17 Variables, arguments, and the runtime stack

17.1 Variables

Variables have a *scope*, that part of the program in which they are recognised. Mostly, variables (and routine arguments) are local to routines/functions. Or they may be even more restricted:

```
  int i;  // scope of i is from here to the end of the routine

  for (i=0; i<m; ++i)
  { int j;  // scope of j is down to the next curly brace
    for ( j=0; j<n; ++j )
      a[i][j] = 0;
  }
```

A variable can be declared outside all routines: then it is a *global variable*

```
#include <stdio.h>

int version_no = 1;

int same_version ( int v )
{
  return v == version_no;
}

main()
{
  printf("Version %d\n", version_no);
  if ( ! same_version (5) )
    printf("Old version\n");
}
```

This is a good use of global variables. *They should be used sparingly,* because they can be changed anywhere in the program, which is a source of errors. *Local is good, remote is bad.*

Where variables with the same name are declared in different places, it is the *closest* variable which prevails.

```
#include <stdio.h>
main()
{
  int i, j=25;  // j is initialised, not i
  int sum=0;
  for ( i=0; i<3; ++i )
  { int j;
    for (j=0; j<i; ++j)
```

```
      sum = sum+j;
  }
  printf("i %d j %d sum %d\n", i,j,sum);
}
```

After careful simulation, we get

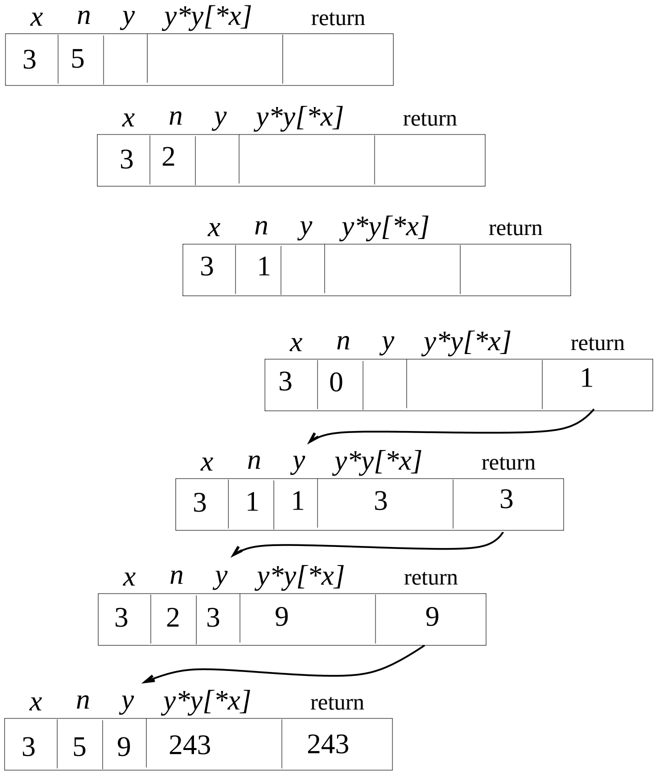
```
  i    i<3   sum  j   j_2  j_2<i
                 25
              0
  0
         1
                       0
                             0
              0
  1
         1
                       0
                             1
              0
                       1
                             0
  2
         1
                       0
                             1
              0
                       1
                             1
              1
                       2
                             0
  3
         0
Prints:
i 3 j 25 sum 1
```

17.2 The runtime stack

Every time a routine or function is called, a *stack frame* is created and ‘pushed’ on the runtime stack. When it ends, its stack frame is removed and the calling routine resumes.

Example:

```
double topow ( double x, int n )
{
  if ( n==0 )
```



```
  { return 1; }
  else
  {
    double y = topow ( x, n/2 );
    if ( n%2 == 0 )
      return y*y;
    else
      return y*y*x;
  }
}
main()
{  printf("%f\n", topow(3,5)); }
```

17.3 Automatic and static variables

Global variables last until the program ends. Routine variables last from the start to the end of the routine only. They are called *automatic variables*.

It is also possible to have *static* variables. Their *scope* is local, but their *duration* is until the end of the program. One can count, for example, the number of times a subroutine has been called using a static variable. *Initialisation of static variables is crucial, and is done just once.* For example,

```
prompt% cat static.c
#include <stdio.h>
```

```
int countme ()
{ static int n = 0;
  ++n;
  return n;
}
main()
{ int i;
  for (i=0; i<3; ++i)
    printf("countme is called for the %d-th time\n", countme() );
}
prompt% gcc static.c
prompt% a.out
countme is called for the 1-th time
countme is called for the 2-th time
countme is called for the 3-th time
prompt%
```