Loop invariant

A loop invariant is a statement that is true across multiple iterations of a loop.

The appropriate loop invariant dependents on the property you want to show that the program has. It doesn't make sense to say: here is a program, what should be the loop invariant.

The most common properties that are used are **eventual termination** (i.e. the program will terminate in finite time) and **correctness of the output** (i.e. when the program terminate, it outputs the value of the associated function on the given input).

Finding a good loop invariant is similar to finding a mathematical proof of a theorem.

Loop Invariants

It is hard to keep track of what is happening with loops. Loops which don't terminate or terminate without achieving their goal behavior is a common problem in computer programming. Loop invariants help. A loop invariant is a formal statement about the relationship between variables in your program which holds true just before the loop is ever run (**establishing the invariant**) and is true again at the bottom of the loop, each time through the loop (**maintaining the invariant**).

Here is the general pattern of the use of Loop Invariants in your code:

```
   ...
// the Loop Invariant must be true here
while ( TEST CONDITION )
{
      // top of the loop
      ...
      // bottom of the loop
      // the Loop Invariant must be true here
}
   // Termination + Loop Invariant = Goal
   ...
```

Between the top and bottom of the loop, headway is presumably being made towards reaching the loop's goal. This might disturb (make false) the invariant. The point of Loop Invariants is the promise that the invariant will be restored before repeating the loop body each time.

A well-chosen loop invariant is useful both **designing**, **testing**, and **modifying** code. It also serves as **documentation** and can be the foundation of a **correctness proof**.

Loop invariant definition

A loop invariant is a statement about program variables that is true before and after each iteration of a loop.

A good loop invariant should satisfy three properties:

1. **Initialization:** The loop invariant must be true before the first execution of the loop.
2. **Maintenance:** If the invariant is true before an iteration of the loop, it should be true also after the iteration.
3. **Termination**: When the loop is terminated the invariant should tell us something useful, something that helps us understand the algorithm.

Let’s start with a simple piece of code.

```
Algorithm sum(n)
    Input: a non-negative integer n
    Output: the sum 1 + 2 + … + n
    sum ← 0
    i ← 1
    while i ≤ n
        // Invariant: sum = 1 + 2 + … + (i - 1)
        sum ← sum + i
        i ← i + 1
    return sum
```

In this example:

1. The loop invariant holds initially since `sum = 0` and `i = 1` at this point. (The empty sum is zero.)
2. Assuming the invariant holds before the `i`th iteration, it will be true also after this iteration since the loop adds `i` to the sum, and increments `i` by one.
3. When the loop is just about to terminate, the invariant states that `sum = 1 + 2 + … + n`, just what’s needed for the algorithm to be correct.

In fact, the three steps above constitute an induction proof, which shows that `sum = 1 + 2 + … + n` when the program leaves the loop.

Notes on Quantifiers

Forms of Quantifier Expressions

Quantifiers are boolean-valued expressions that evaluate a quantified sub- expression multiple times. The number of times the evaluation happens depends on the kind of quantification, which upcoming examples illustrate.

The value of a quantifier expression is true if *all* sub-expression evaluations are true.

Quantifier Evaluation

One can think of quantifiers as a form of programming loop. The important difference between a quantifier and a loop is that the quantifier only produces a single boolean value. In contrast, a program loop typically does not produce a value itself. Rather, the loop executes a body of statements multiple times, with the statements producing value(s) stored in persistent variable(s).

Here's a side-by-side example that illustrates the difference between a `forall` expression and a `for` loop. The idea in both examples is to check that there's no 0 value in a list of integers. The quantifier version is

```
forall (i in l) i != 0
```

The loop version is

```
bool result = true;
for (i = 0; i < length(l); i++) {
    if (l[i] == 0) {
        result = false;
        break;
    }
```

Important features of a structured program.

A structured program uses three types of program constructs i.e. selection, sequence and iteration. Structured programs avoid unstructured control flows by restricting the use of GOTO statements. A structured program consists of a well partitioned set of modules. Structured programming uses single entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, the structured programming principle emphasizes designing neat control structures for programs.

Important advantages of structured programming.

Structured programs are easier to read and understand. Structured programs are easier to maintain. They require less effort and time for development. They are amenable to easier debugging and usually fewer errors are made in the course of writing such programs.

Modular Approach in Programming

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of **modular programming** approached.
- Each sub-module contains something necessary to execute only one aspect of the desired functionality.
- Modular programming emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.

Points which should be taken care of prior to modular program development:

1. Limitations of each and every module should be decided.
2. In which way a program is to be partitioned into different modules.
3. Communication among different modules of the code for proper execution of the entire program.

Advantages of Using Modular Programming Approach –

1. **Ease of Use :**This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability :**It allows the user to reuse the functionality with a different interface without typing the whole program again.
3. **Ease of Maintenance :** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

Part A - Introduction

Modular Programming

Partition source code into modular solution
Compile modules on Linux and Windows platforms

"Decide which modules you want; partition the program so that data is hidden within modules" (Stroustrup, 1997)

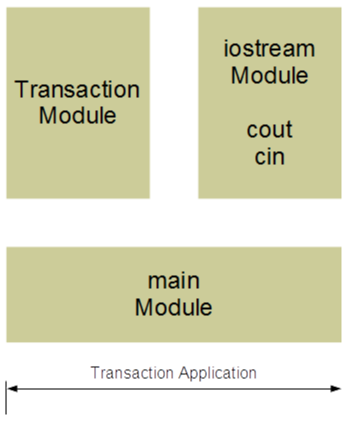
A modular design consists of a set of *modules*, which are developed and tested separately. Modular programming implements modular designs and is supported by both procedural and object-oriented languages. The C programming language supports modular design through library modules composed of functions. The `stdio` module provides input and output support, while hiding its implementation details; typically, the implementation for `scanf()` and `printf()` ships in binary form with the compiler. The `stdio.h` header file provides the interface, which is all that we need to complete our source code.

This chapter describes how to create a module in an object-oriented languages using C++, how to compile the source code for each module separately and how to link the compiled code into a single executable binary. The chapter concludes with an example of a unit test on a module.

MODULES

A well-designed module is a highly cohesive unit that couples loosely to other modules. The module addresses one aspect of the programming solution and hides as much detail as practically possible. A compiler translates the module's source code independently of the source code for other modules into its own unit of binary code.

Consider the schematic of the Transaction application shown below. The `main` module accesses the `Transaction`module. The `Transaction` module accesses the `iostream` module. The `Transaction` module defines the transaction functions used by the application. The `iostream` module defines the `cout` and `cin` objects used by the application.



To translate the source code of any module the compiler only needs certain external information. This information includes the names used within the module but defined outside the module. To enable this in C++, we store the source code for each module in two separate files:

- a header file - defines the class and declares the function prototypes
- an implementation file - defines the functions and contains all of the logic

The file extension `.h` (or `.hpp`) identifies the header file. The file extension `.cpp` identifies the implementation file.

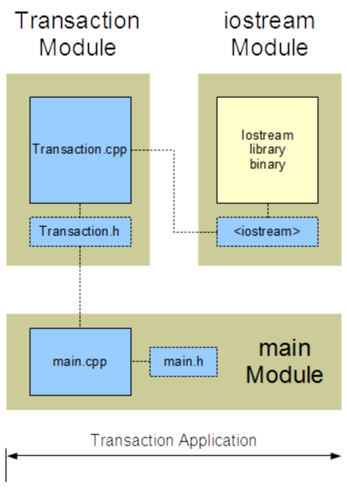
Note, however, that the names of the header files for the standard C++ libraries do not include a file extension (consider for example, the `<iostream>` header file for the `cout` and `cin` objects).

Example

The implementation file for the `main` module includes the header files for itself (`main.h`) and the `Transaction`module (`Transaction.h`). The `main.h` file contains definitions specific to the `main` module and the `Transaction.h` file contains definitions specific to the `Transaction` module.

The implementation file for the `Transaction` module includes the header files for itself (`Transaction.h`) and the`iostream` module. The `Transaction.h` file contains definitions specific to the `Transaction` module and the`iostream` file contains definitions specific to the `iostream` module.

An implementation file can include several header files but DOES NOT include any other implementation file. Note the absence of any direct connections between the implementation files.



We compile each implementation (`*.cpp`) file separately and only once. We do not compile header (`*.h`) files.

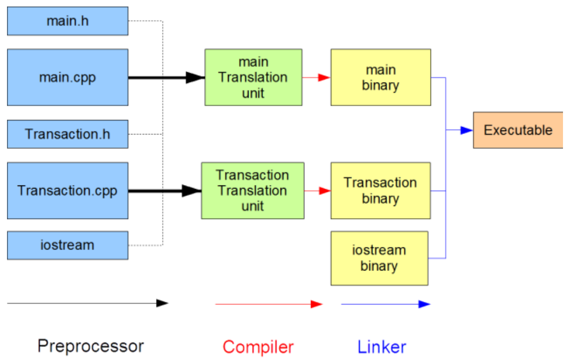
main.cpp
Transaction.cpp

A compiled version of `iostream`'s implementation file is part of the system library.

STAGES OF COMPILATION

Comprehensive compilation consists of three independent but sequential stages (as shown in the figure below):

1. Preprocessor - interprets all directives creating a single translation unit for the compiler - (inserts the contents of all `#include` header files), (substitutes all`#define` macros)
2. Compiler - compiles each translation unit separately and creates a corresponding binary version
3. Linker - assembles the various binary units along with the system binaries to create one complete executable binary



A MODULAR EXAMPLE

Consider a trivial accounting application that accepts journal transactions from the standard input device and displays them on the standard output device. For presentation simplicity, the application does not perform any intermediate calculation.

The application design consists of two modules:

- `Main` - supervises the processing of each transaction
- `Transaction` - defines the input and output logic for a single transaction

Transaction Module

The transaction module defines a structure and functions for a single transaction

- `Transaction` - holds the information for a single transaction in memory

The related functions are global functions

- `enter()` - accepts transaction data from the standard input device
- `display()` - displays transaction data on the standard output device

Transaction.h

The header file for our `Transaction` module defines our Transaction type and declares the prototypes for our two functions:

```
// Modular Example
// Transaction.h

struct Transaction {
     int acct;      //
account number
     char type;     //
credit 'c' debit 'd'
     double amount; //
transaction amount
 };

 void enter(struct
Transaction* tr);
 void display(const struct
Transaction* tr);
```

Note the UML naming convention and the extension on the name of the header file.

Transaction.cpp

The implementation file for our `Transaction` module defines our two functions. This file includes the system header file for access to the `cout` and `cin` objects and the header file for access to the Transaction type.

```
 // Modular Example
 // Transaction.cpp

 #include <iostream> // for
cout, cin
 #include "Transaction.h"
// for Transaction
 using namespace std;

 // prompts for and accepts
Transaction data
 //
 void enter(struct
Transaction* tr) {

     cout << "Enter the
account number : ";
     cin  >> tr->acct;
     cout << "Enter the
account type (d debit, c
credit) : ";
     cin  >> tr->type;
     cout << "Enter the
account amount : ";
     cin  >> tr->amount;
 }

 // displays Transaction
data
 //
 void display(const struct
Transaction* tr) {

     cout << "Account " <<
tr->acct;
     cout << ((tr->type ==
'd') ? " Debit $" : "
Credit $") << tr->amount;
     cout << endl;
 }
```

Note the `.cpp` extension on the name of this implementation file

Main Module

The main module defines a `Transaction` object and accepts input and displays data for each of three transactions.

main.h

The header file for our `Main` module `#define`s the number of transactions:

```
 // Modular Example
 // main.h

 #define NO_TRANSACTIONS 3
```

main.cpp

The implementation file for our `Main` module defines the `main()` function.
We `#include` the header file to provide the definition of the Transaction type:

```
 // Modular Example
 // main.cpp

 #include "main.h"
 #include "Transaction.h"

 int main() {
     int i;
     struct Transaction tr;
     for (i = 0; i <
NO_TRANSACTIONS; i++) {
         enter(&tr);
         display(&tr);
     }
 }
```

Command Line Compilation

Linux

To compile our application on a Linux platform at the command line, we enter the following

```
 g++ -o accounting main.cpp
Transaction.cpp
```

The `-o` option identifies the name of the executable. The names of the two implementation files complete the command.

To run the executable, we enter

```
 accounting
```

Legacy Linux

To compile an application that includes a C++11 feature on a legacy Linux installation, we may need to specify the standard option. For example, to access C++11 features on the GCC 4.6 installation on our matrix cluster, we write

```
 g++ -o accounting
-std=c++0x main.cpp
Transaction.cpp
```

Options for versions 4.7 and later include `c++11` and `gnu++11`.

Visual Studio

To compile our application at the command-line on a Windows platform using the Visual Studio compiler, we enter the command (To open the Visual Studio command prompt window, we press `Start > All Programs` and search for the prompt in the `Visual Studio Tools` sub-directory.)

```
 cl /Fe accounting main.cpp
Transaction.cpp
```

The `/Fe` option identifies the name of the executable. The names of the two implementation files follow this option.

To run the executable, we enter

```
accounting
```

IDE Compilation

Integrated Development Environments (IDEs) are software development applications that integrate features for coding, compiling, testing and debugging source code in different languages. The IDE used in this course is Microsoft's Visual Studio.

Build and Execute

The following steps build and execute a modular application in Visual Studio 2013 or newer:

- Start Visual Studio
- Select New Project
- Select Visual C++ -> Win32 -> Console Application
- Enter Transaction Example as the Project Name | Select OK
- Press Next
- Check Empty Project | Press Finish
- Select Project -> Add New Item
- Select Header .h file | Enter Transaction as File Name | Press OK
- Select Project -> Add New Item
- Select Implementation .cpp file | Enter Transaction as File Name | Press OK
- Select Project -> Add New Item
- Select Header .h file | Enter main as File Name | Press OK
- Select Project -> Add New Item
- Select Implementation .cpp file | Enter main as File Name | Press OK
- Select Build | Build Solution
- Select Debug | Start without Debugging

The input prompts and the results of execution appear in a Visual Studio command prompt window.

Scope rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.

- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>


int main () {


  /* local variable declaration */

  int a, b;

  int c;


  /* actual initialization */

  a = 10;

  b = 20;

  c = a + b;


  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);


  return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>


/* global variable declaration */

int g;


int main () {


  /* local variable declaration */

  int a, b;


  /* actual initialization */

  a = 10;

  b = 20;

  g = a + b;


  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;

}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example –

```
#include <stdio.h>


/* global variable declaration */

int g = 20;


int main () {


  /* local variable declaration */

  int g = 10;


  printf ("value of g = %d\n",  g);


  return 0;

}
```

When the above code is compiled and executed, it produces the following result –

```
value of g = 10
```

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example –

```
#include <stdio.h>


/* global variable declaration */

int a = 20;


int main () {


  /* local variable declaration in main function */

  int a = 10;

  int b = 20;

  int c = 0;


  printf ("value of a in main() = %d\n",  a);

  c = sum( a, b);

  printf ("value of c in main() = %d\n",  c);


  return 0;

}


/* function to add two integers */

int sum(int a, int b) {


   printf ("value of a in sum() = %d\n",  a);

   printf ("value of b in sum() = %d\n",  b);

   return a + b;

}
```

When the above code is compiled and executed, it produces the following result −

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

Parameter Passing in C

When a function gets executed in the program, the execution control is transferred from calling-function to called function and executes function definition, and finally comes back to the calling function. When the execution control is transferred from calling-function to called-function it may carry one or number of data values. These data values are called as **parameters**.

Parameters are the data values that are passed from calling function to called function.

In C, there are two types of parameters and they are as follows...

- **Actual Parameters**
- **Formal Parameters**

The **actual parameters** are the parameters that are specified in calling function. The **formal parameters** are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

- **Call by Value**
- **Call by Reference**

Call by Value

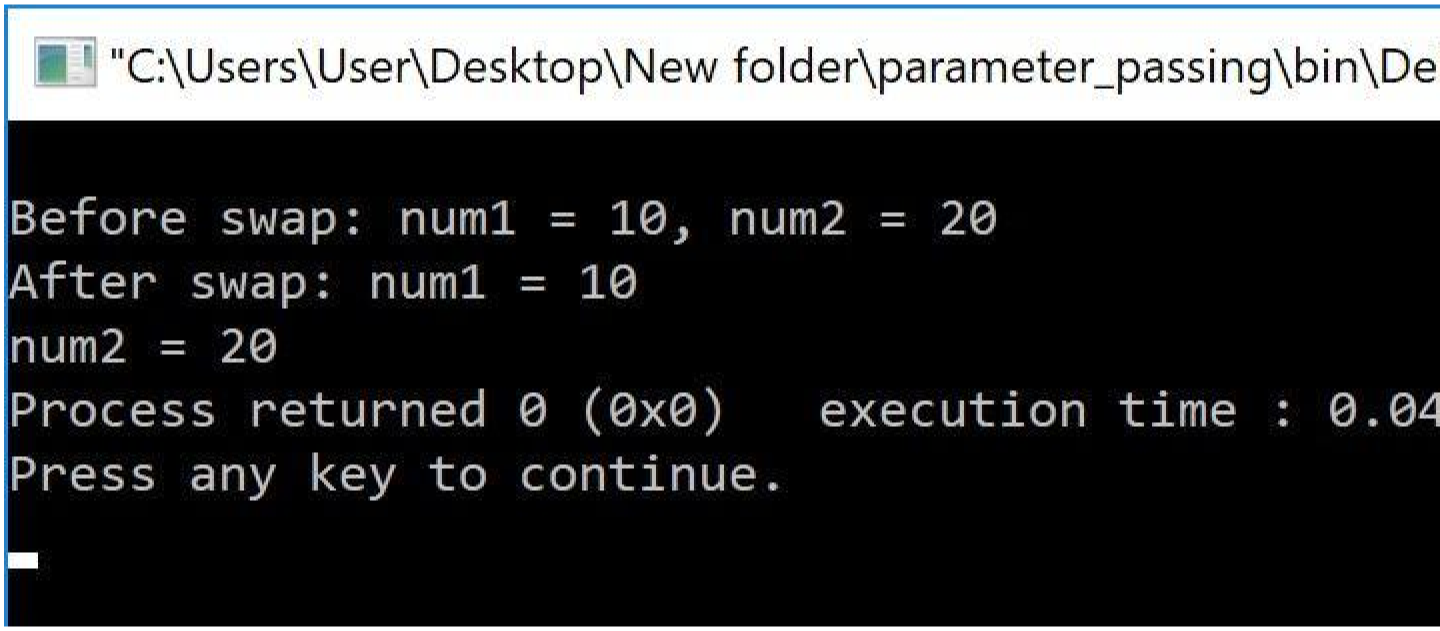
In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. **The changes made on the formal parameters does not effect the values of actual parameters**. That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example consider the following program...

Example Program

```
#include<stdio.h>

#include<conio.h>


void main(){

   int num1, num2 ;

   void swap(int,int) ; // function declaration

   clrscr() ;

   num1 = 10 ;

   num2 = 20 ;


   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;


   swap(num1, num2) ; // calling function
```

```
    printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
    getch() ;
}
void swap(int a, int b)  // called function
{
   int temp ;
   temp = a ;
   a = b ;
   b = temp ;
}
```

Output:



```
"C:\Users\User\Desktop\New folder\parameter_passing\bin\De

Before swap: num1 = 10, num2 = 20
After swap: num1 = 10
num2 = 20
Process returned 0 (0x0)   execution time : 0.04
Press any key to continue.
```

In the above example program, the variables **num1** and **num2** are called actual parameters and the variables **a** and **b** are called formal parameters. The value of **num1** is copied into **a** and the value of num2 is copied into **b**. The changes made on variables **a** and **b** does not effect the values of **num1** and **num2**.

Call by Reference

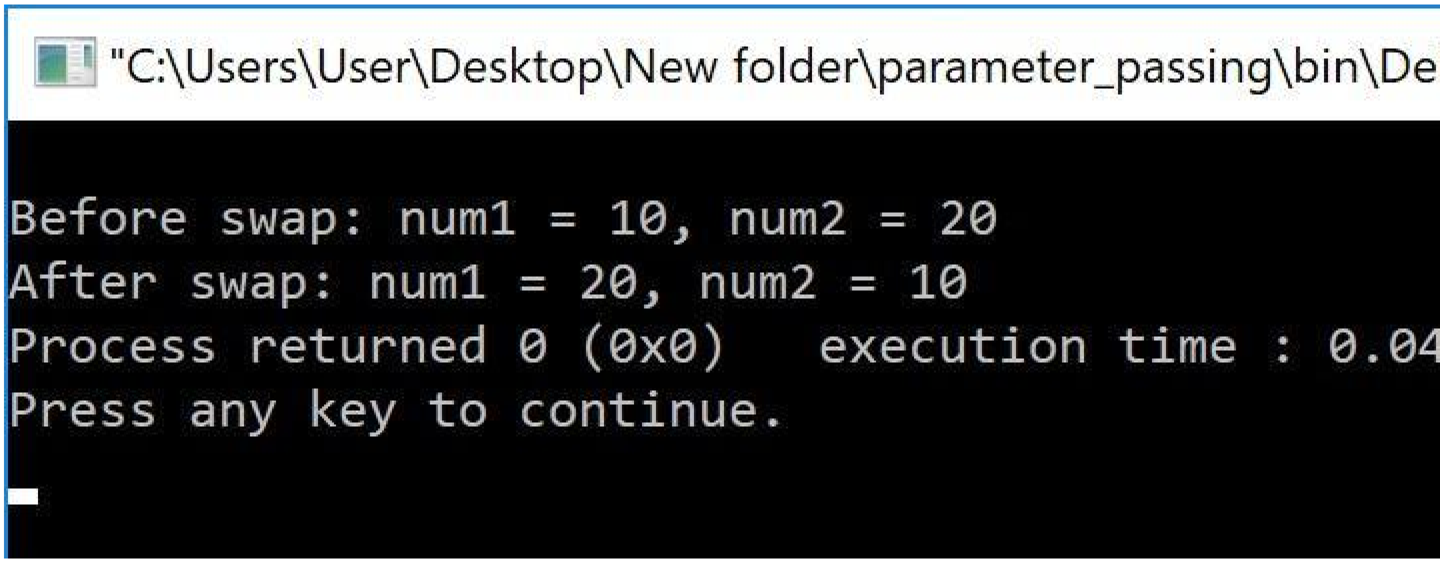
In **Call by Reference** parameter passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function. In this method of parameter passing, the formal parameters must be **pointer** variables.

That means in call by reference parameter passing method, the address of the actual parameters is passed to the called function and is recieved by the formal parameters (pointers). Whenever we use these formal parameters in called function, they directly access the memory locations of actual parameters. So **the changes made on the formal parameters effects the values of actual parameters**. For example consider the following program...

Example Program

```
#include<stdio.h>

#include<conio.h>


void main(){

   int num1, num2 ;

   void swap(int *,int *) ; // function declaration

   clrscr() ;

   num1 = 10 ;

   num2 = 20 ;


   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;

   swap(&num1, &num2) ; // calling function


   printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);

   getch() ;

}

void swap(int *a, int *b)  // called function

{

  int temp ;

  temp = *a ;

  *a = *b ;

  *b = temp ;

}
```

Output:



```
"C:\Users\User\Desktop\New folder\parameter_passing\bin\De

Before swap: num1 = 10, num2 = 20
After swap: num1 = 20, num2 = 10
Process returned 0 (0x0)   execution time : 0.04
Press any key to continue.
```

In the above example program, the addresses of variables **num1** and **num2** are copied to pointer variables **a** and **b**. The changes made on the pointer variables **a** and **b** in called function effects the values of actual parameters **num1** and **num2** in calling function.