

2 INTRODUCTION: Syntax Of Parameters

Call Parameter Types:

2 Copy or Value Parameters

3. Summary Of Relation Of Formal and Actual Parameters

Call Parameter Types:

**4 Example of Copy (value) in and out and in-out Parameters
Reference or Reference InOut or Location Parameter:**

Call Parameters Types:

**5 Reference (Location), and Name.
Call-by-value: Pointer Parameters effects Call by reference (location)**

Problems with Call Types

5 Pass By Value or Copy Inefficiencies

Problems with Call Types

**6 Pass By Result (Copy-out)
Example Different Effects of Global Evaluated inside and Outside A Procedure.
Pass By Reference or Location or Reference
Example Of Optimization Invalidated By Aliasing.**

Problems with Call Types

**7 Pointer Example Of Optimization Invalidated By Aliasing.
Example OK with copy-inout value-result parameters.**

Call By Value Without and With Pointers Swap Example

**8 Problem Swap with Call By Value
With Use Of Copy-Inout Can Achieve Equivalent To Call By Reference
C: All Parameters Are Call by Value or Copy-in With Use Of Pointers Can Achieve
Equivalent To Call By Reference**

9 C:Pointers All Parameters Are Call by Value or Copy-in Array Example

**10 Pass By Name
Value Returning Procedures-Function
Summary**

11, 12 Review Of Pointers In C

CONTENTS

1 Interpretation Of Procedure Parameters

Syntax Of Parameters

A procedure definition consists of a **procedure header** and a **procedure body**.

The **name of the procedure and the formal parameters** are given in the header.

The formal parameters occur again in the body.(see figure 1, pg 2)

The **procedure call** consists of the **procedure name**, and the **actual parameters**. the procedure name is used when the procedure is called. Also part of the procedure call, the **actual parameters, are associated 1-1 with the formal parameters** of that definition 's header. They may be variable names or expressions, or constants (unlike formal parameters).

(A function is distinguished from a procedure in that it returns a value that replaces the procedure call, and is otherwise like a call-by-value procedure.)

[[This association can be by position or by name. The positional association requires the *ith* actual parameter to correspond to the *i*th formal parameter, the name association requires that a name be given for the formal parameter in the header and that name be used to designate the associated actual parameter in the call. Sometimes both association are used in the same procedure, i.e., some parameters are named and some aren't. In the call of that procedure then those parameters given names in the header must be given names in the call, those not so named have positional associations. It is also sometimes desirable to allow an unlimited number of parameters. This can be accomplished by indicating with some special symbol that the last parameter can be in fact be any number of parameters. Alternatively any formal parameter can be associated with an unspecified number of actual parameters if an end symbol is specified in the actual parameter presentation.]]

A procedure call is an abbreviation for the body of the corresponding procedure definition

(With the actual parameters associated in some way with the corresponding formal parameter appearances in the body. This association can take a number of forms. In different languages the same form is designated differently, where different names are used for the same kind of call we have included alternative names in parenthesis.)

The **classical mathematical Procedure interpretation, or call-by-value**, is that, in effect:

1. The a-p's are evaluated
2. These evaluated values replace the corresponding f-p's in the body of P.
3. This instantiated version of P is evaluated and the call is replaced by that evaluation and a returned value (in a function this is provided by a return call)

Notice first of all that this model **does not allow assignments within P to f-p's**. In the early languages, which did not incorporate this concept of a function returning a value, the only way of returning a value was through an a-p. The classical function and strict call-by-value interpretation does not support this use of an a-p to which value is returned. In addition there are some practical problems when this interpretation is modelled. They involve time and space inefficiencies. We give more details on these problems after surveying the many different interpretations of parameters incorporated in modern computer languages.

Call Parameter Types

Copy or Value Parameters: These are the closest models to the mathematical interpretation all have the following characteristics. Before the body of P there are inserted (by the compiler) local declarations for variable with the same name as the f-p (called the **Prefix**), so all f-p occurrences in the remainder of the body become references to that local variable, in the same way they would reference any other local variable. In Java, C and C++, an a-p may be a pointer, in which case a local pointer is declared with the name of the f-p. This actually extends the effect of call-by-value to include the properties of the other call types defined below.

- **value or copy-in parameter:** Add to the Prefix the assignment: f-p = a-p. A value is assigned to an f-p in the procedure but this will have no effect on the associated a-p.

There are different additions to these basic properties which define variations of the copy parameters.

- **result or copy-out parameter:** Before any execution of P a **Suffix** is added to P which provides that after each completed execution of P: the contents of the local variable f-p is made that of the corresponding a-p (this means that there variable can also be assigned a value.

The value of a-p is not assigned to the f-p before entry to P. The first use of an f-p in the procedure must not assume that f-p has been assigned a value, so it must result in the assignment of a value to f-p. (Can be compiler checked)

USER DEFINITION OF Header

```
procedure Funct( fp-1,data type1, call type1; fp-2, data type2,call type2,.., fp-n, ..)
{
    fp-1
    = fp-1
    fp-2 = fp-2 fp-n
}
```

Formal Body



COMPILED DEFINITION OF

```
procedure Funct( fp-1,data type1, call type1; fp-2, data type2,call type2,.., fp-n, ..)
{
    Prefix:For copy -out declare
             For copy in, inout declare & assign ap-j to fp-j
    In this area all occurrences of
    fp-j which are call-by-copy-in, out, inout, are unchanged
    fp-j which are call-by-reference or call-by-name are replaced
    on a call by their corresponding ap-j s, literally for name,
    but address of ap-j for reference.
    Suffix:For copy out, inout assign fp-j to ap-j
}
```

Actual Body

Funct (ap-1, ap-2,,,,,....., ap-n)

ap-j may be an expression involving constants and variables in call by copy-in.

The Call

fp-j = Formal Parameter j ap-j =Actual Parameter j

SUMMARY OF RELATION OF FORMAL AND ACTUAL PARAMETERS

- value-result or copy inout or copy-in copy-out parameter: The value assigned to the f-p local variable in the Prefix, before each execution of P is that of the completely evaluated a-p (a location). The final value assigned to the a-p local variable is assigned to the f-p (a location) after each execution of P. This location may be that computed on exit or that computed on entry this needs to be specified

The Call: F(exp, W, X) (exp can be an expression W and X must be variables)

The Procedure:

F:procedure(V:value (copy-in) integer, R:result (copy-out) real, VR:value-result (copy-inout) real)

V:integer;= ap-1 (exp on call)

R:real (declared but not assigned)

VR:real =ap-3 (X on call)

;		Formal Body	Actual Run Time Body
T :integer	R = V /1	R must be assigned before used	
	T = V+VR	OK anywhere	
	V = V+1	V OK right of "=" (theory(C): not on left-but harmless as implemented)	
	R = 5+T	R must be assigned before exit	
	VR = 5+T-V	VR need not be assigned before exit	
	ap-2 = R	(<u>W</u> on call)	
	ap-3 = VR	(<u>X</u> on call)	
end F			

Copy (value) in and out and in-out Parameters

Reference or Reference InOut or Location Parameter:

The following are characteristics of reference inout parameters: Location(f-p) is made = location(a-p) before each execution of P. Necessarily a-p is restricted to always being the name of the location of a type, albeit possibly a compound name ex B(2) or A(1,2). If a-p is a compound variable like A(I) or A(B(I)) then it must be evaluated before entry to determine the associated location of the name dependent on its index before entry. There are subtypes of reference parameters characterized by certain restrictions on the freedom of the in-out type.

- reference-in parameter: restriction: no assignment of a value to f-p in P is allowed. (This requires a compiler check)
- reference-out parameter: restriction: the value assigned to any variable may not depend on any f-p in the body of P. (This requires a compiler check)

Name or Name InOut or Textual Parameter: f-p is made a literal copy of a-p (without any evaluation) before each execution of P, this includes complex variables and expressions i.e, a-p's A[i], and X + Y, replace the corresponding f-p's in the body of P, not the location of A[i], nor the value at call time of the expression X + Y. On completion any value assigned to the a-p variable in the body will still be there. Again there are restrictions on this general naming parameter.

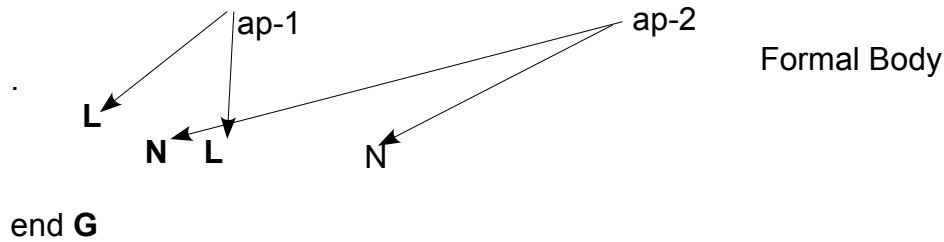
- name-in parameter: restriction: there may not be any assignment of a value to f-p in the procedure body. (This requires a compiler check)
- name-out parameter: restriction: the value assigned to any variable may not depend on any f-p in the procedure body. (This requires a compiler check)

The Call:

G (Y reference, Z name) Y Var and Z Is Var or exp

The Procedure:

procedure G(L:reference(location) real, N:name real)



Actual Body;
 L = ap-1 (=runtime addr of Y With given Call)
 throughout Formal BODY.
 N = ap-2(=Z (literally) With given Call) throughout Formal BODY.]]

Call Parameters are call type: reference (location), and name.

In C all parameters are call by value. However one can use a pointer as a parameter. Its value is the address (location) to which it points. This allows passing in locations using only call by value.

int *pt =9;

The Call: H(pt)

The Procedure:

procedure H (int *Q)

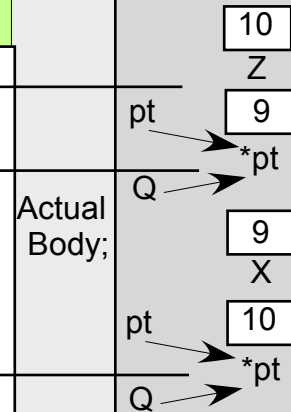
int *Q; Q = ap-1 Thus Q points to the same location as ap-1. So any change made to *Q will also be in *ap-1

int X, Z;
Z = 10;

X = *Q; X is assigned the value in dereferenced pt.

*Q = Z; The contents of Z is assigned to location pointed to by pt.

endH



C: Call Parameters call-by-value: Pointer Parameters effect call by reference (location)

1.1 Problems With Call Types (Variety Of Types Is For Efficiency)

As we said, the origin of parameter passing is in mathematical notation in which all parameters are effectively call by value or copy-in with no assignments to f-p's within the body of the procedure P, and no global parameters, and no pointer parameters. Call by value is sufficient mathematically. It gives no conceptual difficulties. Efficiency concerns gave rise to the variety of parameter types actually found in programming languages. Then, once a new facility is added to a programming language programmers find uses, often unintended, exotic and dangerous, for it and then invent generalizations which are easily implemented. An example of the global variable problems is described below.

Despite its sufficiency, **Call-by-Value (Copy)** requires executing assignments for call generated initial values of parameters to local f-p variables (an **inefficiency, prohibitive in the case of complex data-structures like arrays.**)

If the f-p is not assigned a value anywhere in P (copy in), and the a-p is restricted to being names of data-structures (defined in declared data-structures, or by implicit declaration), and there are no global variables used in P, then his inefficiency can be avoided by using call by reference in, or pointers. However new dangers arise. (Call-by-value and absence of globals) isolates assignments inside from those outside a called procedure.

The problem that arises with this replacement of call-by-value with call-by-reference when global variables are in a procedure is illustrated by the following:

```

procedure F(x: copy in)
  x = ap-1
  A = 75;
  B = x;
end procedure;

```

On using F(A)

```

A = 1;
F(A);

```

```

x = 1
A = 75
B = x; = 1

```

Value of arg A replaces x. Procedure is isolated from outside world

```

procedure F(x: reference)
  A = 75;
  B = x; /*x<- addr( ap-1)
end procedure;

```

On using F(A)

```

A = 1;
F(A);

```

```

x = 1
A = 75
B = A; = 75

```

Name of arg replaces x but same name is assigned in proc. Being global it is alive in and out of proc.

Example Different Effects of Global Evaluated inside and Outside A Procedure.

Pass By Result (Copy-out) When there is more than one result parameter, say R1 and R2 and a call contains the same argument for both, say J. Then before exit it is necessary to assign R1 to J and then R2 to J or vice-versa. The confusion is usually avoided by making such calls illegal. Such calls are usually detectable.

Also if two arguments for result parameters R1 and R2 are A(J) and J, and R1 must be assigned to A(J)'s address as it is on entry and R2 to J just before exit-writing in the wrong order is a danger and misleading.

The call says "compute 2 values and assign the first to R1 and the second to R2" -With Aliasing compute 2 values assign the first to J and the second to J ??

```

procedure F(int X value, int R1 result, int R2 result)

```

```

{int X=ap-1
R1 int
R2 int

```

Formal Body: Compute values for R1 and R2

```

ap-2 = R1
ap-3 = R2 }

```

```

call F(17, J, J)

```

```

{int X=ap-1
R1 int
R2 int

```

Formal Body

```

J = R1
J = R2

```

```

call F(17, J, A[J])

```

```

{int X=ap-1
R1 int
R2 int

```

Formal Body

```

A[J] = R1
J = R2}

```

Pass By Reference or Location or Reference inout Again with two reference parameter, a call with I for both arguments will result in I being assigned twice with perhaps different values. This is called **aliasing**. In addition to the problem above, some procedure optimizations become questionable since they cause error if a call involving aliasing occurs. Such cases of aliasing are usually detectable.

```

procedure F(A,B:location (reference inout) integer, C:result)

```

```

C:integer
A = 5
B = 25
C = A+B

```

```

ap-3 = C
endF

```

Original Code

```

C:integer

```

```

A = 5
B = 25
C = 30

```

```

ap-3 = 30
endF

```

Optimized

```

call F(X, X, Y)

```

```

X := 5
X := 25
C :=50 (=X+X)
Y := 50

```

Original Code

```

X := 5
X := 25
C := 30
Y = 30

```

Optimized

A simple call involving aliasing F(X,X,Y) will leave the following in Y the each of the versions.

C=50 in Original and C=30 in the Optimization

Example Of Optimization Invalidated By Aliasing.

So the optimization can't be used since it gives different results than the original when the reference arguments are the same due to aliasing. A similar thing happens if A and B are call by value and the a-p's for these are pointers.

Then the corresponding program and optimization is:

```
procedure F(int *A, int *B:, int C:result)
```

```
int *A = *ap-1
int *B = *ap-2
int C :
*A = 5
*B = 25
C = *A + *B
ap-3 = C
end F
```

ORIGINAL

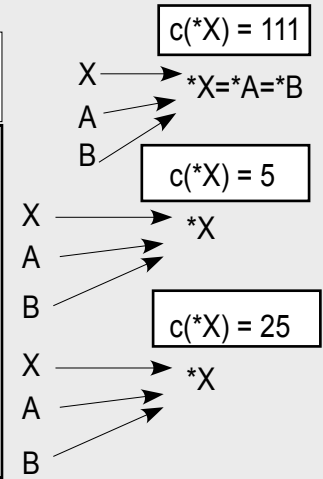
```
int *A = *ap-1
int *B = *ap-2
int C:
*A = 5
*B = 25
C = 30
ap-3 = 30
end F
```

OPTIMIZED

```
int *X=111
call F(X, X, Y)
```

```
int *A; A = X
int *B ;B = X
int C
*A = 5
*B = 25
C = *A + *B
Y = C
(Y = 2 c(*X)
=50)
```

ORIGINAL Called



And this will lead to problems if a call when the a-p's for A and B, are the same pointer.

Pointer Example Of Optimization Invalidated By Aliasing.

This optimization problem does not arise (i.e. the optimization works) if A and B are are changed to **copy-inout value-result parameters.**

```
procedure F( int A copy-inout, int B copy-inout: int C copy-out)
```

```
int A = ap-1;
int B = ap-2;
int C ;
A = 5;
B = 25;
C = A + B;
ap-1 = A;
ap-2 = B;
ap-3 = C;
end F
```

ORIGINAL

```
int A = ap-1;
int B = ap-2;
int C ;
A = 5;
B = 25;
C = 30;
ap-1 = A;
ap-2 = B;
ap-3 = C;
end F
```

OPTIMIZED

Example OK with copy-inout value-result parameters.

A simple call involving aliasing: F(X,X,Y) will give the same result in both versions. This is so because different local variables are used in the procedure for the two value-result parameters even when the corresponding arguments are the same the optimization cannot be destroyed by duplicated arguments. There is no aliasing problem and so the optimization can be used.

Pass By Value-Result (Copy in out) This is like passing by location and might give rise to aliasing, except that locally, within the procedure, even if the call contains two identical arguments, the two corresponding value-result parameters are not the same location so the aliasing difficulty does not occur. On the otherhand, because of the resultant, Prefix assignment to parameters this is less efficient than call by location. Overflow that occurs for a value-result parameter in a procedure will not be reflected in the corresponding argument if processing stops as soon as error is detected. This differs from call by location and the distinction must be known for correct error analysis.

C: Call Parameters

In C all prameters are call-by-value. Pointer Parameters however give the effect of call by reference. This is illustrated in the following examples.

```

void swap (int p1, int p2 )
{ int temp;
  int p1 = ap1;
  int p2 = ap2;
  temp = p1
  p1 = p2;
  p2 = temp;
}

```

Invalid

In C: Can't Assign Directly To A (Value) Parameter. Even if allowed as copy-in does not affect variable in argument when called.

Problem: Swap with Call By Value

```

void swap (int p1 copy-inout int p2 copy-inout)
{ int temp;
  temp =p1
  p1 = p2;
  p2 = temp;
}

```

Formal Body

```

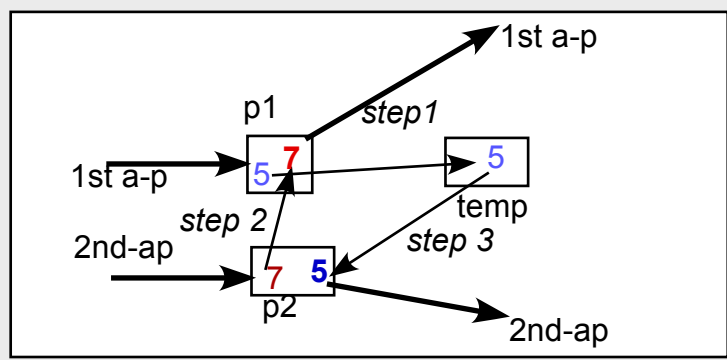
void swap (int p1 copy-inout, int p2 copy-inout)
{ int temp;
  p1 = ap-1
  p2 = ap-2
  temp = p1
  p1 = p2;
  p2 = temp;
  ap-1 = p1
  ap-2 = p2
}

```

Formal Body

Actual Body

The call
 int x= 5; int y =7;
swap (x, y)



With Use Of Copy-Inout Can Achieve Equivalent To Call By Reference

```

void swap (int *p1, int *p2 )
{ int temp;
  temp = *p1
  *p1 = *p2;
  *p2 = temp;
}

```

Formal Body

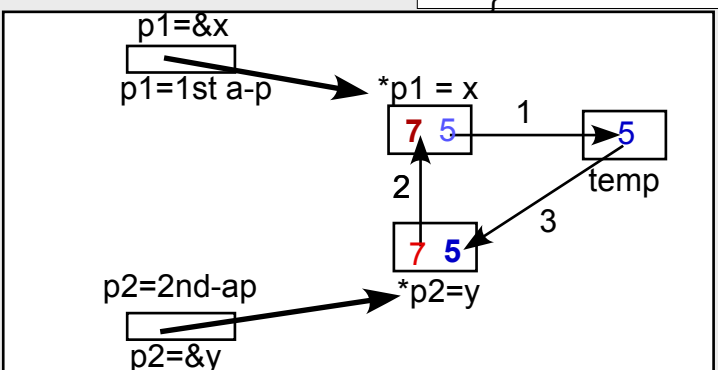
```

void swap (int *p1 , int *p2 )
{ int temp;
  int * p1, p1 = ap-1; ^
  int * p2 = ap-2;
  temp = *p1
  *p1 = *p2;
  *p2 = temp;
}

```

Actual Body

The call
 int x= 5; int y =7;
 int *p1, *p2;
 p1 = &x; p2 = &y;
swap (p1, p2)



C: All Parameters Are Call by Value or Copy-in

With Use Of Pointers Can Achieve Equivalent To Call By Reference

Call By With Copy Inout) and Copy-In With Pointers Swap Example

<pre>double Tave (int a[], int n) {double sum = 0.0; inr t int *p p = a; for(i = 0; i <n; i++); {sum += *p; p++; } if (n != 0) return sum / n; else return 0.0;</pre>	<p>Formal Body</p>
--	------------------------

<pre>double Tave (int a[], int n) {double sum = 0.0; int *a, a = ap-1; int n = ap-2;</pre>	<p>Actual Body</p>
<pre>it i; int *p; p = a; for(i = 0; i <n; i++); {sum += *P; P++; } if (n != 0) return sum / n; else return 0.0;</pre>	

<p>THE CALL: tave (A[], m)</p> <p>RESULTS IN:</p> <pre>int *p = A int n = m;;</pre> <pre>for(i = 0; i < n; i++) { sum += *p; p++; } if (n != 0) return sum / n; else return 0.0;</pre>
--

C: All Parameters Are Call by Value or Copy-in Array Example

CALL BY VALUE WITH POINTERS

Pass By Name This technique has the characteristic of deferred evaluation. The expression is not evaluated until the corresponding parameter is encountered in the procedure body as a result of a call. The text of the argument is substituted for each corresponding parameter occurrence, and executed when encountered. But this does not make clear what will happen when a local variable is the same as part or all of a name argument. Again there would best be a distinction made-which complicates compilation.

Value Returning Procedures-Function

Possibility of side-effects when using location parameters and returning their value for example. (Mixing procedure and function notion can lead to problems). Also there seem to be limitations in most languages in the number and type of data allowed as a return.

Summary

Properties	Parameter Types		
	copy-in=value copy -out=result copy-iinout = value- result	reference= location IN or OUT result	name= textual
Has PREAMBLE in which local variables for f-p are declared	IN or OUT result	no	no
In PREAMBLE f-p local variables are initialized with value of a-p	IN	no	no
In POSTSCRIPT assign f-p value to corresponding a-p address	OUT result	no	no
There are statements in procedure body that assign values to an f-p	IN or OUT result	OUT result	called *OUT parameter
a-p must be a reference to a data location, ex. a variable, an array or record element, etc.	OUT result	OUT result	if *OUT parameter
a-p can be an expression	IN	no	if not *OUT parameter
Before executing procedure SUBSTITUTE a-p's for f-p's	no	SUBST addr of a-p for f-p	yes
Before executing procedure SUBSTITUTE address of a-p's for f-p's	no	yes	no

POINTERS

x means [location] x,

(x is the name of that location)

c(x) is the contents of [location] x, In order for c(x) to be meaningful x must name a location .

so c(c(x)) implies that c(x) as well as x must name locations

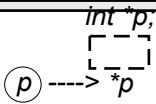
&x is location of [location]x

*x is c(c(x))

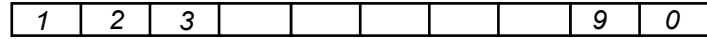
int *x = NULL; It can be pointed to it, but as it stands it doesn't point to any location ,

*x = 5 is a legitimate following instruction.

<pre>int n = 20, m = 111; c(n) == 20 c(m) == 111</pre>	
<pre>int *p, *q = 7; Defines p and q to be pointers which point to (= their dereferenced value) a location containing an integer c(c(p)) = 7, c(c(q)) = 7</pre>	<p>q is a pointer to location *q which contains 7</p>
<pre>p = &n c(p) BECOMES location c(c⁻¹(n))=n</pre>	
<pre>q = &m; c(q) BECOMES c(c⁻¹(m))=m</pre>	
<pre>printf("p points to %d", *p); print c(*p) = print c(c(p)) = print c(n) = print(20)</pre>	
<pre>p = q; c(p) BECOMES c(q); c(c⁻¹(n)) BECOMES c(c⁻¹(m));</pre>	
<pre>*p = m c(c(p)) BECOMES c(m)</pre>	
<pre>*p = *q + 5; c(c(p)) BECOMES c(c(q)) + 5</pre>	
<pre>int **r = 25; c(c(c(r))) BECOMES 25</pre>	
<pre>*r = q</pre>	



```
static int a[10] = {1,2,3,..., 9,0};
```



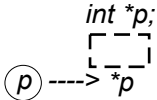
constant cannot be redefined

a[0] a[1] a[2] . . . a[8] a[9]
 a* (a+1)*(a+2)* . . . (a+8)*(a+9)*

```
int *p = a;
while(*p != 0) { printf ("c(%u) == %d, "p, *p);
                ++p; }
```

Unlike a, p can be redefined

c(100)=1,c(104)==2, c(108) == 3 ,..., c(8) == 148, c(9) == 158 Output



```
static char a[] = "abcdefghij";
```



constant cannot be redefined

a[0] a[1] a[2] . . . a[8] a[9]
 a* (a+1)*(a+2)* . . . (a+8)*(a+9)*

```
char *p = a;
while(*p != 0) { printf ("c(%u) == %d, "p, *p);
                ++p; }
```

Unlike a, p can be redefined

c(100)=a,c(101)==b, c(102) == c ,..., c(8) == i, c(9) == j Output

```
char *p = a; p = a+10;
while(--p != a) { putchar(*p);}
putchar("\n");
```

Unlike a, p can be redefined

j i h g ... c b a Output

Pointers-Arrays Integer and Character (Strings)

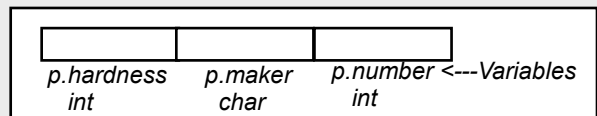
struct pencil {int hardness; char maker; int number}; this is a type definition , that is, pencil is a type of structure

a **pencil** is a collection (type) with 3 datum of the type shown
hardness
maker
number

struct pencil {int hardness; char maker; int number} var; this is a type definition plus a definition of the variable v,

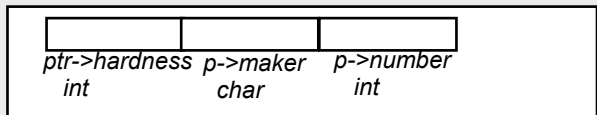
struct pencil p p is a variable of type pencil p is a structure so
 so a set of variables are defined

p.hardness
 p.makerp.
 p.number



struct pencil. *ptr ptr is a pointer to is a variable of type pencil
 so a set of variables are defined

ptr -> hardness
 ptr -> make-
 ptr -> number



Poiners-Structures Collections Of Heterogenious Types