16 Call-by-value

16.1 Digression on variables

Every variable (and routine) in C has a *scope.*

For example,

```
...
void xxx ( int n, double a[10][10] )
{
  int i;
        /*
         * i is a local variable, whose scope is till
         * the end of routine
         * n is an argument, an initialised local variable
         * its scope is to end of routine
         * so is a, but remember a is an address
         */
  for (i=0; i<n; ++i)
  {
    int j;
        // scope of j is from here to next curly brace
    for (j=0; j<n; ++j)
      a[i][j] = i==j;
  }
}
```

Consider

```
#include <stdio.h>
void print ( int n )
{
  printf("Integer... %d\n", n);
  ++n;
  printf("Integer... %d\n", n);
}

main()
{
  int m = 3;
  print(m);
  print(m);
}
```

the routine argument `n` is like an *initialised local variable.* Compiling and running the program:

```
%gcc p.c
```

```
%a.out
Integer... 3
Integer... 4
Integer... 3
Integer... 4
%
```

That is, the value of `m` is copied to `n,` and `n` is local to the routine. The change to `n` (local) does not affect `m` (in the calling routine `main`). If we change the `main()` routine to

```
main()
{
  print (3);
  print (3);
}
```

we get the same output.

There are 5 recognised ways of argument-passing (parameter-passing)

- Call by value
- Call by reference
- Call by result
- Call by value-result
- Call by name

The last three are irrelevant to us: the last is bizarre, occurring in the 1960s language Algol and in the 'funarg problem' in Lisp.

Actually, the `#define` feature in C, which should **never** be used except to give names to constants, if used with arguments has all the difficulties of call-by-name.

In call-by-reference, the subroutine argument is identical with the argument passed, i.e., occupies the same memory location. This was the natural way when Fortran was invented, and in the early compilers it had very odd effects.

Call-by-reference is easily simulated in C, because *location* is central to C design. Not introduced yet.

In most (or all?) languages, the constant 3 would be stored in a memory location when the program `a.out` was loaded into central memory, and whenever 3 was used in the program, the value would be taken from this location. In some early Fortran compilers, the following could happen — illustrated as if it would happen in C, that is, if C had *call-by-reference*:

```
#include <stdio.h>
void print ( int n )
{
  printf("Integer... %d\n", n);
```

```
  ++n;
  printf("Integer... %d\n", n);
}

main()
{
  print( 3 );
  print( 3 );
}

%a.out
Integer... 3
Integer... 4
Integer... 4
Integer... 5
%
```

The programming language PL/I had a mixture of call-by-reference together with 'automatic conversion' to 'dummy arguments' which made odd things happen. If you weren't careful, subroutine calls would be call-by-reference sometimes and call-by-value other times.

Summarising

C has call-by-value. Subroutine arguments are initialised local variables.