# Passing By Value and Its Teaching in C++

Huchuan Fu[1, a], Ouyang Ji[2, b], Chen Qian[3, c]

[1] Department of Computer Science and Technology, Dongguan University of Technology
Dongguan, 523106, China

[2] Department of Computer Science and Technology, Dongguan University of Technology
Dongguan, 523106, China

[3] Department of Computer Science and Technology, Dongguan University of Technology
Dongguan, 523106, China

[a]email: hucf@dgut.edu.cn, [b]email: ouyj@dgut.edu.cn, [c]email: chenq@dgut.edu.cn

**Keywords:** Function; Variable; Passing by Value; Teaching Method

**Abstract.** Function call and passing by value is a very important content in C++ tutorials. It is difficult to understand or grasp too. This article introduced the features of C++'s function and passing by value briefly by analyzing data's storage in memory and the conception of runtime stack. It also discussed how to improve teaching's quality on such courses by means of these deep analyses.

## Introduction

Function, also called module generally, is a code assemble which can finish an independent work. Actually, C++ program is consisted of a series of functions [4]. As an independent functional module, a function could only interact with other functions throughout limited external interface. The interface is defined by function's prototype. There are some arguments needed before a function's running. It is called function's input data. There is also a value returned after a function finished its work. Generally speaking, the returned value is a hand over that called function should give to calling function. The progress of passing by value is done by combining the arguments with parameters when a function call happens. According to years' teaching, the author of this article found that most beginners can not understand this progress. The rest knows little. So, it is very meaningful to analyze how passing by value happened and how memory changed. This would be an important teaching method that makes beginners understand function call. This article elaborates the progress of function call and passing by value by disassembling function's executing code. It elaborates data's change in memory at the same time.

## Memory Variables

A variable's name and type must be known when it is defined in source code files. It is always expressed by words which are easy to read for programmers. However, CPU doesn't recognize literal words. When a program is running, values of variables are stored in memory [2]. Each unit of memory has a number called address. Objective code distinguishes different variables according to their own address in memory [1].

Variables' classification: (1) Global variable: Which is defined outside of any function. (2) Local variable: Which is defined inside of a function [3]. Most of C++'s variables are local. This article mainly elaborates local variables and function's passing by value.

Below is about these two kinds of variables' source code and disassembling code.

Source code:

```
int x;
void main(){
  int y;
```

```
  x = 1;
  y = 2;
}
```

Part of disassembling code when run in VC6.0 environment:

```
14:          x = 0;
00401268     mov          dword ptr [x (00432e10)],1
15:          y = 0;
00401272     mov          dword ptr [ebp-4],2
```

According to the source code and its correspondent disassembling code above, it could be found out that global variable is located by a fixed address (here is 00432e10). However, local variable is located by an address ebp-4. Ebp is a base pointer of a running stack-frame. That is to say, local variable could not be located by means of the same way that global variable does. It needs a special data structure which is called stack. The stack is also named running stack because it is related to runtime. The reason of doing so lies in: (1) Local variable is available when the function enclosing it has been called and unavailable as soon as the function returned. Most local variables' life time is less than that of their belonged programs. If compiler distributes allocation to local variables as it does the same way to global variables, the whole memory spaces' efficiency should be very low. (2) When a recursive function has been called, it appears that the function would call itself again and again before it could finish its work and return. In this circumstance, local variables with the very same name have different values and need different addresses certainly. A single allocation just like global variable does is unthinkable.

### The Way Running Stack Works

Actually, running stack is a segment of memory allocation. It is consisted of stack frames. Each stack frame is correspondent to a function call. The data in stack frame includes arguments, some controlling information, local variables' values and some temporary data. When there is a function called, there is a new stack frame pushed into running stack. After a function returned, its relevant stack frame is popped [1].

When a function is called, it could randomly access to the data of the stack frame it related to, which is on top of the running stack.

When a function is going to call another function, it should give arguments for the function that would be called. This procedure is accomplished by pushing arguments into running stack before calling. Because the current stack frame's allocation could be accessed to by calling function and called function, the calling function's values (arguments) could be passed to the called function's local variables (parameters). That's right. Throughout a simple push operation, the calling function pushed arguments into running stack. Then, the called function regards the allocation (including just pushed values) as its local variables (parameters). This procedure explains perfectly how arguments can be transferred to parameters. Although a function's parameters and local variables' addresses could not be fixed, their offset correspondent to running stack's top is definite. So, parameters and local variables could be located by running stack's top address.

### The Procedure of a Function Call

Before any push or pop operation, the address of running stack's top must be known. It is also necessary when accessing to parameters or local variables. In VC6.0 (32 bits windows system, x86 instruction assembly, the same below), register esp is used to instruct running stack's top. Its name is Stack pointer, as being called.

Influenced by parameters' type and number, different stack frame has different sizes. So, just one stack pointer is not enough for a function to access to its parameters or local variables. Another register is used to hold stack pointer's position when a function is just been called. This register is ebp. It is frame pointer as being called. Esp changes its value as soon as a push or pop operation happens. So, a relevantly stable base is necessary when accessing to parameters. Ebp is such a kind

of base. By disassembly running code in VC6.0, it could be found out that local variables are accessed to throughout ebp. It should be standardized by x86 system and set by Intel.

By a simple function call, the code below is an example of how running stack works and what passing by value is:

```
int swap(int x,int y){
  int t;
  t = x;
  x = y;
  y = t;
       return 1;
}
void main(){
  int a,b,c;
  a = 1234;
  b = 2345;
  c = swap(a,b);
}
```

Firstly, let us take a look of source code in main function and its disassembling code.

```
18:          int a,b,c;
19:          a = 1234; // when running here, ebp=124500. it is main function's base also.
004012B8     mov     dword ptr [ebp-4],4D2h //to set a value to variable a( its address is 1244996)
20:          b = 2345;
004012BF     mov     dword ptr [ebp-8],929h //to set a value to variable b(address is 1244992)
21:          c = swap(a,b);
004012C6     mov          eax,dword ptr [ebp-8]
004012C9     push         eax       //push argument b(address is 1244908)
004012CA     mov          ecx,dword ptr [ebp-4]
004012CD     push         ecx       //push argument a(address is 1244904)
004012CE     call         @ILT+0(swap) (00401005) //call swap function
004012D3     add          esp,8
004012D6     mov          dword ptr [ebp-0Ch],eax //returned value stored in c
```

When running in called function:

```
9:       int swap(int x,int y){
00401250     push         ebp      //store calling function's ebp
00401251     mov          ebp,esp //set stack top to current function's base, ebp=1244896
……          //some irrelevant disassembling code is omitted.
10:          int t;
11:          t = x;
00401268     mov          eax,dword ptr [ebp+8] //x's address is ebp+8, that is 1244904
0040126B     mov          dword ptr [ebp-4],eax
12:          x = y;
0040126E     mov          ecx,dword ptr [ebp+0Ch] //y's address is 1244908
00401271     mov          dword ptr [ebp+8],ecx
13:          y = t;
00401274     mov          edx,dword ptr [ebp-4]
00401277     mov          dword ptr [ebp+0Ch],edx
14:          return 1;
0040127A     mov          eax,1      //store returned value 1 in register eax before return
15:      }
```

Throughout comparing source code of main function and swap function to their disassembling code, it could be found out how an argument is transferred to a parameter when a function call

happens. When function swap(a,b) in main function is called, both values of argument a and b are copied to running stack. However, the allocation that the copies occupied are the very same allocation that parameter x and y have. So, push operation takes a part of transferor here.

Before the called function finished its work, it stores a return value into register eax. This situation always happens whenever a return type is not void. The calling function gets return value by reading register eax as soon as the called function returned.

When an object is passed from calling function to called function, its situation is the same as a basic value is passed. Running stack is also needed. The only difference is that it needs call copy constructor at the same time. In the running stack between calling function and called function, there is an area that could be accessed to by both functions. Calling function write in as arguments before a function call happened and called function read out as parameters after the same function call. So, the only thing and necessary thing is to create passing-by-value objects in the correspondent memory segment of running stack. Objects could be transferred from calling function to called function automatically. The essence of above procedure is to create a brand new object in running stack by calling a copy constructor [5].

## Teaching Method

In C++ programming, function call appears frequently. Data exchange between functions is unavoidable. It tangles learners almost all the time in their studying. All learners want to make it clear. But it is always a perplexing issue. The above example's logic is simple and clear. Even most beginners could understand it. And people with little C++ skills could implement it easily. Its correctness could be tested on any personal computers. In the process of teaching, students could have a direct recognition on function's functions, usages and its data transferring, only if they are introduced to the way function call works from the perspective of assembly and the way that hardware works.

Actually, there are still other questions such as why return value is stored in register eax instead of ebx or ecx or other registers. The author here thinks that it is just a problem of habit. Just think about that function's passing by value. Why does it use running stack other than registers? By the way, the sequence of passing by value is from back to front. It is need to know that all these possibilities could work well if the very first designer wants to do so when he made his complier. But to beginners, it could be perplexing with for a long time. Only if they understood that theory is concluded from practice and is used to instruct practice, they could go farther in computer science.

## Acknowledgement

This article is sponsored by Guangdong University Characteristic Specialty Construction Foundation. Thanks for its supporting for projects of Software Engineering.

## References

[1] Li Zhen, Yuan Dong, and Jiangzhou He, "C++ programming language (the 4th edition)," Tsinghua university publishing house, pp. 529–551, July 2010.

[2] Xiaohon Su, Zhigang Sun, and Huipeng Chen, "C programming language tutorial for universities (the 3rd edition)," Publishing House of Electronics Industry, pp.68–83, May 2012.

[3] Richard Johnsonbaugh and martin Kalin, "C++ Object Oriented Programming (the 2nd edition)," Tsinghua university publishing house, pp. 271–350, August 2005.

[4] D. S. Malik, "C++ Programming Program Design Including Data Structures," Publishing House of Electronics Industry, pp. 710–741, June 2003.

[5] Jiao Geng, Jing Lee, "Comparision of Passing By Value between C/ C++ and JAVA", South North Bridge, pp.151-152, August 2010.