P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #12

(User-defined) Functions in C

defining a function, prototyping, parameter passing, call-by-value/reference

Functions, Procedures and the Like

Wikipedia:

> In computer science, a subroutine (function, method, procedure, or subprogram) is a portion of code within a larger program, which performs a specific task and can be relatively independent of the remaining code.

Remarks:

- the terms subprogram, subroutine, ... are not clearly defined
- there can be differences especially in the context of different languages
- in Fortan e.g. a difference is made between a *function* (does return a value) and a *subroutine* (does not)

Motivation for Subprograms

Historically:

the original motivation (in assembler programming) was to reduce program size (i.e. save memory) by separating units of interrelated statements that are executed repeatedly

Procedural Paradigm:

the idea behind splitting the program up into individual (short) units is to enhance

- readability and clarity of the code structure
- maintainability (changes are local and not duplicated)
- re-use of code, e.g. in the form of libraries

Subprograms in C (1/2)

- In C any subprogram is called a function.
- All functions in C are external, i.e. we cannot define functions within other functions.
- We have already
  - written our own function (`main` is a function)
  - and used library functions (printf(), malloc(), ...)
- Now we want to implement additional user-defined functions.

Subprograms in C (2/2)

- A function definition takes two parts:

  function header + function body

- The general structure of the function header is:

  return type *function name* (list of parameter types and names)

- The function body is a collection of statements enclosed in $\{\ldots\}$

Example (1/3)

```
1  #include <stdio.h>
2
3  int main( void ) {
4
5    int upperLimit = 100;
6    int sum;
7
8    // sum up all integers from 1 to upperLimit
9    sum = gaussSum( upperLimit );
10   printf( "sum_{k=1}^%d = %d\n", upperLimit, sum );
11
12 }
13
14 int gaussSum( int n ) {
15   int val;
16   val = n * (n+1) / 2;    // evaluated left to right!
17   return val;
18 }
```

Example (2/3)

- When we compile our demo program, this is what GCC tells us:

```
gcc gaussSum.c
gaussSum.c: In function ‘main’:
gaussSum.c:9:9: warning: implicit declaration of function
   ‘gaussSum’ [-Wimplicit-function-declaration]
   sum = gaussSum( upperLimit );
         ^~~~~~~~
```

- C is designed to (also) work with one-pass compilers.
- What GCC is telling us is that when it encounters our call to `gaussSum` in line #9, it has no idea, yet, of
  - how many parameters the function expects to receive
  - what their types should be
  - what type of value it returns
- It will implicitely try to derive this from the call/usage.

Example (3/3)

- There are two ways to fix the issue:
  1. We can place the definition of our function `gaussSum()` before instead of after `main` where it is first used.
  2. We can add a (prototype) declaration of `gaussSum()`.
- The second approach is the one we have used so far with the library functions:

> Including `stdio.h` e.g. provides a prototype for the `printf()` function.

Declaration vs. Definition (1/3)

In C/C++ there is a subtle but important difference in these two terms.

Definition

When we define a variable, we inform the compiler on all properties [data type, storage class modifier (e.g. `register`, size in case of a static array), type qualifiers (e.g. `const`)] it needs to know and it will allocate memory for the variable.

When we define a function, we provide both its header and its body to the compiler. All it needs to generate machine code, allocate memory for local variables, etc.

Declaration vs. Definition (2/3)

Declaration

When we declare a variable, we only provide minimal information to the compiler and it will not allocate memory.

When we declare a function, we only provide information on its interface (its header), but not what it does (its body).

This will allow the compiler to check whether we use the function syntactically correct (e.g. right number and type of parameters)

Declaration vs. Definition (3/3)

Summary

A declaration provides basic attributes of a symbol: its type and its name.

A definition provides all of the details of that symbol – if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored.

Often, the compiler only needs to have a declaration for something in order to compile a file into an object file, expecting that the linker can find the definition from another file. If no source file ever defines a symbol, but it is declared, you will get errors at link time complaining about undefined symbols.

Alex Allain

Example with Prototyping

```c
#include <stdio.h>

// add a declaration of gaussSum
int gaussSum( int n );

// int gaussSum( int k );    parameter name not important
// int gaussSum( int );      can also be omitted

int main( void ) {
  // ...
  sum = gaussSum( upperLimit );
  // ...
}

int gaussSum( int n ) {
  int val;
  val = n * (n+1) / 2;
  return val;
}
```

Functions & void

- The data type void must be used to indicate that a function does not return something:

```
void warn( int stepCount ) {
  printf( "Warning: No convergence after %d steps\n",
      stepCount );
}
```

- It should also be used to indicate that it accepts no arguments:

```
double genRandomNumber( void ) {
  return rand() / (double)RAND_MAX;
}
```

Return Values

```c
int main( void ) {

  // returned values are r-values: we can assign them to
  // l-values or use them as parameters in function calls
  double x = pow( 4.0, 2.0 );
  x = sqrt( pow( 3.0, 2.0 ) );

  // we can either make use of the return value
  char mesg[] = "Let's check how many characters this is!";
  int count;
  count = printf( "%s\n", mesg );

  // or ignore it
  printf( "strlen = %lu, count = %d\n", strlen( mesg ), count );
}
```

```
Let's check how many characters this is!
strlen = 40, count = 41
```

Parameter Passing Example

```c
void stepVal( int a ) {
  printf( "inside stepVal (1): a = %d\n", a );
  a += 1;
  printf( "inside stepVal (2): a = %d\n", a );
}

int main( void ) {
  int a = 2;
  printf( "in main before stepVal(): a = %d\n", a );
  stepVal( a );
  printf( "in main after  stepVal(): a = %d\n", a );
}
```

Parameter Passing Example

```
void stepVal( int a ) {
  printf( "inside stepVal (1): a = %d\n", a );
  a += 1;
  printf( "inside stepVal (2): a = %d\n", a );
}

int main( void ) {
  int a = 2;
  printf( "in main before stepVal(): a = %d\n", a );
  stepVal( a );
  printf( "in main after  stepVal(): a = %d\n", a );
}
```

in main before stepVal(): a = 2
inside stepVal (1): a = 2

inside stepVal (2): a = 3
in main after stepVal(): a = 2

Call-by-Value

- Different strategies exist for passing parameters to subprograms.
- The C language uses a technique denoted as call-by-value.
- The formal parameters in the interface of a function
  - are local variables existing only in the scope of the function
  - when the function is called the values of the arguments are copied into the local variables

```
void doSomething( int iVal, double dVal )
{ ... }

int count = 2;
doSomething( count, 4.0 );
```

- Changes to formal parameters do not affect arguments.

Call-by-Reference

- A well-know alternative strategy is denoted as call-by-reference.
- This is the variant used in the Fortran language family.
- The formal parameters in the interface of a function
  - are only placeholders (dummy parameters)
  - when the function is called they are replaced by references to the actual arguments
  - changes to formal parameters do affect the arguments

Classic Swap Test

```
void swap( int aLoc, int bLoc ) {
  int aux = aLoc;
  printf( "in swap (1): (%d,%d)\n", aLoc, bLoc );
  aLoc = bLoc;
  bLoc = aux;
  printf( "in swap (2): (%d,%d)\n", aLoc, bLoc );
}
int main( void ) {
  int a = 1;
  int b = 5;
  printf( "in main before swap(): (%d,%d)\n", a, b );
  swap( a, b );
  printf( "in main after  swap(): (%d,%d)\n", a, b );
}
```

Classic Swap Test

```
void swap( int aLoc, int bLoc ) {
  int aux = aLoc;
  printf( "in swap (1): (%d,%d)\n", aLoc, bLoc );
  aLoc = bLoc;
  bLoc = aux;
  printf( "in swap (2): (%d,%d)\n", aLoc, bLoc );
}
int main( void ) {
  int a = 1;
  int b = 5;
  printf( "in main before swap(): (%d,%d)\n", a, b );
  swap( a, b );
  printf( "in main after  swap(): (%d,%d)\n", a, b );
}
```

in main before swap(): (1,5)
in main after swap(): (1,5)

in swap (1): (1,5)
in swap (2): (5,1)

Emulation with Pointers (1/2)

- C allows us to work with memory addresses.
- Hence we can emulate call-by-reference using pointers.

```
void swap( int* ap, int* bp ) {

  int aux = *ap;

  printf( "in swap (1): (%d,%d)\n", *ap, *bp );
  *ap = *bp;
  *bp = aux;
  printf( "in swap (2): (%d,%d)\n", *ap, *bp );
}
```

Emulation with Pointers (2/2)

- When we call our new `swap()` function we provide the memory address of the objects we want to swap:

```
int main( void ) {
  int a = 1;
  int b = 5;
  printf( "in main before swap(): (%d,%d)\n", a, b );
  swap( &a, &b );
  printf( "in main after  swap(): (%d,%d)\n", a, b );
}
```

- Now the output becomes:

```
in main before swap(): (1,5)
in swap (1): (1,5)
in swap (2): (5,1)
in main after  swap(): (5,1)
```

Overview/Comparison (1/2)

- The languages currently most often used in Computational Science use different approaches:

C	call-by-value
Fortran	call-by-reference
Python	call-by-sharing/object/assignment
C++	call-by-value + call-by-reference

- Python: everything is an object, either mutable or immutable and no variables exist (only names)
- C++: besides pointers also knows references (variables for fixed addresses associated with entities, cannot be copied)

Overview/Comparison (2/2)

	arguments	
	passing via keyword	optional (w/ default values)
C	✖	✖†
Fortran	✔	✔
Python	✔	✔
C++	✖	✔

† only via workarounds

- Python: `print ('comma', 'separated', 'words', sep=', ')`
- C++: `void show( std :: string mesg = "Greetings!" )`

Experiment (1/2)

- What happens, when we run this codelet?

```
int func( int arr[5] ) {
  arr[0] = -2;
  return sizeof( arr );
}

int main( void ) {
  int aVals[5] = {0};
  printf( "aVals[0]: %d\n", aVals[0] );
  printf( "aVals: %d\n", func( aVals ) );
  printf( "aVals[0]: %d\n", aVals[0] );
}
```

Experiment (1/2)

- What happens, when we run this codelet?

```
int func( int arr[5] ) {
  arr[0] = -2;
  return sizeof( arr );
}

int main( void ) {
  int aVals[5] = {0};
  printf( "aVals[0]: %d\n", aVals[0] );
  printf( "aVals: %d\n", func( aVals ) );
  printf( "aVals[0]: %d\n", aVals[0] );
}
```

- We get this output:

```
aVals[0]:  0
aVals:  8
aVals[0]:  -2
```

Experiment (1/2)

- What happens, when we run this codelet?

```
int func( int arr[5] ) {
  arr[0] = -2;
  return sizeof( arr );
}

int main( void ) {
  int aVals[5] = {0};
  printf( "aVals[0]: %d\n", aVals[0] );
  printf( "aVals: %d\n", func( aVals ) );
  printf( "aVals[0]: %d\n", aVals[0] );
}
```

- We get this output:

```
aVals[0]:  0
aVals:  8
aVals[0]:  -2
```
← change in func affects array in main!

Experiment (1/2)

- What happens, when we run this codelet?

```
int func( int arr[5] ) {
  arr[0] = -2;
  return sizeof( arr );
}

int main( void ) {
  int aVals[5] = {0};
  printf( "aVals[0]: %d\n", aVals[0] );
  printf( "aVals: %d\n", func( aVals ) );
  printf( "aVals[0]: %d\n", aVals[0] );
}
```

- We get this output:

```
aVals[0]:  0
aVals:  8          ← size of a pointer!
aVals[0]:  -2      ← change in func affects array in main!
```

Experiment (2/2)

- Compiling with `-Wall` we get

```
gcc -Wall passingArrays.c
passingArrays.c: In function ‘func’:
passingArrays.c:6:16: warning: ‘sizeof’ on array function
parameter ‘arr’ will return size of ‘int *’
[-Wsizeof-array-argument]
   return sizeof( arr );
                  ^
passingArrays.c:4:15: note: declared here
 int func( int arr[5] ) {
               ~~~
```

Summary: Arrays

- The declaration `int arr[5]` does not define a local static array of five ints.
- What the function receives when we pass it an array is the starting address (address of its first element).
- The following three versions work the same

```
int func1( int arr[5] );
int func2( int arr[] );
int func3( int* arr );
```

- The value in [] is not important for 1D arrays (extent in 1st dimension not required for mapping function!).
- Passing an array does not involve costly copying of all its entries.

Rank-n Arrays (1/5)

- For multi-dimensional static arrays we need to provide all info required for the mapping function:

```
void setEntries( int nRows, int nCols, int mat[][nCols] ) {
  for( int i = 0; i < nRows; i++ ) {
    for( int j = 0; j < nCols; j++ ) {
      mat[i][j] = 0.0;
    }
  }
}
```

- Passing the array is then straightforward

```
int main( void ) {
  int nRows = 5, nCols = 5;
  int dVals[nRows][nCols];
  setEntries( nRows, nCols, dVals );
}
```

Rank-n Arrays (2/5)

- When we instead use

```
void setEntries( int nRows, int nCols, int mat[][] );
```

- We run into problems

```
passingArraysMultiDim.c:63:45: error: array type has incomplete
element type 'int[]'
void setEntries( int nRows, int nCols, int mat[][] ) {
                                            ^~~
passingArraysMultiDim.c:63:45: note: declaration of 'mat' as
multidimensional array must have bounds for all dimensions except
the first
```

Rank-n Arrays (3/5)

- What happens when we use a pointer parameter variable?

```
void printEntries( int nRows, int nCols, int** mat ) {
  for( int i = 0; i < nRows; i++ ) {
    for( int j = 0; j < nCols; j++ ) {
      printf( "%d ", mat[i][j] );
    }
    printf( "\n" );
  }
}
```

- now we try to pass our static array:

```
  int dVals[nRows][nCols];
  printEntries( nRows, nCols, dVals );
```

Rank-n Arrays (4/5)

We get a compiler warning

```
passingArraysMultiDim.c: In function ‘main’:
passingArraysMultiDim.c:48:31: warning: passing argument 3 of ‘printEntries’
from incompatible pointer type [-Wincompatible-pointer-types]
   printEntries( nRows, nCols, dVals );
                               ^~~~~
passingArraysMultiDim.c:12:6: note: expected ‘int **’ but argument is of type
‘int (*)[(sizetype)(nCols)]’
 void printEntries( int nRows, int nCols, int** mat ) {
      ^~~~~~~~~~~~
```

Result is garbage

```
-1 0 0 0 955216488
955216488 32766 -1351448816 5424 0
-1351448120 5424 0 0 0
Segmentation fault (core dumped)
```

Rank-n Arrays (5/5)

- If we use a dynamic array (variant #1 or #2) then we need to pass this as `datatype**`:

```
int main( void ) {
  int nRows = 5;
  int nCols = 5;
  int** mat = alloc2dArray( nRows, nCols );
  // setEntries( nRows, nCols, mat ); would fail
  printEntries( nRows, nCols, mat );
}
```

Structs (1/4)

- Consider this struct which wraps a static array

```
#define DIM 5
typedef struct {
  double values[DIM];
  int size;
} compound;
```

- We can pass it to a function as we do with a standard datatype

```
void func( compound cLoc ) {
  printf( "sizeof cLoc = %lu\n", sizeof( cLoc ) );
  cLoc.values[0] = -1.0;
  for( int i = 0; i < cLoc.size; i++ ) {
    printf( "%3.1f .. ", cLoc.values[i] );
  }
  printf( "\n" );
}
```

Structs (2/4)

- Program using our struct and function `func()`

```
int main( void ) {
  compound c = { {0}, DIM };  // define and initialise
  for( int i = 0; i < c.size; i++ ) {
    c.values[i] = (double)i;  // set c to new values
  }
  func( c );  // pass struct to function
  for( int i = 0; i < c.size; i++ ) {
    printf( "%3.1f .. ", c.values[i] );
  }
  printf( "\n" );
}
```

- Output we get:

```
sizeof cLoc = 48
-1.0 .. 1.0 .. 2.0 .. 3.0 .. 4.0 ..
0.0 .. 1.0 .. 2.0 .. 3.0 .. 4.0 ..
```

Structs (3/4)

- Now we change our struct to wrap a dynamic array:

```
typedef struct {
  double* values;
  int size;
} compound;
```

- We do not need to make changes to our function:

```
void func( compound cLoc ) {
  printf( "sizeof cLoc = %lu\n", sizeof( cLoc ) );
  cLoc.values[0] = -1.0;
  for( int i = 0; i < cLoc.size; i++ ) {
    printf( "%3.1f .. ", cLoc.values[i] );
  }
  printf( "\n" );
}
```

Structs (4/4)

- Our main program needs small modifications:

```
int main( void ) {
  compound c = { NULL, 0 };
  c.values = (double*) malloc( DIM * sizeof( double ) );
  c.size = DIM;
  [...]

}
```

- This time the output is different:

```
sizeof cLoc = 16
-1.0 .. 1.0 .. 2.0 .. 3.0 .. 4.0 ..
-1.0 .. 1.0 .. 2.0 .. 3.0 .. 4.0 ..
```

Summary: Structs

- The header `void func( compound cLoc )` defines a local variable of our structure datatype.
- Call-by-copy means that the members of the argument struct are copied into the local struct `cLoc`.
- Static Array Member:
  - all entries in static array are copied!
  - local changes do not affect struct/memory at caller
  - deep copy
- Dynamic Array Member:
  - address stored in pointer member is copied
  - local changes affect struct/memory at caller
  - shallow copy

Type Qualifiers

Wikipedia:

Type qualifiers are a way of expressing additional information about a value through the type system, and ensuring correctness in the use of the data.

As of C11 the language supports four so called type qualifiers which are

`const`	object is not changed by the program
`restrict`	no other pointer references object
`volatile`	object value may change between accesses
`_Atomic`	related to multi-threading

Const (1/2)

- Adding the `const` qualifier to an object in C is a constract between the programmer and the compiler. The programmer promises that their code will not change the object.
- Definining `const int a = 5;` does not mean, however, that the value stored in a may not change (only that we will not change it!)
- Example: Value from a special read-only register in an embedded system
- C behaviour is different from C++.

Const (2/2)

- We can add the const qualifier to formal parameters.

```
void printStatus( const int status ) {
  printf( "current status: %d\n", status );
  status = 0;  // breaking the contract!
}
```

```
gcc -Wall -Wextra constLocalVar.c
constLocalVar.c: In function 'printStatus':
constLocalVar.c:5:10: error: assignment of read-only
   parameter 'status'
   status = 0;  // breaking the contract!
          ^
```

- This get's especially interesting when working with pointers.

Const and Pointers (1/3)

- For pointers the question is: What is to be constant?
  - The address stored in the pointer variable?
  - That object at this address?
- Depends on what we request:

`int const a`	a is a constant `int`
`int const * a`	a is a pointer to a constant `int`
`int * const a`	a is a constant pointer to an int
`int const * const a`	a is a constant pointer to a constant int
`const int * a`	same as `int const * a`

Const and Pointers (2/3)

```
void Foo( int * ptr,
          int const * ptrToConst,
          int * const constPtr,
          int const * const constPtrToConst )
{
  *ptr = 0;     // OK: modifies the "pointee" data
  ptr  = NULL; // OK: modifies the pointer

  *ptrToConst = 0;     // Error! Cannot modify the "pointee" data
  ptrToConst  = NULL; // OK: modifies the pointer

  *constPtr = 0;     // OK: modifies the "pointee" data
  constPtr  = NULL; // Error! Cannot modify the pointer

  *constPtrToConst = 0;     // Error! Cannot modify "pointee" data
  constPtrToConst  = NULL; // Error! Cannot modify pointer
}
```

Const and Pointers (3/3)

- Using formal pointer parameters allows to change argument values from within the called function.
- If we do not plan to do this, we can use `const` to express our intent.
- The compiler will make sure that we do not fail; const-correctness
- Example:

```
void *memcpy(void *dest, const void *src, size_t n);
```

  from its prototype we see that `memcpy` will not change entries in the source buffer.
- const-correctness can quickly become difficult in the design of larger programs: What if `foo( const int *ptr )` passes `ptr` to a function that expects `int *ptr`? (C: warning; C++: error)

Aliasing (1/2)

```
// sum up values pointed to by a, b, c;
// return results at address a
void tripleSum( int *a, int *b, int *c )
{
  *a += *b;
  *a += *c;
}

int main( void ) {
  int i = 1, j = 1, k = 1, m = 1;

  tripleSum( &i, &j, &k );
  printf( "1 + 1 + 1 = %d\n", i );

  tripleSum( &m, &m, &m );
  printf( "1 + 1 + 1 = %d\n", m );
}
```

What will the code on the left print?

Aliasing (1/2)

```
// sum up values pointed to by a, b, c;
// return results at address a
void tripleSum( int *a, int *b, int *c )
{
  *a += *b;
  *a += *c;
}

int main( void ) {
  int i = 1, j = 1, k = 1, m = 1;

  tripleSum( &i, &j, &k );
  printf( "1 + 1 + 1 = %d\n", i );

  tripleSum( &m, &m, &m );
  printf( "1 + 1 + 1 = %d\n", m );
}
```

What will the code on the left print?

```
1 + 1 + 1 = 3
1 + 1 + 1 = 4
```

This is an example of aliasing:

two or more formal parameters refer to the same object or (overlapping) memory area.

Aliasing (2/2)

```
void test1( double *p, double*q, double *r ) {
  for( int k = 0; k < 256; k++ ) {
    r[k] = p[k] + q[k];
  }
}
```

Compiler must take aliasing into account; this might prohibit certain optimisations; if we are sure the there will be no aliasing we can make a promise to the compiler:

```
void test2( double * restrict p,
            double * restrict q,
            double * restrict r ) {
  for( int k = 0; k < 256; k++ ) {
    r[k] = p[k] + q[k];
```

Static Local Variables

- Local variables become undefined once they go out-of-scope.
- If we need to remember their contants between function invocations, we can mark them as `static` (a storage class modifier).
- Initialisation happens only on first invocation.

```
void printValue( void ) {
  static int a = 1;
  int b = 2;
  printf( "a = %d, b = %d\n", a++, b++ );
}

int main( void ) {
  for( int k = 1; k <= 5; k++ ) {
    printValue();
  }
}
```

```
a = 1, b = 2
a = 2, b = 2
a = 3, b = 2
a = 4, b = 2
a = 5, b = 2
```