

Functions

What are functions good for?

- Help **avoid** code **repetition**
- parameterization of functional blocks, encapsulation of code
- **Scalability**: build large and robust software systems
- modular, top-down design, unit testing of code modules
- Portability and **re-use**

Terminology

- **Subroutine/Procedure**: A call/return block of code that **does not return a value**, but may take arguments and modify their value.
- **Function**: A call/return block of code that **returns a value**. Functions may take arguments and modify their value.
- **Method**: A function or subroutine that is **part of a class**.
- **Argument**: An **expression used when calling a function** or subroutine.
- **Parameter**: The **identifier declared in** a function or subroutine **definition** that will connect to the corresponding argument in a function call.

Parameter Passing

- **Pass by value**: the computer evaluates the argument expression and **places a copy** of its value into the memory address referenced by the parameter. (C, C++, Java)
 - **Pass by reference**: the computer evaluates the argument expression and **places a reference** to the result into the memory address referenced by the parameters. (C++)
 - **Pass by value-result**: the computer evaluates the argument expression to a memory location and then copies its value to the memory address referenced by the parameter. When the function returns, the computer copies the value of the parameter back to the location holding the argument.
-
- Show Swap.java, ask the output for swap(int, int) [won't swap a and b]
 - ask the output for swap(Integer, Integer) [won't swap a and b]
 - ask the output for resetVal (Item item) [reset the item val]
 - Why? [Java is pass by value]

```

public class Swap {
    public static void swap (int a, int b) {
        int t = a;
        a = b;
        b = a;
    }

    // If change the reference to refer to another location,
    // any changes to the reference are not reflected
    // back in the main function
    public static void swap (Integer a, Integer b) {
        Integer t = a;
        a = b;
        b = a;
    }

    // If change the member of an object referred by the reference
    // the changes are reflected back, as even it's a copy of
    // the object's reference, it refers to the same object
    public static void resetVal (Item item) {
        item.val = 0;
    }

    public static void main (String[] args) {
        int a = 5;
        int b = 10;
        System.out.println("Before swapping a = " + a + " b = " + b);
        swap(a, b);
        System.out.println("After swapping a = " + a + " b = " + b);

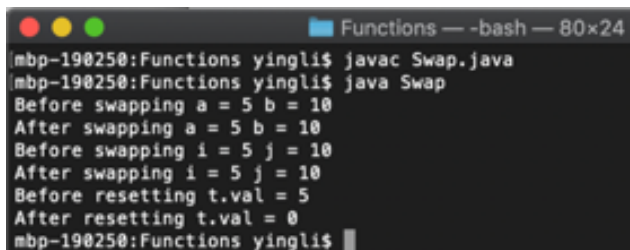
        Integer i = 5;
        Integer j = 10;
        System.out.println("Before swapping i = " + i + " j = " + j);
        swap(i, j);
        System.out.println("After swapping i = " + i + " j = " + j);

        Item t = new Item(5);
        System.out.println("Before resetting t.val = " + t.val);
        resetVal(t);
        System.out.println("After resetting t.val = " + t.val);
    }
}

class Item {
    int val;

    public Item (int v) {
        val = v;
    }
}

```



```

mbp-190250:Functions yingli$ javac Swap.java
mbp-190250:Functions yingli$ java Swap
Before swapping a = 5 b = 10
After swapping a = 5 b = 10
Before swapping i = 5 j = 10
After swapping i = 5 j = 10
Before resetting t.val = 5
After resetting t.val = 0
mbp-190250:Functions yingli$

```

- Now we know that Java is pass-by-value. How about C?
- Using swap function is a good way to check whether a language uses pass-by-value or pass-by-reference. The change happened inside the swap function cannot reflect to the calling function, the language is pass-by-value.
- Show swap.c, and ask the output and the parameter passing of C. [pass by value]

```
#include <stdio.h>

int swap_f (int x, int y) {
    int t;
    t = x;
    x = y;
    y = t;

    return 0;
}

int main () {
    int a = 3, b = 5;
    printf("Before swap a = %d, b = %d \n", a, b);
    swap_f(a, b);
    printf("After swap a = %d, b = %d \n", a, b);

    return 0;
}
```

- Although C uses passing by value, we can use pointers to make it possible to change the values of variables in the calling function. This is getting to what we will mean by pass-by-reference.

```
#include <stdio.h>

int swap (int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;

    return 0;
}

int main () {
    int a = 3, b = 5;
    printf("Before swap a = %d, b = %d \n", a, b);
    swap(&a, &b);
    printf("After swap a = %d, b = %d \n", a, b);

    return 0;
}
```

However, passing pointers as values is not equivalent to pass-by-reference. We can tell the difference after knowing what is pass-by-reference in C++.

- C++ allows passing by reference. Show [swap.cc](#).

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

int swap (int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;

    return 0;
}

int main () {
    int a = 3, b = 5;
    cout << "Before swap a = " << a << ", b = " << b << "\n";
    swap(a, b);
    cout << "After swap a = " << a << ", b = " << b << "\n";

    return 0;
}
```

- The above swap function passes arguments as references. Using pass-by-reference, the only thing the swap function can do is update the values in the memory referred by the references. It's syntactically incorrect to change the reference values inside the swap function. This is to say it's invalid to do anything like `&a = &b` in the above swap function.

- Another example demonstrating that passing by reference is not equivalent to passing pointer by value.
- If a function is pass-by-reference, the intent is clearer – we are meant to change the value at the given address. There is no way to use that variable to point to a different address.
- If a function is pass-by-value and the value is a pointer, it's possible to reassign another address to the pointer, and the changes on the value referred by the new address won't impact the value stored in the original address.
- Note that this doesn't do anything untoward to the variable declared in the calling code (so it isn't a sneaky way to change something). It is just yet another way that C allows you to write code that is not super readable.

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

// difference between pass by reference and
// passing a pointer by value.
void f( int &a ) {
    a = 3;
}

void g( int *a ) {
    *a = 3;
    int *y = (int*)malloc( sizeof(int) );
    *y = 5;
    a = y;
    *a = 7;
}

int main () {
    int a = 1;
    cout << "Before f, a = " << a << endl;
    f( a );
    cout << "After f, a = " << a << endl;
    cout << "Before g, a = " << a << endl;
    g( &a );
    cout << "After g, a = " << a << endl;

    return 0;
}
```

Output:

```
Before f, a = 1
After f, a = 3
Before g, a = 3
After g, a = 3
```

- Ada is a language that *may* use pass-by-value-result. Every parameter in an Ada function is designated as “in”, “out”, or “in out”. In parameters are read-only (cannot be changed by the code in the function). Out parameters are write-only (the code cannot read a value before it has written one). In Out parameters can be both read to and written from. The Ada language specification does not mandate whether the compiler handles this by implementing pass-by-reference or pass-by-value-result.

- Run these online: https://www.tutorialspoint.com/compile_ada_online.php

- Example 1: Swap

```
-- Ada scope demonstration program.
-- Ada has nested block and function scope and automatic for loop scope
with Text_IO; use Text_IO;      -- This gets the IO facility.
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
```

```
procedure demo_swap is          -- Main, but no special name is
needed.
```

```
    pt1 : Integer := 4;
    pt2 : Integer := 5;
```

```
    procedure swap( a : in out Integer;
                    b : in out Integer ) is
```

```
        c : Integer;
    begin
        c := b;
        b := a;
        a := c;
    end swap;
```

```
begin
    Put( "Before calling swap pt1 is " );
    Put( pt1 );
    Put( ", pt2 is " );
    Put( pt2 );
    Put_Line( "" );
    swap( pt1, pt2 );

    Put( "Before calling swap pt1 is " );
    Put( pt1 );
    Put( ", pt2 is " );
```

```
Put( pt2 );  
Put_Line( "" );  
swap( pt1, pt2 );
```

```
end demo_swap;
```

- Example 2. This demonstrates pass-by-value-result.

```
-- Ada parameter-passing demonstration program.  
-- This shows that the gnatmake 8.1 compiler uses pass-by-value-result  
-- for the in out parameter.  
-- https://stackoverflow.com/questions/37210311/call-by-result-example  
with Gnat.IO; use Gnat.IO;
```

```
procedure passByValueResult is  
  x : Integer;  
  Procedure NonSense (A: in out integer) is  
  begin  
    x := x + 1;  
    A := A + 4;  
    Put(x);  
  end NonSense;  
  
begin  
  x := 0;  
  NonSense (x);  
  Put(" ");  
  Put(x);  
  New_Line;  
end passByValueResult;
```

- Another thing you may want to pay attention to pass-by-reference is the **aliasing**, which means **reference to the same entity by different names**.

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

int open_to_problems (int &a, int &b) {
    a = a + 1;
    b = b + 1;

    return 0;
}

int main () {
    int x = 10, y = 20;

    cout << "Before open_to_problems, x = " << x << ", y = " << y << endl;
    open_to_problems( x, y );
    cout << "After open_to_problems, x = " << x << ", y = " << y << endl;

    cout << "Before open_to_problems, x = " << x << endl;
    open_to_problems( x, x );
    cout << "After open_to_problems, x = " << x << endl;

    return 0;
}
```

Function Call Implementation

- An easy way to implement factorial? [recursion]

```
/**
 * factorial.c
 */

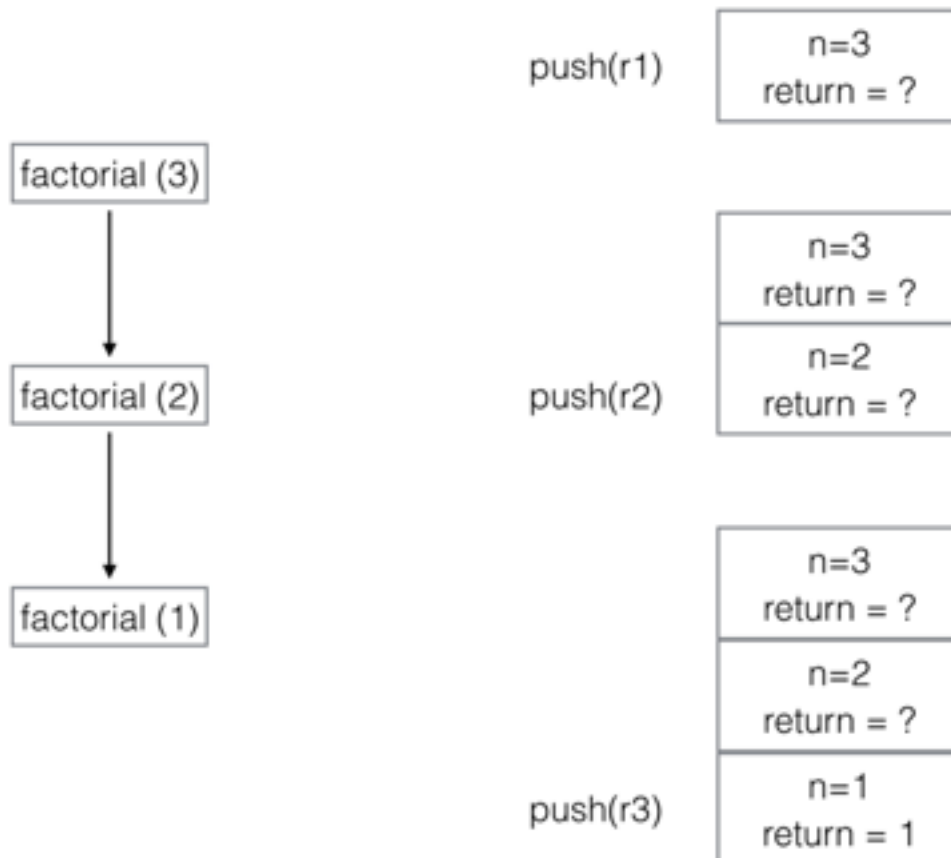
#include <stdio.h>

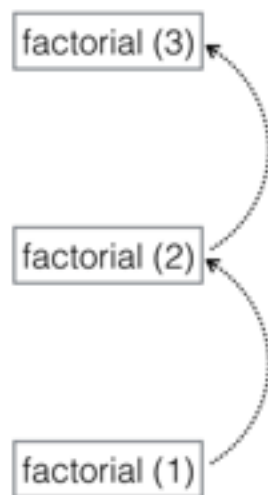
int factorial (int n) {
    if (n < 2)
        return 1;
    else
        return n * factorial (n-1);
}

int main () {
    int n = 3;
    int i = factorial (n);
    printf("factorial(%d) = %d\n", n, i);
    return 0;
}
```

- A key component of a recursive function?
 - The **base case**. Otherwise, the function recurs forever.
 - In the above program, the base case is the if branch.
- How does recursive function work?
 - The factorial function is first called in the main function, and passes value 3 to it.
 - When executing the factorial function, factorial function is called again, and passed value 2 to it.
 - When calculating factorial 2, the function is called for the third time to calculate factorial 1.
 - When get the return value of factorial 1, factorial 2 is calculated.
 - After getting the value of factorial 2, factorial 3 is calculated.
 - Then the final value is returned to the main function.
- This process makes sense logically. But **how the computer handles it? How are those active records saved for recursive calls?**
 - Generally, the computer uses the system or program **stack** to hold active records.
 - **Active records, also called stack frames**, need to hold the following types of information.
 - Return address
 - Return value
 - Arguments
 - Parameters
 - Local variables
 - A pointer to the prior activation record
 - A pointer to the function's static context
 - Saved registers

- This graphic explains how the stack holds the active records.
- This stack is a up-side-down stack, and for simplicity, stack frame only holds arguments and return values





pop(r1)

n=3 return = 6

pop(r2)

n=3 return = ?
n=2 return = 2

pop(r3)

n=3 return = ?
n=2 return = ?
n=1 return = 1

Notes from Bruce:

Function calls require the computer to transfer data from the parent context to the function context, then transfer execution control to the function. When the function is done executing, it must transfer data back to the parent context and then transfer execution control. The standard approach to implementing these capabilities is to use an activation record. The parent context builds part of the activation record, the function builds the rest. The function has access to the whole activation record, enabling information transfer between the two contexts.

Languages other than older versions of Fortran—which did not permit recursion—must dynamically create activation records. Generally, the computer uses the system or program stack to hold activation records, and explicit support for them exists in almost all hardware instruction sets, even in RISC machines.

Activation records, also called stack frames, need to hold the following types of information.

- Return address
- Return value
- Arguments
- Parameters
- Local variables
- A pointer to the prior activation record
- A pointer to the function's static context (e.g. class context or base context)
- Saved registers

Which context saves which pieces? It depends partly upon convention, partly upon which context has access to the information when. The return address is the only piece actually required to occur at the level of hardware.

Parent context:

- Push space for the return value
- Push on the arguments
- Push on the local stack frame (activation record) pointer
- Push on the static context reference
- Call the function (pushes on return address)

Function context:

- Set up its own stack frame pointer (all information is relative to this location)
- Push on space for local variables
- Push on any registers the function will modify
- Do its thing
- Copy the return value to its location on the stack
- Restore registers
- Remove local variables
- Return (pops return address)

Parent context:

- Pop the static context
- Restore the stack frame pointer
- Pop the arguments
- Pop and store the return value

Note that responsibility for several of the pieces is based on convention. The function is generally responsible for saving any registers it modifies. In the example above I gave the parent context the responsibility of handling the stack frame, but that also could be treated by the function as a register that needs saving.