2. PARAMETER PASSING

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case, A is called the "caller function" and B is called the "called function or callee function". Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Terminology

Formal Parameter: A variable and its type as they appear in the prototype of the function or method.

Actual Parameter: The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Modes:

IN: Passes info from caller to the callee.

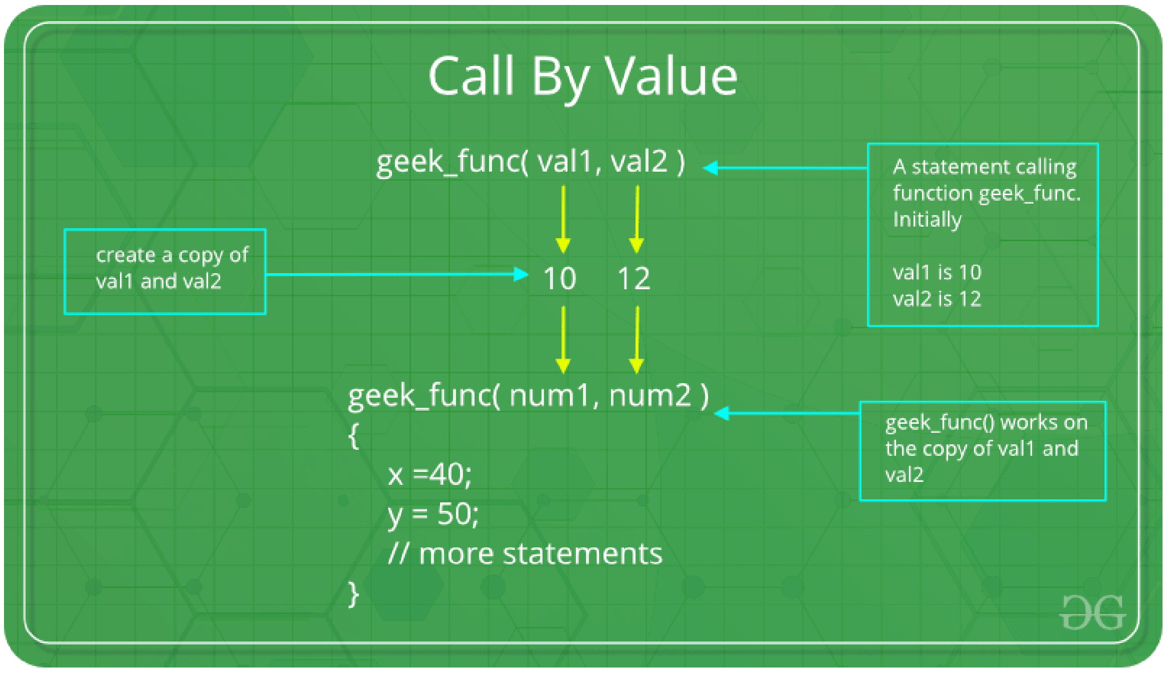
OUT: Callee writes values in the caller.

IN/OUT: The caller tells the callee the value of the variable, which may be updated by the callee.

Important Methods of Parameter Passing are as follows:

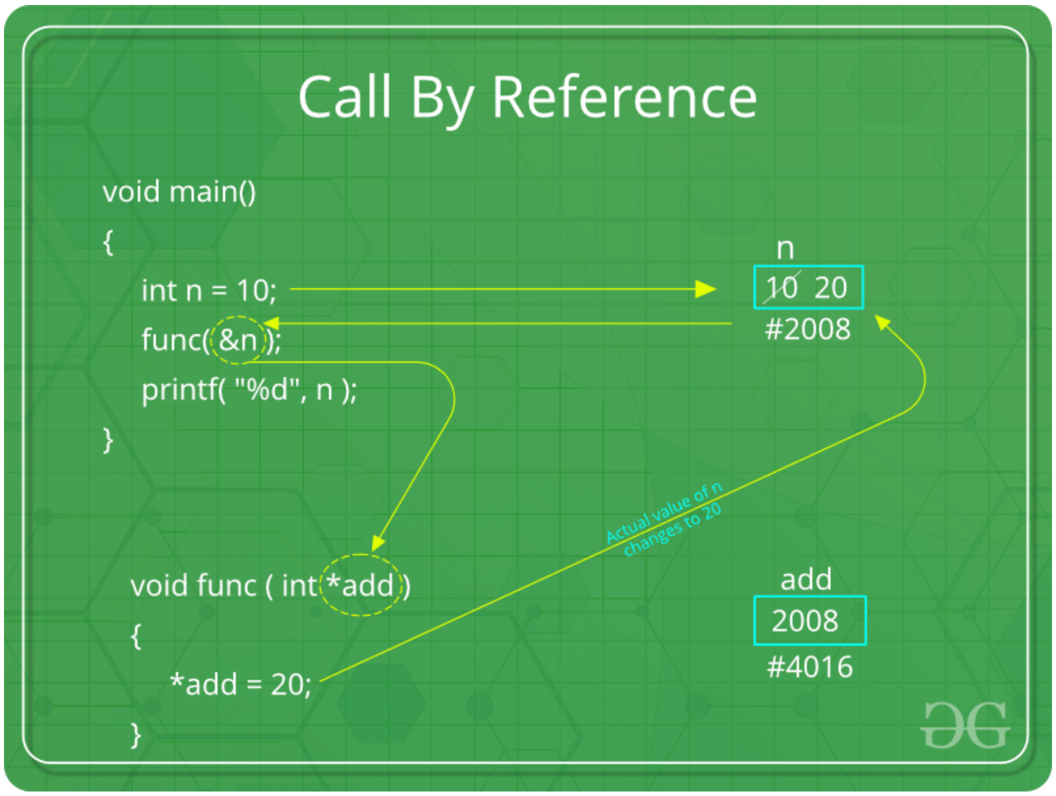
1. Pass by Value

This method uses in-mode semantics. Changes made to formal parameters do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called call by value.


Call By Value
geek_func(val1, val2)
A statement calling function geek_func. Initially
val1 is 10
val2 is 12
create a copy of val1 and val2
10 12
geek_func(num1, num2)
{
x =40;
y = 50;
// more statements
}
geek_func() works on the copy of val1 and val2

2. Pass by reference(aliasing)

This technique uses in/out-mode semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as call by reference. This method is efficient in both time and space.



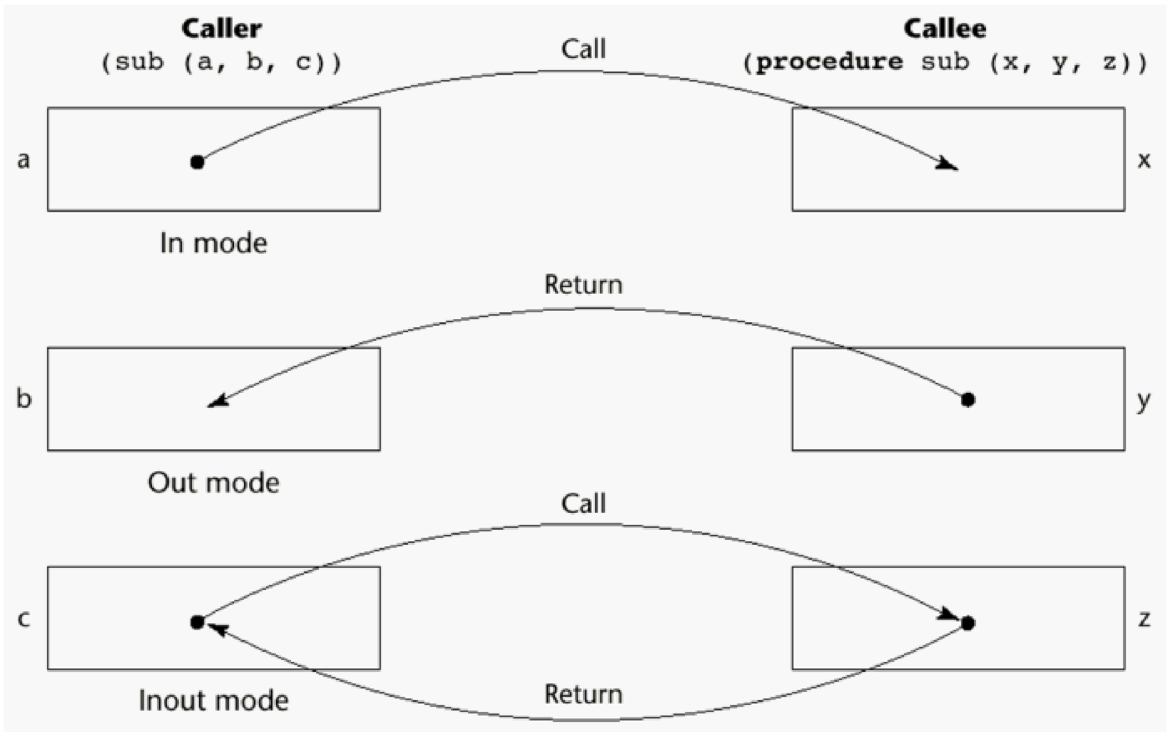
```
void main()
{
    int n = 10;
    func( &n );
    printf( "%d", n );
}

void func ( int *add )
{
    *add = 20;
}
```

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

- **Physically move a path**
- **Move an access path**

Pass-by-Value (In Mode)

• The value of the actual parameter is used to initialize the corresponding formal parameter

– Normally implemented by copying

– Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)

– Disadvantages (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large

parameters)

– Disadvantages (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Result (Out Mode)

• When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller’s actual parameter when control is returned to the caller, by physical move

– Require extra storage location and copy operation

• Potential problem: sub(p1, p1);

whichever formal parameter is copied back

will represent the current value of p1

Pass-by-Value-Result (inout Mode)

• A combination of pass-by-value and pass-byresult

• Sometimes called pass-by-copy

• Formal parameters have local storage

• Disadvantages:

– Those of pass-by-result

– Those of pass-by-value

Pass-by-Reference (Inout Mode)

• Pass an access path

• Also called pass-by-sharing

• Advantage: Passing process is efficient (no copying and no duplicated storage)

• Disadvantages

– Slower accesses (compared to pass-by-value) to formal parameters

– Potentials for unwanted side effects (collisions)

– Unwanted aliases (access broadened)

Pass-by-Name (Inout Mode)

• By textual substitution

• Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment

• Allows flexibility in late binding

Implementing Parameter-Passing Methods

• In most language parameter communication takes place thru the run-time stack

• Pass-by-reference are the simplest to implement; only an address is placed in the stack

• A subtle but fatal error can occur with passby-reference and pass-by-value-result: a formal parameter corresponding to a constant can mistakenly be changed\

Parameter Passing Methods of Major Languages

• **C**

– Pass-by-value

– Pass-by-reference is achieved by using pointers as parameters

• C++

– A special pointer type called reference type for pass-by-reference

• Java

– All parameters are passed are passed by value

– Object parameters are passed by reference

• **Ada**

– Three semantics modes of parameter transmission: in, out, in out; in is the default mode

– Formal parameters declared out can be assigned but not referenced; those declared in can be

• **Fortran 95**

- Parameters can be declared to be in, out, or inout mode

• **C#**

- Default method: pass-by-value

– Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref

• **PHP**: very similar to C#

• Perl: all actual parameters are implicitly placed in a predefined array named @_

• Python and Ruby use pass-by-assignment (all data values are objects)

Type Checking Parameters

• Considered very important for reliability

• FORTRAN 77 and original C: none

• Pascal, FORTRAN 90, Java, and Ada: it is always required

• ANSI C and C++: choice is made by the user

  – Prototypes

• Relatively new languages Perl, JavaScript, and PHP do not require type checking

• In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Parameters

• If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know thedeclared size of that array to build the storage mapping function

Multidimensional Arrays as Parameters: C and C++

• Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

• Disallows writing flexible subprograms

• Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

Multidimensional Arrays as Parameters: Ada

• Ada – not a problem

– Constrained arrays – size is part of the array's type

– Unconstrained arrays - declared size is part of the object declaration

Multidimensional Arrays as Parameters: Fortran

• Formal parameter that are arrays have a declaration after the header

– For single-dimension arrays, the subscript is irrelevant

– For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays as Parameters: Java and C#

• Similar to Ada

• Arrays are objects; they are all singledimensioned, but the elements can be arrays

• Each array inherits a named constant (length in Java, Length in C#) that is set to the length of the array when the array object is created

Design Considerations for Parameter Passing

• Two important considerations

– Efficiency

– One-way or two-way data transfer

• But the above considerations are in conflict

– Good programming suggest limited access to variables, which means one-way whenever possible

– But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

• It is sometimes convenient to pass subprogram names as parameters

• Issues:

1. Are parameter types checked?

2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameter Type Checking

• C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked

• FORTRAN 95 type checks

• Ada does not allow subprogram parameters; an alternative is provided via Ada's generic facility

• Java does not allow method names to be passed as parameters

Names: Referencing Environment

• Shallow binding: The environment of the call statement that enacts the passed subprogram

- Most natural for dynamic-scoped languages

• Deep binding: The environment of the definition of the passed subprogram

- Most natural for static-scoped languages

• Ad hoc binding: The environment of the call statement that passed the subprogram

Overloaded Subprograms

• An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment

– Every version of an overloaded subprogram has a unique protocol

• C++, Java, C#, and Ada include predefined overloaded subprograms

• In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)

• Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

• A generic or polymorphic subprogram takes parameters of different types on different activations

• Overloaded subprograms provide ad hoc polymorphism

• A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism

- A cheap compile-time substitute for dynamic binding

• Ada

– Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement

– Generic subprograms are preceded by a generic clause that lists the generic variables, which can be types or other subprograms

• C++

– Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator

– Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names

• Java 5.0

- Differences between generics in Java 5.0 and those of C++ and Ada:

1. Generic parameters in Java 5.0 must be classes

2. Java 5.0 generic methods are instantiated just once as truly generic methods

3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters

4. Wildcard types of generic parameters Generic Subprograms (continued)

• C# 2005

- Supports generic methods that are similar to those of Java 5.0

- One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type

Examples of parametric polymorphism: C++

```
template <class Type>

Type max(Type first, Type second) {

            return first > second ? first : second;

}
```

The above template can be instantiated for any type for which operator > is defined

```
int max (int first, int second) {

           return first > second? first : second;

}
```