

Addresses are 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.

- In other words, it’s passed by value!

- If you change the function parameter’s value, you are only changing a copy. Consequently, the original pointer argument will not be changed.

```
void setToNull(int *tempPtr)
{
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main(){
    int five = 5;    int *ptr = &five;
    std::cout << *ptr; // This will print 5

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

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

    return 0;
}
```

```cpp
void setToNull(int *tempPtr){
    *tempPtr = 6;
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main(){
    int five = 5;    int *ptr = &five;
    std::cout << *ptr; // This will print 5

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

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

    return 0;
}
```

Passing addresses by reference

```
void setToNull(int *&tempPtr){
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main(){
    int five = 5;     int *ptr = &five;
    std::cout << *ptr; // This will print 5

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

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

    return 0;
}
```

There is only pass by value

- we can conclude that C++ really passes everything by value!
  - passing by reference, address, and value
  - Reference parameter, pointer parameter, and normal value parameter

- The **properties** of pass by address (and reference) **comes *solely* from** the fact that we can dereference the passed address to change the argument, which we can not do with a normal value parameter!

Returning values by value, reference, and address

Return by address

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

Return by address

```
int* allocateArray(int size)
{
    return new int[size];
}

int main()
{
    int *array = allocateArray(25);

    // do stuff with array

    delete[] array;
    return 0;
}
```

Return by reference

```
int& doubleValue(int x)
{
    int value = x * 2;
    return value; // return a reference to value here
} // value is destroyed here
```

Return by reference

```
// Returns a reference to the index element of array
int& getElement(std::array<int, 25> &array, int index){
        return array[index];
}

int main(){
    std::array<int, 25> array;

    // Set the element of array with index 10 to the value 5
    getElement(array, 10) = 5;
    std::cout << array[10] << '\n';

    return 0;
}
```

This prints:

5

Quiz time

- Write function prototypes for each of the following functions. Use the most appropriate parameter and return types (by value, by address, or by reference), including use of const where appropriate.

- 1) A function named sumTo() that takes an integer parameter and returns the sum of all the numbers between 1 and the input number.

- 2) A function named printEmployeeName() that takes an Employee struct as input.

- 3) A function named minmax() that takes two integers as input and returns the smaller and larger number as separate parameters.

Quiz time

- Write function prototypes for each of the following functions. Use the most appropriate parameter and return types (by value, by address, or by reference), including use of const where appropriate.

- 4) A function named getIndexOfLargestValue() that takes an integer array (as a pointer) and an array size, and returns the index of the largest element in the array.

- 5) A function named getElement() that takes an integer array (as a pointer) and an index and returns the array element at that index (not a copy). Assume the index is valid, and the return value is const.

Inline functions

Inline functions

- Benefits of using functions
  - The code inside the function can be **reused**.
  - It is much **easier to change** or update the code in a function (which needs to be done once) than for every in-place instance. **Duplicate code is a recipe for disaster.**
  - It makes your code **easier to read and understand**, as you do not have to know how a function is implemented to understand what it does.

- one major **downside** of functions is that every time a function is called, there is a certain amount of **performance** overhead that occurs

- Code written in-place is significantly faster.
- For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run.

Inline functions

```
inline int min(int x, int y){
    return x > y ? y : x;
}
int main(){
    std::cout << min(5, 6) << '\n';
    std::cout << min(3, 2) << '\n';
    return 0;
}
```

```
int main(){
    std::cout << (5 > 6 ? 6 : 5) << '\n';
    std::cout << (3 > 2 ? 2 : 3) << '\n';
    return 0;
}
```

- This will execute quite a bit faster, at the cost of the compiled code being slightly larger.
- The compiler may ignore your request to inline a lengthy function.
- Modern compilers have gotten really good at inlining functions automatically

Function overloading

Function overloading

- create multiple functions with the same name, so long as they have different parameters.
- `int add(int x, int y); // integer version`
- `double add(double x, double y); // floating point version`
- `int add(int x, int y, int z)`

Function return types are not considered for uniqueness

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

- **Typedefs are not distinct**
- `typedef char *string;`
- `void print(string value);`
- `void print(char *value);`

How function calls are matched with overloaded functions

- 1) A match is found.
  - The call is resolved to a particular overloaded function

- 2) No match is found.
  - The arguments can not be matched to any overloaded function.

- 3) An ambiguous match is found.
  - The arguments matched more than one overloaded function.

no exact match

- If no exact match is found, C++ tries to find a match through promotion
  - Char, unsigned char, and short is promoted to an int.
  - Unsigned short can be promoted to int or unsigned int, depending on the size of an int
  - Float is promoted to double
  - Enum is promoted to int

- `void print(char *value);`
- `void print(int value);`
-
- `print('a'); // promoted to match print(int)`

no promotion is possible

- If no promotion is possible, C++ tries to find a match through standard conversion.

- `struct Employee; // defined somewhere else`
- `void print(float value);`
- `void print(Employee value);`
-
- `print('a'); // 'a' converted to match print(float)`

Ambiguous matches

```
void print(unsigned int value);
void print(float value);

// all the three function call are ambiguous => compile-time error
print('a'); // no promotion to int, may use standard conversion to both unsigned int and floating point value => ambiguous
print(0); // 0 is int, matches both calls via standard conversion.
print(3.14159); // 3.14159 is float, matches both calls via standard conversion

print(static_cast<unsigned int>(0)); // will call print(unsigned int)
```

Default parameters

Default parameters

```
void printValues(int x, int y=10){
    std::cout <<…}

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

void openLogFile(std::string filename="default.log");
int rollDie(int sides=6);
void printStringInColor(std::string, Color color=COLOR_BLACK); //
Color is an enum
```

Multiple default parameters

- `void printValues(int x=10, int y=20, int z=30)`
- `void printValue(int x=10, int y); // not allowed`

Default parameters can only be declared once

```
void printValues(int x, int y=10);

void printValues(int x, int y=10) // error: redefinition of default parameter
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

Rule: If the function has a forward declaration, put the default parameters there. Otherwise, put them on the function definition.

Default parameters and function overloading

- the following is not allowed:
- `void printValues(int x);`
- `void printValues(int x, int y=20);`

- Functions with default parameters may be overloaded
- `void print(std::string string);`
- `void print(char ch=' ');`
- If the user were to call print(), it would resolve to print(' '), which would p rint a space.

Function Pointers

Function Pointers

- Much like variables, functions live at an assigned address in memory.

```
int foo() // code for foo starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    foo(); // jump to address 0x002717f0

    return 0;
}
```

```cpp
int foo() // code starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    std::cout << foo;

    return 0;
}
```

On the author’s machine, this printed:

0x002717f0

If your machine doesn't print the function's address

```
int foo() // code starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    std::cout << reinterpret_cast<void*>(foo); // Tell C++ to interpret function foo as a void pointer

    return 0;
}
```

Pointers to functions

- // fcnPtr is a pointer to a function that takes no arguments and returns an integer
- int (*fcnPtr)();

- const int (* fcnPtr)(); // point to a function, who return a const int
- **int const (*fcnPtr)();** // the same

- int (*const fcnPtr)()=print;// point to a const function pointer. Assuming there is a function: int print()
- int (const* fcnPtr)();// compile error

Assigning a function to a function pointer

```
int foo();
double goo();
int hoo(int x);

// function pointer assignments
int (*fcnPtr1)() = foo; // okay
int (*fcnPtr2)() = goo; // wrong -- return types don't match!
double (*fcnPtr4)() = goo; // okay
fcnPtr1 = hoo; // wrong -- fcnPtr1 has no parameters, but hoo()
does
int (*fcnPtr3)(int) = hoo; // okay
```

Calling a function using a function pointer

- Call by explicit dereference

```
int foo(int x){    return x;}
int main(){
    int (*fcnPtr)(int) = foo; // assign fcnPtr to function foo
    (*fcnPtr)(5); // call function foo(5) through fcnPtr.
```

- Call by implicit dereference

```
    int (*fcnPtr)(int) = foo; // assign fcnPtr to function foo
    fcnPtr(5); // call function foo(5) through fcnPtr.
```

some older compilers do not support the implicit dereference method

Function pointers & default parameters

- Default parameters won’t work with function pointers.
- Default arguments are resolved at compile-time
- However, function pointers are resolved at run-time.
- You’ll explicitly have to pass in values for any defaulted parameters in this case.

Passing functions as arguments to other functions

- One of the most useful things to do with function pointers is pass a function as an argument to another function.
- Functions used as arguments to another function are sometimes called **callback functions**.

- Example: give the caller the ability to control how selection sort does its job.

```
// Note our user-defined comparison is the third parameter
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
{
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        int smallestIndex = startIndex;
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            if (comparisonFcn(array[smallestIndex], array[currentIndex])) // COMPARISON DONE HERE
                smallestIndex = currentIndex;
        }
        std::swap(array[startIndex], array[smallestIndex]);
    }}
```

```
bool ascending(int x, int y){
    return x > y;}
bool descending(int x, int y){
    return x < y;}

int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    selectionSort(array, 9, descending);
    printArray(array, 9);

    selectionSort(array, 9, ascending);
    printArray(array, 9);
```

- Is that cool or what? We've given the caller the ability to control how our selection sort does its job.

```
bool evensFirst(int x, int y){
     if ((x % 2 == 0) && !(y % 2 == 0))
          return false;
     if (!(x % 2 == 0) && (y % 2 == 0))
          return true;
     return ascending(x, y);
}
```

The above snippet produces the following result:
2 4 6 8 1 3 5 7 9

```
int main(){
    int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    selectionSort(array, 9, evensFirst);
    printArray(array, 9);
```

Providing default functions

- `// Default the sort to ascending sort`
- `void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int) = ascending);`

Making function pointers prettier with typedef

- `typedef bool (*validateFcn)(int, int);`

- Now instead of doing this:
- `bool validate(int x, int y, bool (*fcnPtr)(int, int));`

- You can do this:
- `bool validate(int nX, int nY, validateFcn pfcn)`

Using std::function in C++11

- Introduced in C++11, an alternate method of defining and storing function pointers is to use tempstd::function

```
std::function<bool(int, int)> fcn;


int foo() {     return 5;}
int goo() {     return 6;}


std::function<int()> fcnPtr; // declare function pointer that re
turns an int and takes no parameters
fcnPtr = goo; // fcnPtr now points to function goo
std::cout << fcnPtr(); // call the function just like normal
```

Quiz time!

- …

std::vector capacity and stack behavior

Size vs capacity

- In the context of a std::vector,
  - **size** is how many elements are being used in the array,
  - **capacity** is how many elements were allocated.

```
int main()
{
std::vector<int> array;
array = { 0, 1, 2, 3, 4 }; // okay, array length = 5
std::cout << "size: " << array.size() << "  capacity: " << array.capacity() << '\n';

array = { 9, 8, 7 }; // okay, array length is now 3!
std::cout << "size: " << array.size() << "  capacity: " << array.capacity() << '\n';
```

size: 5 capacity: 5
size: 3 capacity: 5

Stack behavior with std::vector

- push_back() pushes an element on the stack.
- back() returns the value of the top element on the stack.
- pop_back() pops an element off the stack.

- Example begins:

```
void printStack(const std::vector<int> &stack)
{
for (const auto &element : stack)
      std::cout << element << ' ';
std::cout << "(cap " << stack.capacity() << " size " << stack.size() << ")\n";
}
```

```cpp
std::vector<int> stack;  printStack(stack);

stack.push_back(5); printStack(stack);
stack.push_back(3); printStack(stack);
stack.push_back(2); printStack(stack);

std::cout << "top: " << stack.back() << '\n';

stack.pop_back(); printStack(stack);
stack.pop_back();printStack(stack);
stack.pop_back();printStack(stack);
```

```
(cap 0 size 0)

5 (cap 1 size 1)
5 3 (cap 2 size 2)
5 3 2 (cap 3 size 3)

top: 2

5 3 (cap 3 size 2)
5 (cap 3 size 1)
(cap 3 size 0)
```

resizing the vector is expensive, …

```
std::vector<int> stack;
stack.reserve(5); // Set the capacity to (at least) 5
printStack(stack);
```

(cap 5 size 0)

```
stack.push_back(5); printStack(stack);
stack.push_back(3); printStack(stack);
stack.push_back(2); printStack(stack);
```

5 (cap 5 size 1)
5 3 (cap 5 size 2)
5 3 2 (cap 5 size 3)

```
std::cout << "top: " << stack.back() << '\n';
```

top: 2

```
stack.pop_back(); printStack(stack);
stack.pop_back();printStack(stack);
stack.pop_back();printStack(stack);
```

5 3 (cap 5 size 2)
5 (cap 5 size 1)
(cap 5 size 0)

Recursion

Recursion

- A **recursive function** in C++ is a function that calls itself

```cpp
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);
```

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

A more useful example

```
// return the sum of 1 to value
int sumTo(int value)
{
    if (value <= 0)
        return 0; // base case (termination condition)
    else if (value == 1)
        return 1; // base case (termination condition)
    else
        return sumTo(value - 1) + value; // recursive function call
}
```

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).
  - **Iterative** functions (those using a for-loop or while-loop) are almost always more **efficient**

Handling errors (assert, cerr, exit, and exceptions)

Errors

- **Syntax errors: grammar of c++**
- **Semantic errors: syntax right, but does not do what you intended!**
  - **Logic error:** incorrectly codes the logic of a statement
    - `if (x >= 5)`
    - `    std::cout << "x is greater than 5";`
  - **violated assumption**

```
std::string hello = "Hello, world!";
std::cout << "Enter an index: ";

int index;
std::cin >> index;

std::cout << "Letter #" << index << " is " << hello [index] << std::
endl;
```

Defensive programming

- is a form of program design that involves trying to identify areas where assumptions may be violated,
  - writing code that detects and
  - handles any violation of those assumptions so that
  - the program reacts in a predictable way when those violations do occur.

```
void printString(const char *cstring)
{
    // Only print if cstring is non-null
    if (cstring)
        std::cout << cstring;
}
```

Defensive programming

```
int main(){
    std::string hello = "Hello, world!";
    std::cout << "Enter a letter: ";
    char ch;     std::cin >> ch;

    int index = hello.find(ch);
    if (index != -1) // handle case where find() failed to find the character in the string
        std::cout << ch << " was found at index " << index << '\n';
    else
        std::cout << ch << " wasn't found" << '\n';
```

```
std::string hello = "Hello, world!";     int index;
do{
   std::cout << "Enter an index: ";          std::cin >> index;
   if (std::cin.fail()) //handle case where user entered a non-intege
r
   {
       std::cin.clear(); // reset any error flags
       std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
       index = -1;
       continue;
    }
} while (index < 0 || index >= hello.size()); // handle case where us
er entered an out of range integer

std::cout << "Letter #" << index << " is " << hello [index] << std::e
ndl;
```

Handling assumption errors

- 1) Quietly skip the code that depends on the assumption being valid:

```
void printString(const char *cstring){
    if (cstring)
        std::cout << cstring;
}
```

- 2) return an error code

```
int getArrayValue(const std::array &array, int index){
    // use if statement to detect violated assumption
    if (index < 0 || index >= array.size())
        return -1; // return error code to caller

    return array[index];
}
```

Handling assumption errors

- 3) If we want to terminate the program immediately

```
int getArrayValue(const std::array &array, int index){
    // use if statement to detect violated assumption
    if (index < 0 || index >= array.size())
        exit(2); // terminate program and return error number 2 to OS
    return array[index];
}
```

- 4) If the user has entered invalid input, ask the user to enter the input again.

Handling assumption errors

- 5) cerr is another mechanism that is meant specifically for printing error messages.

```
void printString(const char *cstring) {
    // Only print if strString is non-null
    if (cstring)
        std::cout << cstring;
    else
        std::cerr << "function printString() received a null parameter";
}
```

- 6) If working in some kind of graphical environment (eg. MFC, SDL, QT, etc…), it is common to pop up a message box with an error code and then terminate the program.

Assert

- is a preprocessor macro that evaluates a conditional expression.

```
#include <cassert> // for assert()
int getArrayValue(const std::array<int, 10> &array, int index){
    // we’re asserting that index is between 0 and 9
    assert(index >= 0 && index <= 9); // this is line 6 in Test.cpp
    return array[index];
}
```

Assertion failed: index >= 0 && index <=9, file C:\\VCProjects\\Test.cpp, line 6

NDEBUG and other considerations

- The assert() function comes with a small performance cost that is incurred each time the assert condition is checked.
- Furthermore, asserts should (ideally) never be encountered in production code (because your code should already be thoroughly tested).
- Consequently, many developers prefer that asserts are only active in debug builds.
- C++ comes with a way to turn off asserts in production code: (project level, see project setting of your IDE)

```
#define NDEBUG

// all assert() calls will now be ignored to the end of the file
```

Exceptions

- C++ provides one more method for detecting and handling errors known as exception handling.

- The basic idea is that when an error occurs, the error is “thrown”.
- If the current function does not “catch” the error, the caller of the function has a chance to catch the error.
- If the caller does not catch the error, the caller’s caller has a chance to catch the error.
- The error progressively moves up the stack until it is either caught and handled,
- or until main() fails to handle the error. If nobody handles the error, the program typically terminates with an exception error.

Exception handling is an advanced C++ topic

comprehensive quiz

comprehensive quiz

- 1) Write function prototypes for the following cases. Use const if/when necessary.

- a) A function named max() that takes two doubles and returns the larger of the two.
- b) A function named swap() that swaps two integers.
- c) A function named getLargestElement() that takes a dynamically allocated array of integers and returns the largest number in such a way that the caller can change the value of the element returned (don’t forget the length parameter).

comprehensive quiz

- 2) What's wrong with these programs?

a)

```
int& doSomething(){
    int array[] = { 1, 2, 3, 4, 5 };
    return array[3];
}
```

- b)

```
int sumTo(int value){
    return value + sumTo(value - 1);
}
```

comprehensive quiz

- 2) What's wrong with these programs?

c)

```
float divide(float x, float y){
    return x / y;
}
double divide(float x, float y){
    return x / y;
}
```

comprehensive quiz

- 2) What's wrong with these programs?

d)

```
#include <iostream>

int main(int argc, char *argv[]){
    int times = argv[1];
    for (int count = 0; count < times; count++)
        std::cout << count << ' ';

    return 0;
}
```