

# Function I

- general structure or the *prototype* of a function:

- new-style (preferred):

```
return-type
function-name(parameter1-type parameter1-name,
              parameter2-type parameter2-name, ...)
{
    function-body
}
```

- old-style (might see in legacy code):

```
return-type
function-name(parameter1-name,
              parameter2-name, ...)
parameter1-type parameter1-name;
parameter2-type parameter2-name;
...
{
    function-body
}
```

## Function II

- a simple example:

```
double
power1 (double xx, int nn)
{
    int ii;
    double retVal = 1.0;

    for (ii = 0; ii < nn; ++ii) {
        retVal = retVal * xx;
    }
    return retVal;
}
```

## Function III

- *declaration*: [this usually appears in a header or a .h file]

```
extern double  
power1 (double xx, int nn);
```

- *definition*: [this usually appears in an implementation or a .c file]

```
double  
power1 (double xx, int nn)  
{  
    int ii;  
    double retVal = 1.0;  
  
    for (ii = 0; ii < nn; ++ii) {  
        retVal = retVal * xx;  
    }  
    return retVal;  
}
```

- the keyword `extern` is only used when the *declaration* and the *definition* live in separate files: a .h and a .c file or two different .c files (*bad practice!*)

## Function IV

- always make sure the either the declaration or the definition of a function appears before it is used
- all the variables within a function are its *local* variables and they are *born* the when program using the function enters it
- these local variables *die* when the program using the function leaves it
- these local variables are *born* again when the program using the function re-enters it
- we would say these local variables have *function scope*

## Function VI

- the special keyword `void` means that
  - the function has no arguments/parameters
  - the function returns nothing, as you might guess!

```
void
a_void_funcion1 (void)
{
    printf("inside a void function1\n");
}
```

- in fact you might be able to omit the second `void`, but the previous usage is preferred, its more explicit

```
void
a_void_funcion2 ( )
{
    printf("inside a void function2\n");
}
```

## Aside I

- so, repeat: within the curly braces i.e. within { ... } of a function we can declare local variables which would have *function scope*
- similarly within any set of curly braces i.e. within { ... } of e.g. for, while, do-while, if, else if, else we can declare local variables to have the corresponding *local scope*, a silly example:

```
for (ii = 0; ii < N_SMALL; ++ii) {
    int jj = ii * ii;

    printf("%d\t", jj);
}
printf("\n");

ii = 0;
while (ii < N_SMALL) {
    int jj = ii * ii;

    printf("%d\t", jj);
    ++ii;
}
printf("\n");
```

## Aside II

- note, in such a case the variables declaration should be the first line of the scope!
- do not use this feature with `switch`, might get compiler warnings
- these local variables *die* when the program using the scope leaves it
- in general, use as many local variables as you can

# Call by Value I

- pass a copy of the argument to the function:

```
double
power2 (double xx, int nn)
{
    double retVal = 1.0;

    for ( ; nn > 0; --nn) {
        retVal = retVal * xx;
    }
    return retVal;
}
```

- note inside `power2`, the value of `nn` is being changed and it would be 0 when the program leaves the function
- the user of the function `power2` wouldn't notice this change in the value of `nn`

## Call by Value II

- since we pass a copy of the argument to the function any change of the value within the function would not be reflected outside
- when is this useful?
  - when we want to prevent the called function from (inadvertently) changing the value of the parameters

- example:

```
int
main (int argc, char **argv)
{
    int nn = 3, pp = 3;
    double xx = 2.0, yy = 2.0;

    printf("2^3 = %g\n", power2(2.0, 3));
    printf("%g^%d = %g\n", xx, nn, power2(xx, nn));
    printf("%g^%d = %g\n", yy, pp, power2(yy, pp));
    return 0;
}
```

- note the name of the variables don't matter: both `power2(xx, nn)` and `power2(yy, pp)` work fine!

# Call by Reference I

- pass the address to the function:

```
void
power3 (double xx, int nn, double *retVal)
{
    for (*retVal = 1.0; nn > 0; --nn) {
        (*retVal) = (*retVal) * xx;
    }
}
```

- note inside `power2`,
  - the value of `nn` is being changed and it would be 0 when the program leaves the function
  - the value of `*retVal` is being changed it will be the required power value:  $xx^{nn}$  when the program leaves the function
- the user of the function `power2` wouldn't notice the change in the value of `nn` but s/he will see the change in the value of `*retVal`

## Aside III

- note the usage `*retVal = 1.0;` inside the `for` loop, you could if wanted to have a (fictious) `for` loop like:

```
for (ii = 0, jj = 123, *dd = 3.45; ii < 5; ++ii) {  
    loop-body  
}
```

i.e. several initializations in one expression

- note several initializations in one expression is done using a comma i.e. “,” not a semi-colon i.e. “;”
- of course the following do work just fine as well:

```
jj = 123;  
*dd = 3.45;  
for (ii = 0; ii < 5; ++ii) {  
    loop-body  
}
```

- one-line `for`, `while`, `do-while`, `if`, `else if`, `else` statements may do without the curly braces, the following is perfectly valid:

```
for (ii = 0; ii < nn; ++ii)  
    retVal *= xx;
```

## Call by Reference II

- since we pass the address of a variable or a pointer any change within the function would be reflected outside
- so its risky, isn't it? when is this useful then?
  - sending large amounts of data to a function through copying isn't a good idea, is it? call by reference comes to the rescue:

- example:

```
int
main (int argc, char **argv)
{
    int nn = 3;
    double xx = 2.0, retVal;

    power3(xx, nn, &retVal);
    printf("%g^%d = %g\n", xx, nn, retVal);
    return 0;
}
```

- note the double `retVal` in `main( )` is a different animal from the parameter `double *retVal` of `power2( )`

## Some Keywords I

- the *static* keyword, appearing a .c file:

```
static double
power1 (double xx, int nn)
{
    int ii;
    double retVal = 1.0;

    for (ii = 0; ii < nn; ++ii) {
        retVal = retVal * xx;
    }
    return retVal;
}
```

- here *static* makes the function `power1( )` visible only within the .c file
- we say the function `power1( )` has *file scope*

## Some Keywords II

- the *static* keyword, within a function:

```
double
power1 (double xx, int nn)
{
    static int ii;
    double retVal = 1.0;

    for (ii = 0; ii < nn; ++ii)
        retVal = retVal * xx;
    return retVal;
}
```

- here *static* makes the variable *ii* *live* as long as the program using the function `power1()` lives i.e. it is not *reborn* every time the program enters the function
- *use this feature sparingly*

## Some Keywords III

- consider the function:

```
int
baz (void)
{
    static int ii = 0;

    return ++ii;
}
```

- now what would repeated calls to baz( ) do?

```
printf("static value: %d\n", baz( ));
printf("static value: %d\n", baz( ));
printf("static value: %d\n", baz( ));
```

- it would print 1, 2, 3, does that make sense?

## Some Keywords IV

- the `static` keyword, within a file, *outside of* any function i.e. at the top level of a file:

```
static double retVal;

double
power1 (double xx, int nn)
{
    static int ii;

    retVal = 1.0;
    for (ii = 0; ii < nn; ++ii) {
        retVal = retVal * xx;
    }
    return retVal;
}
```

- here `static` makes the variable `retVal` a *file scope* “global” variable i.e. it is visible to only and all the functions in the enclosing file
- these types of variables are often used to avoid repeated initialization of “stuff”: (discuss likelihood-evaluation-data-reading example)

## Some Keywords V

- removing the `static` keyword i.e. `double retVal;` (at the top level) would make it potentially visible to other `.c` files as well, it has a “global” scope now
- now `extern double retVal;` could be used in, say, `file2.c` for accessing the “global” variable `retVal` where at the top level of `file1.c` we first said `double retVal;`
- do not make things global unless it is absolutely necessary, avoid using “global” variables at all cost:
  - one way is to pass around a pointer to the potential “global” variable (or a pointer to a *structure*, whatever that means, containing multiple “global” variables) to all the functions

## Some Keywords VI

- the `const` keyword as variable type modifier:

```
char *  
strcpy(char *dst, const char *src);
```

- here `const` makes the variables `src` constant and thus prevents any (inadvertant) change to `src` attempted by (implementer of the) function `strcpy()`
- it is always a good idea to declare as many parameters `const` as feasible
- when we are passing by reference then if we don't have a `const` before a pointer variable then we know that the function intends to or may change it
- declaring pointer arguments which may not be changed by the implementer of a function as `const` is a good practice and etiquette
- declaring non-pointer arguments as `const` is superfluous because they are passed by value anyway

## Some Keywords VII

- the order of the `const` keyword does not matter:
  - example with `const` before type: more commonly used

```
void
arrcpy1 (int len, double *dst, const double *src)
{
    int ii;

    for (ii = 0; ii < len; ++ii)
        dst[ii] = src[ii];
}
```

- example with `const` after type: more readable from right to left

```
void
arrcpy2 (int len, double *dst, double const *src)
{
    int ii;

    for (ii = 0; ii < len; ++ii)
        dst[ii] = src[ii];
}
```

## Some Keywords VIII

- const modifier for return values of a function:
  - superfluous when the function returns a non-pointer
  - THIS IS A BIT TOO ADVANCED!: its useful to return a const pointer to an “internal” member of a structure a private data members of a class which are meant to be readonly e.g. think about an Java-like immutable string class
  - read the links I posted on our class bulletin board for more on this

## Some Keywords IX

- declaring const variables: const variables can be initialized only once while declaring for good reason

```
double dd = 1.111;  
double const *ddp = &dd;
```

- so what would the following do? it will generate an error!

```
*ddp = 2.222;
```

- but you can fool the compiler by “casting”, but don’t do it!

```
*((double *) ddp) = 2.222;  
printf("dd = %g, *ddp = %g\n", dd, *ddp);
```

- note, (double \*) is the “cast” operator, in general (typename) foo casts variable foo to type typename

## Some Keywords X

- consider the function:

```
double const *  
bar (double *ddp)  
{  
    return ddp;  
}
```

- what do you think the following would do?

```
double dd = 1.111, *ddp;  
  
ddp = bar(&dd);  
*ddp = 3.333;  
printf("dd = %g, *ddcp = %g\n", dd, *ddp);
```

- gcc -pedantic -ansi will just give you warning
- g++ -pedantic -ansi will not compile, will give you error

# Function Within a Function I

- frankly, speaking I don't know a good use of this tool but here is what I discovered
- consider this function:

```
void
test1 (void)
{
    printf("within test1\n");
    void test1 (void)
    {
        printf("within test1::test1\n");
    }
    test1( );
}
```

- when called from main( ) it spits out:

```
within test1
within test1::test1
```

- so you can define function within a function

## Function Within a Function II

- some cool things you could do once you are able to read the following code, for now ignore it:

```
typedef void (*FucntionWithVoidReturnVoidArg) (void);

FucntionWithVoidReturnVoidArg
gen_test2 (void)
{
    void test2 (void)
    {
        printf("within test2\n");
    }
    return test2;
}

int
main (int argc, char **argv)
{
    FucntionWithVoidReturnVoidArg test2;

    test2 = gen_test2( );
    test2( );
    return 0;
}
```

## Function Within a Function III

- this feature is gcc-specific and not very portable
- `gcc -pedantic -ansi` will compile but give you warnings like  
ISO C forbids nested functions
- `g++ -pedantic -ansi` will terminate with an error and refuse to compile the program
- because its an esoteric feature of gcc use of this is not recommended unless you have a compelling reason to use this

# Variable Number of Arguments I

- suppose we could do the following:

```
int
main (int argc, char **argv)
{
    int nn;

    nn = 2;
    printf("sum of first %d positive numbers: %g\n",
          nn, sum_so_many(nn, 1.0, 2.0));
    nn = 3;
    printf("sum of first %d positive numbers: %g\n",
          nn, sum_so_many(nn, 1.0, 2.0, 3.0));
    nn = 4;
    printf("sum of first %d positive numbers: %g\n",
          nn, sum_so_many(nn, 1.0, 2.0, 3.0, 4.0));

    return 0;
}
```

- isn't it nice and handy? I didn't have to write three functions: `sum_two`, `sum_three`, `sum_four`

# Variable Number of Arguments II

- here is how you write such a versatile function:

```
#include <stdio.h>
#include <stdarg.h>

double
sum_so_many (int nn, ...)
{
    int ii;
    double arg, sum = 0.0;
    va_list ap;

    va_start(ap, nn);
    for (ii = 0; ii < nn; ++ii) {
        arg = va_arg(ap, double);
        sum += arg;
    }
    va_end(ap);
    return sum;
}
```

- note you need to use: `#include <stdarg.h>` and the functions, `va_start`, `va_arg` and `va_end`

## Code Files

prog2.c

prog12.c