7 Functions

Functions in C serve two purposes. First, functions define reusable program units. Second, functions implement local storage and provide for *recursion*.

Functions are usually construct to break a large computation into smaller blocks. Well constructed functions can hide details of implementation from parts of the program that do not need to know about them, improving usability and correctness of a complex code. Ideally, individual functions can be written and tested separately from the rest of the program.

Examples of functions we have used so far are: `main`, `printf`, `scanf`, and `sqrt`.

Once functions are written, they can be pre-compiled and stored in a library file for further use by any program. Your program interacts with functions through *prototypes* which are often stored in *header files*.

For example, the mathematical function `sqrt` has been pre-compiled with other functions and stored in the library file `libm.a`. Our programs interact with `sqrt` through its prototype which is located in the C source file `math.h`. We use the pre-processor directive `#include <math.h>` to access the prototype and the flag `-lm` is passed to the compiler to *link* our program with the actual code that implements `sqrt` and is stored in the file `libm.a`.

Functions in C are defined with the general syntax structure:

```
return_type
function_name (argument declarations) {

  declarations;
  statements;
  return expression;

}
```

It is no coincidence that this is the structure we have been using when writing our C programs, which so far are made of the single function `main`. Other functions will be defined in the exact same way.

Any valid C type can be a `return_type`, such as: int, float, double and void, as well as any pointers. The `return_type` can be ommited, in which case it is assumed to be `int`.

The `function_name` must be a valid C identifier, and will be used to execute the function anywhere in your programs.

Argument declaration, also known as formal parameter declaration, is a comma-separated list of `type` followed by a valid identifier, similar to what is used to declare a variable in the body of a function. The main difference here is that we need one type per variable identifier and that variables of different types are still separated by commma.

The keyword `return` is used for returning values to the statement in the program that called your function. The value will be copied to the caller. Only a single value can be returned by any C function. If you must return more than one parameter you will need to use a workaround, which we will study later.

So here is a complete example of a simple function:

```
double
my_first_function(double x, int n) {

  /* computes the function 1 / (1 + x^2)^n */

  int i;
  double y;

  /* compute 1 + x^2 */
  x *= x;
  x += 1;

  /* compute (1 + x^2)^n */

  y = 1;
  for (i = 0; i < n; i++)
    y *= x;

  /* return 1 / (1 + x^2)^n */

  return 1 / y;

}
```

In the above function, the arguments are the double `x` and the integer `n`. The function is named `my_first_function` and the function returns a double after performing some calculations.

The following simple program uses the above function, we say it *calls* or *makes*

a call to the function `my_first_function`:

```
main() {

  double x, y;

  x = 1.;
  y = my_first_function(x, 2);

  /* this will print x = 1 and y = 0.25 */
  printf("x = %f, y = %f\n", x, y);

}
```

In C, all function arguments (as well as return values) are passed *by value*. That is, a copy of the argument is made for use by the function being called and this copy is later destroyed when the function returns.

For example, the function `my_first_function` defined above modifies its argument `x` while performing its computation. This argument is, however, a *copy* of the variable `x` when it is called by the program `main`. That is, the variable `x` in the program `main` and in the function `my_first_function` are not the same, they are simply made to have the same value when the function is called. That is why the program `main` will print the unmodified value 1 even though `x` in `my_first_function` has been changed to the value 2 at the end of the computation.

Also important is the fact that the values of the arguments in a function only exist *during the execution of the function*. We say that they are defined *locally* to the function or *within the scope* of the function, that is they are *local parameters*. Hence, in the above example, it is impossible for the program `main` to access the parameter `x` which is used by the function `my_first_function`. We will see more on that later.

The same is true for any variable defined inside a function. In the above example, variables `i` and `y` are *local variables* that can only be accessed by the function `my_first_function`. They are created (memory is reserved) when the function is called and they are destroyed when the function returns. If the function is never called, they are never created.

The following is a complete example with the above function:

```
/* Program w7-1.c */
```

```c
/* Create and use a simple function */

#include <stdio.h>

double
my_first_function(double x, int n) {

  /* computes the function 1 / (1 + x^2)^n */

  int i;
  double y;

  /* compute 1 + x^2 */
  x *= x;
  x += 1;

  /* compute (1 + x^2)^n */

  y = 1;
  for (i = 0; i < n; i++)
    y *= x;

  /* return 1 / (1 + x^2)^n */

  return 1 / y;

}


main() {

  double x, y;

  x = 1.;
  y = my_first_function(x, 2);

  /* this will print x = 1 and y = 0.25 */
  printf("x = %f, y = %f\n", x, y);

}
```

A program can have as many functions as you like, and each function can have as many parameters as you like. You can create functions that take no arguments and/or that return no values. Your functions may also have more than one `return` statement, with the understanding that they will return immediatelly when the first `return` is found. The next program illustrate some

of these possibilities.

```c
/* Program w7-3.c  */

#include <stdio.h>

/* some functions */
int
function_with_no_arguments(void) {
  printf("This function takes no arguments.\n");
  return -1;
}

void
function_with_no_return_value(int a) {
  printf("This function takes argument a = '%d' but returns none.\n",
         a);
}

void
function_with_no_arguments_and_no_return_value(void) {
  printf("This function takes no argument and returns none.\n");
  return;
  printf("THIS IS NEVER EXECUTED!\n");
  return;
  printf("THIS IS NEVER EXECUTED!\n");
}

main() {

  int a;

  a = function_with_no_arguments();

  function_with_no_return_value(a);

  function_with_no_arguments_and_no_return_value();

}
```

7.1 Variable scope, global and local variables, and static storage

We now pay more attention to a feature mentioned earlier: *variable scope*. The scope of a variable refers to what parts of a program have access to that variable. For example, we have seen that variables declared within a function or a parameter are *local* to the function, that is, they exist only during the

execution of the function. In C, there are several types of construction that control variable scoping. The most general is *block scope*, in which a programmer can control the visibility of a program variable by creating a block delimited by braces '{}'. This is what is at work on a function. One may also *nest* blocks. Take a look at the following example:

```
/* Program w7-2.c */

/* Examples of block scopes */

#include <stdio.h>
#include <stdlib.h>

main() {

  /* this is the first block scope */
  int k = 1;

  printf("> k = %d\n", k++);

  {
    /* this is a second block scope */
    int k = 2;

    printf("> k = %d\n", k++);
    printf("> k = %d\n", k);

  }

  printf("> k = %d\n", k);

  exit(0);

}
```

It produces the following output:

```
> k = 1
> k = 2
> k = 3
> k = 2
```

Variables defined outside any functions are said to be within *program scope*. Those variables are known as *global variables* and they are visible to all functions. This can be seen at work in the program:

```
/* Program w7-2a.c */
```

```c
/* Examples of block and program scopes */

#include <stdio.h>
#include <stdlib.h>

/* global variable k */
int k;

void
set_k(void) {
  k = 7;
}

main() {

  /* this is the first block scope */
  k = 1;

  printf("> k = %d\n", k++);

  {
    /* this is a second block scope */
    int k = 2;

    printf("> k = %d\n", k++);
    printf("> k = %d\n", k);

  }

  printf("> k = %d\n", k);

  set_k();

  printf("> k = %d\n", k);

  exit(0);

}
```

which produces the following output:

```
> k = 1
> k = 2
> k = 3
> k = 2
> k = 7
```

WARNING: You should avoiding using global variables. Global variables complicate the understanding of a program and can hide severe bugs that will reveal themselves only in complex execution scenarios.

Variables can be declared with the modifier `static`, which instructs the compiler to not destroy a variable when it goes out of scope. A static variable will also be initialized only once. When it is accessed in its own scope for a second time, it will have the same value as when it left the scope the first time.

```
/* Program w7-12.c */

#include <stdio.h>

void
f(void) {
  static int i = 2; /* a static local integer */
  int j = 2; /* an automatic (default) local integer */

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

main() {

  /* the use of a static variable */
  int i;
  for (i = 0; i < 5; i++)
    f();

}
```

The output is:

```
i = 2, j = 2
i = 3, j = 2
i = 4, j = 2
i = 5, j = 2
i = 6, j = 2
```

Since `i` is a static integer, the previous value was retained on each call of `f`. By contrast, the (automatic) local integer `j` was reset to 2 each time `f` was executed.

WARNING: One should be very careful when using `static` variables. The reason for that is that many operating systems support *multithreading*, which allows many copies of the same function to be run simultaneously by a code. In such an environment, initialization of static variables become very tricky and

severe mistakes can be unintentionally produced by your code!

7.2 Passing arguments and retuning by value versus by reference

As we have noted before, all function arguments in C are passed *by value*, that is they are copied to a local variable which is visible only to the function. Other languages, FORTRAN being one of the examples, can pass only *references* as arguments to functions. Using such a mechanism, functions have access and can modify variables being passed as arguments. There are languages, C++ for instance, that explicitely allow both mechanisms and languages, Java the case in point, that adopt a mixed standard depending on the type of the variable.

Sometimes, passing references is the right thing to do. One example is when a function need to modify the value of their variables. Indeed, we have used the following construction before:

```
double d;
scanf("%lf", &d);
```

to parse keyboard input into the `double` variable `d`. Clearly `scanf` needs to modify the contents of `d`! In other words, you need a *reference* to `d` to be passed to `scanf`, not a *copy* of `d`. In the above example, this puzzle has been solved by passing a *pointer* by value. That is, `scanf` gets a copy of a pointer to `d`. Using the pointer it can then modify the contents of `d`.

Another situation is when a large amount of data is to be accessed but not necesarily modified by a function. In this case, passing arguments by value can create unecessary copies that will be immediately destroyed after the function returns. This takes extra time and valuable memory space. Take a look at these statements:

```
char text[] = "This is some large text I want to print.\n";
printf(text);
```

Once again, by using pointers, we have effectively passed a *reference* to the character string `text`. The argument of `printf` is `text` which is of type `char*`, that is a pointer. Here the *pointer is being passed by value*, and therefore being copied, but the contents of the array pointed by it are not being copied. No mather how large the text is, a single copy of a pointer will be performed.

A downside of the above technique is that `printf`, having a pointer to the variable `text`, might decide to modify the contents of `text`. In order to

distinguish between functions that modify their arguments and functions that do not, you can use the qualifier `const`. The use of this qualifier is relatively new to C, and has been borrowed from C++. If you look for the prototype[4] of function `printf` in the file `stdio.h` you will find something like:

```
int printf(const char * text, ...);
```

where the `const char *` type means that `printf` will be passed a pointer to char but that pointer cannot be used to modify the contents of the character string `text`.

As a matter of fact, you can, and indeed should, use the `const` qualifier whenever it applies. For example, in program `w7-3.c`, redefining the function

```
void
function_with_no_return_value(const int a) {
  printf("This function takes argument a = '%d' but returns none.\n",
  a);
}
```

to use the `const` qualifier might allow the compiler to optimize its execution. As the compiler now *knows* that the value of a will not be modified by this function, it can decide not to create a copy of a after all when calling this function.

When dealing with arrays, passing by reference is the mechanism of choice. As we have seen in Section 5.3, arrays and pointers are often intercheangeable in C. For example, consider the function

```
void array_set(const double d, const int n, double * a) {
  int i;

  for (i = 0; i < n; i++)
    a[i] = d;
}
```

which can be used to set all elements of an array of `doubles` to the constant value `d`. The following fragment

```
double array[20];
array_set(-1., 20, array);
```

calls the function `array_set` to set all entries of the array `array` to '-1'. Here the entire array is passed by reference. Note that because of the way arrays are organized, all that is needed is a reference to the first element of the array and

[4] More on that in the next section.

the number of elements. Note also that the last argument of `array_set`, the `double * a`, cannot be `const`, as this would prevent the function from modifying the contents of the array.

Unfortunately, the mechanism illustrated above may sometimes not be generic enough. For example consider, a function `resize_array` which takes an array and dynamically resizes it[5], and sets all its elements to a given double `d`. Here is a first attempt to write such a function:

```
void bad_array_resize(const double d, const int n, double * a) {

  /* release current storage */
  if ( a != NULL )
    free(a);

  /* allocate new array */
  a = (double*) malloc(n * sizeof(double));

  /* set entries to zero */
  for (i = 0; i < n; i++)
    a[i] = d;

}
```

See Section 7.6 for more details on the functions `malloc` and `free`. While one might at first be tempted to believe that a call to `bad_array_resize` as in the fragment:

```
double *array = NULL;
bad_array_resize(-1., 20, array);
```

would properly resize the array, a closer look at the passage mechanism by value used in C shall reveal its limitations. The problem here is that the pointer variable '`double * a`' in `bad_array_resize` is itself passed by value. When `bad_array_resize` is called in the above fragment, a new variable '`a`' is created. This variable '`a`' is local to the function `bad_array_resize` and contains a copy of the pointer '`array`'. It is this local variable '`a`' that gets set to the newly allocated array space while the original variable '`array`' remains, in this case, pointing to `NULL`. When `bad_array_resize` returns, '`a`' is destroyed and its value is never copied back to '`array`'. Indeed, a call to `bad_array_resize` will produce a memory leak, as a subsequent call to `free(array)` will not be able to release the memory allocated in the function

[5] Much more on dynamic memory allocation in Section 7.6.

bad_array_resize.

One possible solution here is to pass by reference not the pointer to first element of the array, but a pointer to the pointer of the first element of the array. The following function:

```
void good_array_resize(const double d, const int n, double ** a) {

  int i;

  /* release current storage */
  if ( *a != NULL )
    free(*a);

  /* allocate new array */
  *a = (double*) malloc(n * sizeof(double));

  /* set entries to zero */
  for (i = 0; i < n; i++)
    (*a)[i] = d;

}
```

does just that. A call to:

```
double *array = NULL;
good_array_resize(-1., 20, &array);
```

would now properly free the existing array `array` and set `array` to the newly allocated storage. Note that 'a' now contains a pointer to the pointer variable 'array', as opposed to a pointer to the first element of the array 'array' as in the previous version of this function. When 'a' is dereferenced in good_array_resize, that is when '(*a)' is used, the contents of the pointer variable 'array' are directly accessed, and not just a duplicate copy as in bad_array_resize. The price to be paid here is that in order to access the elements of the array we need to dereference 'a' twice. If 'a' is a pointer to the pointer variable 'array', then '(*a)' is now a pointer to the first element of 'array', i.e. '(*a) == array', and therefore the statement '*(*a)', or simply '**a', accesses the first element of the array because '*(*a) == *array = array[0]' . The function good_array_resize performs such double dereferencing operation when zeroing the entries of the array. Indeed, the statement '(*a)[i]' is equal to 'array[i]'. Note here the importance of the use of parenthesis. Because of precedence rules, the statement '*a[i]' is interpreted as '*(a[i])' which is not quite the same as

the desired '(*a)[i]'!

A similar mechanism can be used to return values. As in the case of arguments, return values are always passed by value. Whereas one can emulate passing return values by reference using pointers, you should be extremely careful here to ensure that the pointer is valid even after the function returns. The function below is an example of code with potentially disastrous consequences:

```
char *
bad_function(double d) {
  char buffer[120];

  sprintf(buffer, "Here is number %g.", d);

  return buffer;
}
```

The problem with this code is that the variable `buffer` is local to the function `bad_function` and will be destroyed after it returns. Therefore, any pointer to `buffer` used outside the function `bad_function` is invalid!

The following similar code is valid though:

```
char *
good_function(double d) {
  static char buffer[120];

  sprintf(buffer, "Here is number %g.", d);

  return buffer;
}
```

Note now the use of the modifier `static` that prevents the variable `buffer` from ever destroyed after `good_function` returns.

Another valid possibility is:

```
char *
good_function(char * buffer, double d) {
  sprintf(buffer, "Here is number %g.", d);
  return buffer;
}
```

which transfer the responsibility of dealing with the storage of buffer to the caller of `good_function`.

A fourth possibilty is for the user to dynamically create a variable inside the function that will not be automatically destroyed by the compiler. This is

precisely what has been done in the functions `bad_array_resize` and `good_array_resize` and we will study in more detail later in Section 7.6. As a matter of fact, an alternative to the pointer-to-pointer solution adopted in `good_array_resize` would be to return by reference. Indeed, the function:

```
double * array_resize(const double d, const int n, double * a) {

  int i;

  /* release current storage */
  if ( a != NULL )
    free(a);

  /* allocate new array */
  a = (double*) malloc(n * sizeof(double));

  /* set entries to zero */
  for (i = 0; i < n; i++)
    a[i] = d;

  return a;

}
```

will work fine as long as the user is disciplined enough to always duplicate the array argument as in the fragment:

```
double *array = NULL;
array = array_resize(-1., 20, array);
```

Bear in mind that such construction is however prone to errors as the user might inadvertently assign the return value to another pointer variable or simply forget to assign it at all without a warning from the compiler.

7.3 Recursion

Functions in C can call themselves, hence allowing for *recursion*. Many algorithms admit both a loop-type construction as well as a recursive construction. For example, consider the following naive algorithm for computing positive integer powers of a number based on our discussion in Section 4.7:

```
double
power(double x, int n) {

  int i = 0;
  double y = 1.;
```

```
  for (i = 1; i <= n; i++)
    y *= x;

 return y;

}
```

A recursive version of the above algorithm can be constructed by noticing that the same result can be obtained through the recursion:

$$y_0 = 1, \qquad\qquad y_n = x\, y_{n-1}.$$

A function that implements the above recursion is the elegant:

```
double
power(double x, int n) {

  if (n == 0)
    return 1.;

  return x * power(x, n - 1);

}
```

Note however that elegance often comes up at a price. In this case, each time the function is called, copies of `x` and `n` must be created, whereas in the first version a single copy and a couple of local variables is all that was needed. Temporary variables are also created by the return statement.

All such operations consume more memory and take more time. Note that this implies that an "infinite recursion", that is, a recursion that never terminates, will end up consuming the entire finite computer memory, wereas an "infinite loop" will simply not terminate. An extra complication is that local variables and parameters are created in a dedicated part of the memory called the *stack* and this segment of the memory, often of small size as compared with today's computer memory, can be exhausted very fast.

However, recursion plays an important its role in the theory of computing. It is know that all looping-type constructions admit a recursive implementation but the converse is false. That is, there are problems for which a solution is inherently recursive! These algorithms require an ever growing amount of information to be stored in order to monitor its progress, a task which, although could be emulated by a programer using dynamic memory allocation (to be

discussed later in this section), is better executed by a compiler. One example of an algorithm which we will use (but not study) that is inherently recursive is a sorting algorithm known as *quicksort*, which is implemented in the `stdlib`.

We finallize the discussion of recursion with a classic algorithm for searching sorted lists known as *binary search* which is often presented in recursive form. Our list here will be represented by a `double` array of size `n`. The following function finds the `double x` in the sorted array:

```
/* search.c */

/*
 * implements the binary search function
 * for double arrays
 */

int
binary_search(const double x, const double *list,
              const int start, const int end) {

  int mid_point;

  if ( start > end)
    return -1;

  /* compute mid-point using integer division */
  mid_point = start + (end - start) / 2;

  if (list[mid_point] == x)
    /* found x */
    return mid_point;

  else if (list[mid_point] > x)
    /* search lower half */
    return binary_search(x, list, start, mid_point - 1);

  else
    /* search upper half */
    return binary_search(x, list, mid_point + 1, end);

}

int
bbsearch(const double x, const double *list, const int n) {

  /* call recursive binary search algorithm */
  return binary_search(x, list, 0, n - 1);
```

```
}
```

A user calls the function `bbsearch`. If succesful, it returns the index of the located value in the array. If unsucessful, it returns '`-1`'.

Note that use of the `const` qualifier in order to ensure that the function `bbsearch` does not modify the array it is searching.

7.4 Multiple source files and function prototyping

Now that we learned how to write reusable code, it will often be the case that a set of functions will be developed, stored and tested as a separate set of source files. These files will later be combined with other sources files that call these functions to perform specific tasks. Consider for example the function `bbsearch` that we discussed in the last section. It is stored in the file `search.c`. This file does not contain a function `main`, so it will not generate an executable program.

Consider now the program that *calls* the function `bbsearch`:

```
/* Program w7-20.c */

/* This program uses the bbsearch function defined in search.c */

#include <stdio.h>

int
main() {

  const int N1 = 7;
  double array1[] = {-2.3, -1.2, -.5, 3.7, 10., 15.3, 27.};

  const int N2 = 7;
  double array2[] = {3.7, 27., -2.3, 0., 1., 30., -10.};

  int i, j;

  /* call function bbsearch */
  for (i = 0; i < N2; i++) {

    j = bbsearch(array2[i], array1, N1);

    if ( j < 0 )
      printf("'%f' is not an element of the array!\n", array2[i]);
```

```
    else
      printf("Found '%f' at position '%d'!\n", array2[i], j);

  }

  return 0;

}
```

If we compile `w7-20.c` we get a linking error during compilation:

```
iacs5.ucsd.edu% gcc w7-20.c
Undefined symbols:
  "_bbsearch", referenced from:
      _main in ccuDZ7Mv.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

What went wrong here is that the function `bbsearch` that is used at the source code `w7-20.c` is not defined in that same file, nor it is available as part of the standard C libraries. In fact, it is defined in another file (`search.c`), and we need to inform gcc about that. One simple (and not the best) way to do this is to compile `search.c` *together* with `w7-20.c`. This can be done by simply adding `search.c` to the list of files passed to `gcc`:

```
iacs5.ucsd.edu% gcc search.c w7-20.c
iacs5.ucsd.edu% ./a.out
Found '3.700000' at position '3'!
Found '27.000000' at position '6'!
Found '-2.300000' at position '0'!
'0.000000' is not an element of the array!
'1.000000' is not an element of the array!
'30.000000' is not an element of the array!
'-10.000000' is not an element of the array!
```

Unfortunately life is not that simple, and the above "trick" would not really work if we were using any function that did not return an `int`. Do you remember that C assumes that any function returns an `int` unless defined otherwise? This is essentially what makes it work in this case! Without further ado, let's take a look at a better way to handle the potential problems here. First let us have `gcc` help us by turning on all warnings:

```
iacs5.ucsd.edu% gcc -Wall w7-20.c
w7-20.c: In function 'main':
w7-20.c:21: warning: implicit declaration of function 'bbsearch'
Undefined symbols:
  "_bbsearch", referenced from:
```

```
      _main in ccuDZ7Mv.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

Now it is clear what happened: besides the fact that `bbsearch` is not implemented anywhere, `gcc` tells you that `w7-20.c` does not know anything about it. The default behaviour is then to implicitely assumes the parameters of the function `bbsearch` are as called in `w7-20.c` and, more important in this case, that it returns an `int`. Note that this is, unfortunately, not an *error* but simply a *warning* (there are deeper reasons for that!).

Adding `search.c` to the list of files to be compiled seems to fix the *implementation* problem. However, it does not solve the *declaration* problem:

```
iacs5.ucsd.edu% gcc -Wall search.c w7-20.c
w7-20.c: In function 'main':
w7-20.c:21: warning: implicit declaration of function 'bbsearch'
```

When compiling `w7-20.c` jointly with `search.c`, the function `bbsearch` *remains implicitly declared*! The reason for this, as we will see in the next section, is that `search.c` and `w7-20.c` *are still being compiled separately*. The only step that is done jointly is *linking*.

We solve the declaration problem by creating a *function prototype*. A prototype tells the compiler about the arguments and return types of a function without implementing the function. The first couple line of the file:

```
/* Program w7-20a.c */

/* This program uses the bbsearch function defined in search.c */

#include <stdio.h>

/* prototype for bbsearch */
int
bbsearch(const double, const double *, const int);

int
main() {

  const int N1 = 7;
```

which is otherwise identical to `w7-20.c`, contain a prototype for the function `bbsearch`. The declaration of the prototype clears the warning and would have fixed the problem with the return type if it were different than `int`. Note that

the names of the arguments are not needed in a function prototype. If present, they are ignored. For example:

```
int bbsearch(double y, double *list, int n);
```

is a perfectly fine prototype for function `bbsearch`. Furthermore, as the argument names are ignored, they do not need to match the ones actually used in `search.c`.

7.5 Header and library files

As we saw on the last sections, function prototypes are required anytime we do not implement a function directly in our code. This is always the case, for instance, when we use functions bundled in libraries. For example, the functions `printf` and `sqrt` that we have used are implemented in a *library*, the standard C library (no flag required at compilation) and the `libm.a` library (flag `-lm` required at compilation). In addition, we have been informing the compiler about their prototypes by using the `#include` pre-processor directive to include the *header files* `stdio.h` and `math.h`. In this section we will learn more about header files and library files. We start by header files.

Header files are standard C source files, often with the extension `.h`, which can contain macros, variable and function declarations. Even though one could use header files to implement functions, in C, this is almost never the case. In C, most of the function implementation is done in source code, which may or may not have been pre-compiled into a library as we will see later. Typically header files contain mostly function prototypes, global variables declarations and macros. For example, the header file `math.h` contains two lines similar to the ones:

```
#define M_PI        3.14159265358979323846264338327950288   /* pi */

extern double sqrt( double );
```

The first line defines a macro that we can use as π in the largest available floating point type. The other line define a prototype for the function `sqrt` that takes a `double` argument. Note the presence of the keyword `extern` which tells the compiler that such functions should be available for use by all files being compiled and not only the one in which these functions have been implemented.

Header files are often more than just a couple of prototypes. Here is a practical

header file that can be used to provide a prototype to the function `bbsearch`:

```
/* bbsearch.h */

#ifndef _BB_SEARCH_H_
#define _BB_SEARCH_H_

/* declare prototype for function bbsearch */

extern
int
bbsearch(const double, const double *, const int);

#endif /* _BB_SEARCH_H_ */
```

The only surprise here is the use of the pre-processor directive `#ifndef` which allows for the prototype to be declared only once even if the file `bbsearch.h` gets included multiple times, for example, through a series of nested include statements. Do you see how this could throw the compiler in an infinite loop? or generate multiple conflicting declarations of global variables?

Now that we have a header file, the declaration of function `bbsearch` is now encapsulated in the header and the program `w7-20a.c` can be modified to read:

```
/* Program w7-20b.c */

#include <stdio.h>

/* include header file for function bbsearch */
#include "bbsearch.h"

int
main() {

  const int N1 = 7;
```

Now that the declaration is properly done in the header file, we can also think of precompiling the function `bbsearch` and use the compiled code, instead of freshly compiling a new version every time we need to use it. For that we use the switch `-c` to compile a file without generating an executable:

```
iacs5.ucsd.edu% gcc -Wall -c bbsearch.c
```

This operation produces the *object file* `bbsearch.o`, which contains a compiled version of our function `bbsearch`. If we want to use it our program we can simply add the object file, as opposed to the source file, to the arguments of `gcc`:

```
iacs5.ucsd.edu% gcc -Wall w20-7c.c bbsearch.o
```

We can take one more step and build a library file that contain our compiled functions. In order to generate a library file you should use

```
iacs5.ucsd.edu% ar cr libbbsearch.a bbsearch.o
```

This bundles all functions compiled in `bbsearch.o`[6] with the extern attribute into the *library* file `libbbsearch.a`.

After creating a library file, the function `bbsearch` is accessible to any C program that links with the library `libbbsearch.a`. This can be done as for standard C libraries, such as the mathematical library that we used quite a bit, through the flag `-l`. The following command will compile `w7-20c.c` and link it with the library `libbbsearch.a`:

```
iacs5.ucsd.edu% gcc -Wall w20-7c.c -L. -lbbsearch
```

The flag `-L.` instruct the compiler to search for library files in the current directory (`.`) as well as on the standard library locations, such as `/usr/lib`, `/usr/local/lib`, etc.

7.6 Dynamic memory allocation

In the previous sections we have seen how the compiler will allocate memory to create storage for local variables within different scopes, such as in program blocks and functions. The mechanism used by the compiler is made available to the user through the functions `malloc`, `calloc` and `free` which are part of `stdlib.h`. These functions allow for the user to manage the memory *dynamically*, that is depending on the state of the variables of a program. The main drawback is that once memory is allocated, say using `malloc` or `calloc`, it is not freed automatically by the compiler: the user is responsible for calling `free` to release the memory for other purposes. Combine this with sloppy programming and you have a great recipe for so called *memory leaks* that plage many large programming projects.

We will introduce the idea of dynamic memory allocation by looking at an example. Consider the program `w6-3.c` in Section 6.2, which reads pairs of doubles stored in the file `input.dat`. Here is the file for a refresh:

```
5
```

[6] Jointly with any other object files passed in the command line.

```
12. 1.
3. -3.
45. 4.
7. -6.
8. 0.
```

The first entry of the file is an integer which tells you the number of lines to come which, in this example, will come in pairs. Program `w6-3.c` reads the file and convert each pair or cartesian coordinates to polar coordinates, which are saved in the file `output.dat`. We will now do the same thing, except that we will write the entries in the output files in reverse order. In order to do that we will first store the pairs of two-dimensional vectors into two one-dimensional arrays. The problem here is that the size of the array is now known until you actually start reading the file! The solution we will adopt here is to allocate the array dynamically, right after the number or lines is read from the file.

Suppose we have just read the first line of the file and stored the number of lines in the integer `nlimes`. Then the following call to `malloc` allocates memory enough to hold exactly `nlines` doubles for each of the arrays.

```
x_array = (double *) malloc(nlines * sizeof(double));
y_array = (double *) malloc(nlines * sizeof(double));
```

The function `malloc` takes as argument the number of bytes to be alocated, and returns a `void*` pointer to n bytes of uninitialized storage. If memory cannot be allocated it returns `NULL`.

Note that the `void*` pointer returned by `malloc` has to be cast into an appropriate type. In our example, to a `double *`, as the memory will be accessed as a one-dimensional array.

As mentioned earlier, after we are done using the memory space allocated dynamically, we have to use the function `free` to release the memory for other uses. In our examples we will call:

```
free(x_array);
free(y_array);
```

Alternatively we could have used the function `calloc` to allocate memory. This function takes two arguments: the number of elements to allocate and the size of each element in bytes. In our case we would have called

```
x_array = (double *) calloc(nlines, sizeof(double));
y_array = (double *) calloc(nlines, sizeof(double));
```

Just like `malloc`, `calloc` returns a `void *` pointer that is equal to `NULL` in case of failure.

Here is the complete program `w6-3a.c` rewritten to use dynamic memory allocation:

```
/* Program w6-3a.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main() {

  /* declare variables */
  int nlines, i;
  double x, y, r, theta;
  FILE *fp1, *fp2;

  double *x_array, *y_array;

  if ( (fp1 = fopen("input.dat", "r")) == NULL ) {
    printf("Error in opening input file\n");
    exit(1);
  }

  if ( (fp2 = fopen("output.dat", "w")) == NULL ) {
    printf("Error in opening output file\n");
    exit(1);
  }

  /* print message */
  printf("Processing input file...\n");

  /* read number of lines */
  fscanf(fp1, "%d", &nlines);

  /* allocate memory space for arrays */
  x_array = (double *) malloc(nlines * sizeof(double));
  y_array = (double *) malloc(nlines * sizeof(double));

  /* read remaining lines with pairs of doubles */
  i = 0;
  while ( i < nlines && fscanf(fp1, "%lf %lf", &x, &y) != EOF ) {

    /* convert to polar and store it on array */
    x_array[i] = sqrt(x * x + y * y);
    y_array[i] = atan2(y, x);
```

```
    /* increment line counter */
    i++;

  }

  /* write pairs on the output in reverse order */
  fprintf(fp2, "%d\n", nlines);

  for ( i = nlines - 1; i >= 0; i-- ) {

    /* write it on the output */
    fprintf(fp2, "%g %g\n", x_array[i], y_array[i]);

  }

  /* close files */
  fclose(fp1);
  fclose(fp2);

  /* free memory allocated to arrays */
  free(x_array);
  free(y_array);

  /* print message */
  printf("Done!\n");

  exit(0);

}
```

7.7 Pointers to functions

A function has an associated memory address, just like any regular variable. This memory address is related to the *entry-point* of the function, that is the point where the first instruction of the function is stored. Therefore, it is possible to create *pointers to functions*. These are useful, for instance, in generic algorithms where specific functions are plugged in based on a choice of the user or simply will be implemented later. The syntax is similar to the one used to create pointers to variables, except that a function prototype-like structure is used to define the pointer variable. For example, start with the prototype of the function:

```
  double power(double x, int n);
```

we have implemented earlier. A pointer `fptr` to a function of this type can be declared by simply adding $*$ to the function name and enclosing it in parenthesis:

```
double (*fptr)(double x, int n);
```

The pointer to function `fptr` is and behaves like a regular variable. Hence its value can be assigned during the program using the construction

```
fptr = &power;
```

Of course, a value can also be assigned during declaration, for example:

```
double (*fptr)(double x, int n) = &power;
```

Once a value has been assigned to `fptr` it can be used like a regular pointer, by dereferencing using the operator $*$. Therefore

```
fptr = &power;
(*fptr)(3.4, 2);
```

will compute 3.4^2.
It is common practice to use the keyword `typedef` in order to be able to use a function pointer as a regular native C type. For example:

```
double (*fptr1)(double x, int n);
double (*fptr2)(double x, int n);
```

is the same as

```
typedef double (*function_ptr)(double x, int n);
function_ptr fptr1, fptr2;
```

Using that construction we can define more complicated things, such as an array of pointers to functions. For example:

```
typedef double (*function_ptr)(double x, int n);
function_ptr fptr_array[10];
```

declares an array `fptr_array` of pointers to functions with 10 elements.

Because pointers to functions are declared for a *family of functions* that share the same prototype, a single pointer can be used to call a number of different functions. For example, the simple mathematical functions in `math.h`:

```
double fabs(double);
double sqrt(double);
double exp(double);
double log(double);
double sin(double);
double cos(double);
```

all share the same prototype. Using this fact, the following code allows the user to input a number and select a function to be computed using pointers to functions:

```c
/* Program w7-30.c */

/* This program uses an array of pointers to function to implement a
   simple calculator */

#include <stdio.h>
#include <math.h>

/* declares a new type */
typedef double (*fptr)(double);

main() {

  const int N = 6;
  char *labels[] = {"fabs", "sqrt", "exp", "log", "sin", "cos"};
  fptr functions[] = {&fabs, &sqrt, &exp, &log, &sin, &cos};
  double x, y;
  int i, n;

  n = 1;
  while ( n != 0 ) {

    /* ask for a number */
    printf("> Enter a number: ");
    scanf("%lf", &x);

    while ( 1 ) {

      /* ask for a function */
      printf("> Choose a function:\n"
             "  0. exit\t");
      for (i = 1; i <= N; i++) {
        printf("  %d. %s\t", i, labels[i - 1]);
        if ( (i + 1) % 2 == 0 )
          printf("\n");
      }
      printf("\n> ");

      scanf("%d", &n);

      if ( n < 0 || n > 6 )
        printf("Error: Invalid choice '%d'!\n", n);
      else
        break;
```

```
    }

    /* evaluate function */
    if ( n > 0 ) {
      y = functions[n - 1](x);
      printf("> %s(%g) = %g\n", labels[n - 1], x, y);
    }

  }

}
```

We finish this discussion by illustrating how to use the sorting function `qsort` defined in `stdlib.h`. This function implements a generic sorting algorihtm. Its prototype is:

```
  void qsort(void * array,
             size_t num, size_t size,
             int (*comparator) (const void *, const void *) );
```

where `array` is a pointer to the first element of the array to be sorted, `num` is the number of elements in the array and `size` is the size in bytes of each element in the array.

The fourth parameter is a pointer to a function. This function is used to compares two elements. It must accept two parameters that are pointers to elements, type-casted as void*. These parameters should be internally cast to the appropriate data type before they are compared. The return value is a negative value, zero or a positive value meaning that the first element is less than, equal to, or greater than the second element, respectively. The following program uses the function `qsort` to sort an array of integers:

```
/* Program w7-31.c */

/* This program uses qsort to sort an array of integers */

#include <stdio.h>
#include <stdlib.h>

int comp_ascending(const void * a, const void * b) {
  return ( *((int*) a) - *((int*) b) );
}

int comp_descending(const void * a, const void * b) {
  return ( *((int*) b) - *((int*) a) );
```

```
}

int main () {

  const int N = 10;
  int array[] = { 3, -2, 5, 8, 2, 1, 10, 4, -3, 0 };
  int i;

  /* print unsorted array */
  printf ("unsorted  : { ");
  printf ("%d", array[0]);
  for (i = 1; i < N; i++)
    printf (", %d", array[i]);
  printf ("}\n");

  /* sort in ascending order */
  qsort(array, N, sizeof(int), comp_ascending);

  /* print sorted array */
  printf ("ascending : { ");
  printf ("%d", array[0]);
  for (i = 1; i < N; i++)
    printf (", %d", array[i]);
  printf ("}\n");

  /* sort in descending order */
  qsort(array, N, sizeof(int), comp_descending);

  /* print sorted array */
  printf ("descending: { ");
  printf ("%d", array[0]);
  for (i = 1; i < N; i++)
    printf (", %d", array[i]);
  printf ("}\n");

  return 0;
}
```

This program produces as output:

```
unsorted  : { 3, -2, 5, 8, 2, 1, 10, 4, -3, 0}
ascending : { -3, -2, 0, 1, 2, 3, 4, 5, 8, 10}
descending: { 10, 8, 5, 4, 3, 2, 1, 0, -2, -3}
```