Procedural programming with C

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
cc@cse.concordia.ca

August 23, 2018

Functions

- ► Similarly to its mathematical counterpart, a computing *function* is a (named) block that normally receives some input, performs some task and normally returns a result.

- ► Unlike its mathematical counterpart, a computing function may receive no input or may produce no output.

- ► A function call implies transfer of control of execution to the function. When a function completes its task and terminates, control of execution is transferred back to the client.

- ► Synonyms for *function* exist in various languages (such as *method*, *procedure*, or *subroutine*). It is also important to note that some languages make a distinction between functions that return a result and those that do not, the latter ones being referred to as *procedures*.

Functions /cont.

- The general form of a function definition in C is

  `return-type function-name ( parameter-list ) { body }`

  where *return-type* is the type of the value that the function returns, *function-name* is the name of the function, and *parameter-list* is the list of parameters that the function takes, defined as

  `( type1 parameter1, type2 parameter2, ... )`

Functions /cont.

- If no type is in front of a variable in the parameter list, then int is assumed. Finally, the body of the function is a sequence of statements.
- If the function will be accessed before it is defined, then we must let the compiler know about the function by defining the function's *prototype* (or *declaration*) as follows:

  `return-type function-name (parameter-type-list);`

  where *return-type* and *function-name* must correspond to the function definition.
- The *parameter-type-list* is a list of the types of the function's parameters. Function main() requires no prototype.

Recursion: An example

- ▶ C supports recursion, i.e. like its mathematical counterpart a function in C can call itself.
- ▶ The program below computes the factorial of a non-negative integer. It is composed by two functions: main() and factorial(..).

```c
#include<stdio.h>
long factorial(int);
int main() {
  ...
}
long factorial(int n) {
  ...
}
```

Recursion: An example /cont.

```c
#include<stdio.h>
long factorial(int);
int main() {
  int n;
  long f;
  printf("Enter an non-negative integer: ");
  scanf("%d", &n);
  if (n < 0)
    printf("Negative integers are not allowed.\n");
  else {
    f = factorial(n);
    printf("%d! = %ld\n", n, f);
  }
  return 0; }
long factorial(int n) {
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1)); }
```

Recursion: An example /cont.

- The statement

  ```
  long factorial(int);
  ```

  defines the prototype for function `factorial`.

- The code below

  ```
  long factorial(int n) {
    ...
  }
  ```

  is the actual function definition.

- In C, execution always starts from `main()` which calls all other functions, directly or indirectly.

Recursion: An example /cont.

- In `long factorial(int n) ..`, n is called the *parameter* (or *formal argument*, also *dummy argument*) of the function.
- The result of a function is called its *return value* and the data type of the return value is called the function's *return type*.
- The return type of `main()` is int (integer), whereas the one of factorial is `long` (long integer).
- A *function call* allows us to use a function in an expression.

Recursion: An example /cont.

▶ In the statement

```
f = factorial(n);
```

the function on the right-hand-side executes and the value that it returns is assigned to the variable f on the left-hand-side. We refer to n as the *actual argument* (or just *argument* if the distinction is clear from the context).

▶ Obviously the actual argument(s) to a function would normally vary from one call to another.

▶ The values of the actual arguments are copied into the formal parameters, with a correspondence in number and type.

▶ Let us execute the program:

```
Enter an non-negative integer: 5
5! = 120
```

Example: Fahrenheit to Celcius conversion

▶ In this example we are writing a program whose main function requests and receives the value of the fahrenheit temperature to be converted to celsius and proceeds to call function f2c() to perform this conversion.

▶ The function f2c() will apply the conversion formula and return its result to the caller (function main()).

```c
int main (void) {
  int fahrenheit;
  printf("Enter the temperature in degrees fahrenheit: ");
  scanf("%d", &fahrenheit);
  printf ("The corresponding temperature in celsius is %d\n",
                                        f2c(fahrenheit));
  return 0; }
int f2c (int fahrenheit) {
  int celsius;
  celsius = (5.0/9.0) * (fahrenheit-32);
  return celsius; }
```

Example: Fahrenheit to Celcius conversion /cont.

▶ Let us execute the program:

```
Enter the temperature in degrees fahrenheit: 30
The corresponding temperature in celsius is -1
```

Global and local variables

▶ We distinguish between global and local variables.

▶ A global variable is defined at the top of the program file and can be accessed by all functions.

▶ A local variable is accessed only within the function which it is declared, called the scope of the variable.

▶ Though not a good programming practice, in the case where the same name is used for a global and local variable then the local variable takes preference within its scope.

▶ This is referred to as *shadowing*.

▶ Global variables have default initializations, whereas local variables do not.

Global and local variables /cont.

▶ Consider the following program that contains a global and a local variable with the same name.

▶ Within function func(..) the global variable a is not visible as the local variable a takes precedence.

```c
#include<stdio.h>
int a = 3;
int func() {
  int a = 5;
  return a;
}
int main() {
  printf("From main: %d\n", a);
  printf("From func: %d\n", func());
  printf("From main: %d\n", a);
}
```

▶ The output is as follows:

```
From main: 3
From func: 5
From main: 3
```

Variable and function modifiers

▶ Two modifiers are used to explicitly indicate the visibility of a variable or function: The `extern` modifier indicates that a variable or function is defined outside the current file, whereas the `static` modifier indicates that the variable or function is visible only from within the file it is defined in.

▶ The default (i.e. no modifier) indicates that the variable or function is defined in the current file and it is visible in other files.

Variable and function modifiers: Summary

- extern: Variable/function is defined outside of current file.
- (blank): Variable/function is defined in current file and visible outside.
- static: Variable/function is visible only in current file.

Example: Sorting

▶ Consider a program that reads in a collection of elements and proceeds to sort them by calling a function bubbleSort() that is defined outside the current file and thus must be declared extern.

```c
#include <stdio.h>
extern void bubbleSort(int[], int);
int main() {
  int array[10], numberOfElements;
  printf("Enter number of elements: ");
  scanf("%d", &numberOfElements);
  printf("Enter %d integers: ", numberOfElements);
  for (int i = 0; i < numberOfElements; i++)
    scanf("%d", &array[i]);
  bubbleSort(array, numberOfElements);
  printf("Sorted list (ascending order): ");
  for (int i = 0 ; i < numberOfElements ; i++)
    printf("%d ", array[i]);
  return 0;
}
```

Example: Sorting /cont.

▶ Function `bubbleSort()`, defined in some other file, makes use of function `swap()` that swaps two elements.

▶ As function `swap()` need not be visible outside the file in which it is defined, it is declared `static`:

```
static void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp; }


void bubbleSort(int numbers[], int array_size) {
  int i, j;
  for (i = (array_size - 1); i > 0; i--) {
    for (j = 1; j <= i; j++) {
      if (numbers[j-1] > numbers[j])
        swap(&numbers[j-1], &numbers[j]); }}}
```

Example: Sorting /cont.

```
Enter number of elements: 10
Enter 10 integers: 4 6 8 12 45 66 23 43 11 2
Sorted list (ascending order): 2 4 6 8 11 12 23 43 45 66
```

The C standard library

▶ An *application programming interface* (API) is a protocol that constitutes the interface of software components.

▶ In the C language this is a collection of functions grouped together according to their domain.

▶ We can access this API (called the *C standard library*) by adding the #include directive at the top of our program file.

▶ Perhaps the most common is the group of functions that support input-output and are accessed by <stdio.h>.

The C standard library: Common header files

- math.h: Defines common mathematical functions.
- stdio.h: Defines core input and output functions.
- stdlib.h: Defines numeric conversion functions, pseudo-random number generation functions, memory allocation, process control functions.
- string.h: Defines string manipulation functions.

Data types

- A program is composed by *constructs*, such as functions and variables.
- A *data type* (or simply a *type*) is a description of the possible values that a construct can store or compute to, together with a collection of operations that manipulate that type, e.g. the set of integers together with operations such as addition, subtraction, and multiplication.
- Common types among most programming languages include booleans, numerals, characters and strings.

Classes of data types

- The *Boolean type* contains the values `true` and `false`.
- The *numeral type* includes integers that represent whole numbers and floating points that represent real numbers.
- The *character type* is a member of a given set (ASCII) and, finally, strings are sequences of alphanumeric characters.
- We can distinguish between *simple types* and *composite* (or *aggregate*) types based on whether or not the values of a type can contain subparts.
- As an example, we can say that integer is a simple type, whereas record is composite. A *data item* is an instance (also: a *member*) of a type.

Primitive data types

▶ With respect to a given programming language, a *primitive* type is one that is built in (provided by the language).

▶ The C language supports two different classes of data types, namely numerals and characters, which are divided into four type identifiers (int, float, double, char), together with four optional specifiers (signed, unsigned, short, long):

IDENTIFIER	TYPE	RANGE
int	integer	$-32,768$ to $32,767$
float	real	1.2×10^{-38} to 3.4×10^{38}
double	real	2.2×10^{-308} to 1.8×10^{308}
char	character	ASCII

Optional specifiers: Short, long, signed and unsigned

- The four specifiers define the amount of storage allocated to the variable.
- We distinguish between *short* and *long* numeral data types that differ in their range.
- The amount of storage is not specified, but ANSI places the following rules:

$$short\ int \leq int \leq long\ int$$

and

$$float \leq double \leq long\ double$$

Optional specifiers: Short, long, signed and unsigned /cont.

- We can also distinguish between *signed* and *unsigned* numeral data types.
- Signed variables can be either positive or negative. On the other hand unsigned variables can only be positive, thus covering a larger range.

OPTIONAL SPECIFIER	RANGE
short int	$-32,768$ to $32,767$
unsigned short int	0 to $65,535$
unsigned int	0 to $4,294,967,295$
long int	$-2,147,483,648$ to $2,147,483,647$

Type conversion

- Type conversion is the transformation of one type into another.
- In C, implicit type conversion (or *coercion*) is the automatic type conversion done by the compiler.
- In the following example, the value of a float variable is assigned to an integer variable which in turn is assigned to another float variable.

```c
#include <stdio.h>
int main() {
 int intvar;
 float floatvar = 3.14;
 float floatvar2;
 intvar = floatvar;
 floatvar2 = intvar;
 printf("%d : %.2f : %1.3f\n", intvar, floatvar, floatvar2);
 return 0; }
```

- The output of the program is 3 :   3.14 :   3.000

Defining constants

- A *constant* defines a data type whose value cannot be modified.
- We can define a constant either with the const keyword as in
  float const pi = 3.14
- or with #define as in
  #define TRUE 1
  #define FALSE 0

Composite data types

- A *composite type* is one that is composed by primitive types or other composite types.
- Normally a composite type is called a *data structure*: a way to organize and store data so that it can be accessed and manipulated efficiently.
- Common composite types include arrays and records.

Arrays

- An *array* is a collection of values (called its elements), all of the same type, held in a specific order (and in fact stored contiguously in memory).
- Arrays can be either static (i.e,. fixed-length) or dynamic (i.e. expandable). An array of size n is indexed by integers from 0 up to and including $n - 1$.
- The composition of primitive (and possibly other composite) types into a composite type results in a new type.
- For example, integers can be composed to construct an array of integers.

Arrays: Example

▶ In the following program, we declare and initialize an array, `numbers`, and then pass it as an argument, together with its size, to function `getAverage()` that will compute and return the average of the elements of the array.

```c
#include<stdio.h>
float getAverage(float[], int);
int main() {
  float numbers[5] = {1, 2.5, 9, 11.5, 23.5};
  printf("Array average: %.1f.\n", getAverage(numbers, 5));
  return 0; }
float getAverage(float list[], int size) {
  int i;
  float sum = 0.0;
  float average = 0.0;
  for (i=0; i<size; i++)
    sum = sum + list[i];
  average = (sum/size);
  return average; }
```

▶ The output is: `Array average:   9.5.`

Pointers

- A *pointer* is a type that references (*"points to"*) another value by storing that other value's address.

- A *pointer variable* (also called an *address variable*) is declared by putting an asterisk * in front of its name, as in the following statement that declares a pointer to an integer.

  `int *ptr;`

- There are two operators that are used with pointers:
    * The "dereference" operator: Given a pointer, obtain the value of the object referenced (pointed at).
    & The "address of" operator: Given an object, use & to point to it. The & operator returns the address of the object pointed to.

Pointers: Example 1

▶ The following code segment declares an integer variable a which is assigned to 42 (line 3), and a pointer p that points to an integer object (line 4).

▶ In line 5, the pointer p is assigned the address of variable a, and in line 6 we display the contents of the object pointed to by p.

▶ Accessing the object being pointed at is called *dereferencing*.

```
1  #include <stdio.h>
2  int main() {
3    int a = 42;
4    int *p;
5    p = &a;
6    printf("p: %d\n", *p);
7    return 0;
8  }
```

▶ The output of the program is p: 42.

Pointers: Example 1 /cont.

```
int a = 42;
```
a is an integer variable, assigned the value 42.

Address: 0x1000
Type: int
Name: a
Value: 42

```
int *p;
```
p is is an integer pointer.

Address: 0x1000
Type: int
Name: a
Value: 42

Address: 0x1004
Type: int
Name: p
Value:

```
p = &a;
```
p is assigned the address of a.

Address: 0x1000
Type: int
Name: a
Value: 42

Address: 0x1004
Type: int
Name: p
Value: 0x1000

```
printf("p: %d\n", *p);
```
Displays the contents of the object pointed to by p: 42.

Pointers: Example 2

- In this example an integer pointer `ptr` points to an integer variable `my_var`.
- We then proceed to modify the contents of `my_var` through `ptr` and finally we verify that the value of `my_var` has indeed been modified.

```c
#include<stdio.h>
int main() {
  int my_var = 13;
  int *ptr = &my_var;
  *ptr = 17;
  printf("my_var: %d\n", my_var);
}
```
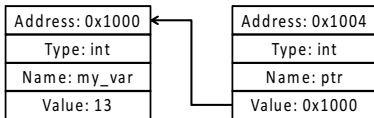
Pointers: Example 2 /cont.: Illustration

`int my_var = 13;`
my_var is an integer variable,
assigned the value 13.

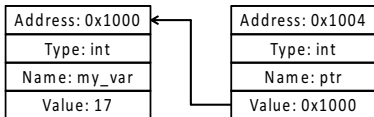| Address: 0x1000 |
| --- |
| Type: int |
| Name: my_var |
| Value: 13 |

`int *ptr = &my_var;`
ptr is is an integer pointer,
assigned the address of my_var.

Address: 0x1000
Type: int
Name: my_var
Value: 13

Address: 0x1004
Type: int
Name: ptr
Value: 0x1000

`*ptr = 17;`
The object pointed to by ptr is
assigned the value 17.

Address: 0x1000
Type: int
Name: my_var
Value: 17

Address: 0x1004
Type: int
Name: ptr
Value: 0x1000

`printf("my_var: %d\n", my_var);`
Displays the contents of my_var: 17.

Pointers: Example 2 /cont.

- ▶ Note that a statement such as `*my_var` would have been illegal as it asks C to obtain the object pointed to by `my_var`. However, `my_var` is not a pointer.
- ▶ Similarly, a statement such as `&ptr` though legal is rather strange as it asks C to obtain the address of `ptr`.
- ▶ The result is a pointer to a pointer (i.e. the address of an object that contains the address of another object).

Aliasing

- Aliasing is a situation where a single memory location can be accessed through different variables.
- Modifying the data through one name implicitly modifies the values associated to all aliased names.
- Consider the program below:

```c
#include<stdio.h>
int main() {
  int a = 7;
  int *ptr;
  ptr = &a;
  printf("a: %d\n", a);
  printf("a: %d\n", *ptr);
  a = 9;
  printf("a: %d\n", a);
  printf("a: %d\n", *ptr);
  *ptr = 11;
  printf("a: %d\n", a);
  printf("a: %d\n", *ptr);
  return 0; }
```

Aliasing /cont.

- In this example, we create an integer variable a and an integer pointer ptr that points to a.
- We then verify that the two variables contain the same value:

```
int a = 7;
int *ptr;
ptr = &a;
printf("a: %d\n", a);
printf("a: %d\n", *ptr);
```

- This will display

```
a: 7
a: 7
```

Aliasing /cont.

- ▶ We then proceed to modify the value of a, first directly
  ```
  a = 9;
  printf("a: %d\n", a);
  printf("a: %d\n", *ptr);
  ```
- ▶ and then through the pointer
  ```
  *ptr = 11;
  printf("a: %d\n", a);
  printf("a: %d\n", *ptr);
  ```
- ▶ This will display
  ```
  a: 9
  a: 9
  a: 11
  a: 11
  ```

Pointers and arrays

- The elements of an array are assigned consecutive addresses. We can use a pointer to an array in order to iterate through the array's elements. Suppose we have the following:

  ```
  int arr[5];
  int *ptr;
  ```

- In the following statement, we assign the first element of the array as the value of the pointer:

  ```
  ptr = &arr[0];
  ```

- Pointer arithmetic makes *(ptr + 1) the same as arr[1].

Pointers and arrays: Example

- In this example we explore pointer arithmetic to assign an array to a pointer, and then use the pointer to display the values of the first three elements of the array.

- We say that we are displaying the contents of the array by *dereferencing the pointer*.

```
#include <stdio.h>
int main() {
  int arr[5] = {1, 3, 5, 7, 11};
  int *ptr;
  ptr = &arr[0];
  printf("arr[0]: %d, arr[1]: %d, arr[2]: %d\n",
         *ptr, *(ptr + 1), *(ptr + 2));
  return 0;
}
```

- The output is:

```
arr[0]: 1, arr[1]: 3, arr[2]: 5
```

Pointers and arrays: Example /cont.

▶ Note that `*(ptr + 1)` is not the same as `*(ptr) + 1`.

▶ In the latter expression the addition of 1 occurs after the dereference, and it would be the same as `arr[0] + 1`.

▶ The statement

  `printf("arr[1]: %d\n", *(ptr) + 1);`

  will display

  `arr[1]: 2`

Pointers as function parameters

▶ In the program below, we deploy function swap that defines two integer
formal parameters, and with the help of a temporary variable it swaps
their values:

```
#include <stdio.h>
void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
int main() {
  int first, second;
  printf("Enter two integers: ");
  scanf("%d%d", &first, &second);
  printf("First: %d, Second: %d.\n", first, second);
  printf("Swap in progress...\n");
  swap(first, second);
  printf("First: %d, Second: %d.\n", first, second);
  return 0;
}
```

Pointers as function parameters /cont.

- ▶ Let us execute the program:
  ```
  Enter two integers: 5 7
  First: 5, Second: 7.
  Swap in progress...
  First: 5, Second: 7.
  ```
- ▶ What is wrong with the program? In C, arguments are passed *by value*, i.e. a copy of the value of each argument is passed to the function.
- ▶ As a result, a function cannot modify the actual argument(s) that it receives.
- ▶ To make the function swap the actual arguments we must pass the arguments *by reference*, i.e. pass the addresses of the actual arguments.

Pointers as function parameters /cont.

- The correct program is shown below:

```c
#include <stdio.h>
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
int main() {
  int first, second;
  printf("Enter two integers: ");
  scanf("%d%d", &first, &second);
  printf("First: %d, Second: %d.\n", first, second);
  printf("Swap in progress...\n");
  swap(&first, &second);
  printf("First: %d, Second: %d.\n", first, second);
  return 0;
}
```

Pointers as function parameters /cont.

- We can execute the program as follows:

  ```
  Enter two integers: 5 7
  First: 5, Second: 7.
  Swap in progress...
  First: 7, Second: 5.
  ```

Function pointers

- Pointers to variables and arrays are examples where a pointer refers to data values.
- A pointer can also refer to a function, since functions have addresses.
- We refer to these as *function pointers*. Consider the following (rather cryptic) declaration:

  long (*ptr)(int);

- This declares a function pointer; It points to a function that takes an integer argument and returning a long integer.

Function pointers /cont.

- We could now initialize the pointer by making it point to an actual function as follows:

  ptr = &factorial;

- This makes ptr point to function factorial(..).

- The function can be invoked by dereferencing the pointer while passing arguments as any regular function call, only in this case we refer to this as an *indirect call*.

Function pointers /cont.: Putting everything together

```c
#include <stdio.h>
long factorial(int);
int main() {
  int n;
  long f;
  long (*ptr)(int);
  ptr = &factorial;
  printf("Enter a non-negative integer: ");
  scanf("%d", &n);
  if (n < 0)
    printf("Negative integers are not allowed.\n");
  else
    f = ptr(n);
  printf("%d! = %ld\n", n, f);
  return 0; }
long factorial(int n) {
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1)); }
```

Function pointers /cont.

- We can execute the program as follows:

  ```
  Enter a non-negative integer: 5
  5! = 120
  ```

Records

- A *record*, or *structure*, is a collection of elements, *fields* (or *members*), which can possibly of different types.
- The syntax of declaring a structure in C is

```
struct <name> {
  field declarations
};
```

Record initialization and assignment: Example

- To create a structure to represent a coordinate on the Cartesian plane, we can say:

```
struct coordinate {
  float x;
  float y;
};
```

- To create a coordinate variable we can now say

```
struct coordinate p;
```

Record initialization and assignment: Example /cont.

- We can eliminate the word struct every time we declare a coordinate variable by making coordinate into a new type with typedef.

  ```
  typedef struct {
    float x;
    float y;
  } coordinate;
  ```

- We can now create a coordinate variable with

  ```
  coordinate p;
  ```

Record initialization and assignment: Example /cont.

- In the following program we define a new type `coordinate` to be a record (struct) with two float members.
- We declare and initialize four variables of type `coordinate`.
- Members of the coordinate type can be initialized during declaration either inline as in

  `coordinate p1 = {0, 0};`

  or by designated initializers as in

  `coordinate p2 = {.x = 1, .y = 3};`

Record initialization and assignment: Example /cont.

- ▶ Subsequently members of a record can be assigned values as in

  ```
  p3.x = 2;
  p3.y = 7;
  ```

  or by assigning the value of one record to another, as in

  ```
  p4 = p3;
  ```

  that copies the member values from p3 into p4.

Record initialization and assignment: Example /cont.

```c
#include<stdio.h>
typedef struct {
  float x;
  float y;
} coordinate;
int main() {
  coordinate p1 = {0, 0};
  coordinate p2 = {.x = 1, .y = 3};
  coordinate p3;
  coordinate p4;
  p3.x = 2;
  p3.y = 7;
  p4 = p3;
  printf("p1 = (%.0f, %.0f)\n", p1.x, p1.y);
  printf("p2 = (%.0f, %.0f)\n", p2.x, p2.y);
  printf("p3 = (%.0f, %.0f)\n", p3.x, p3.y);
  printf("p4 = (%.0f, %.0f)\n", p4.x, p4.y);
  return 0;
}
```

Record initialization and assignment: Example /cont.

▶ The output of the program is

```
p1 = (0, 0)
p2 = (1, 3)
p3 = (2, 7)
p4 = (2, 7)
```

Records and pointers

- A pointer can be deployed to point to a record as in

  ```
  coordinate p = {0, 0};
  coordinate *ptr = &p;
  ```

- The pointer can subsequently be dereferenced using the * operator as in

  ```
  (*ptr).x = 3;
  ```

- An alternative binary operator exists (->): The left operand dereferences the pointer, where the right operand accesses the value of a member of the record:

  ```
  ptr->y = 3;
  ```

Records and pointers /cont.: Putting everything together

```c
#include<stdio.h>
typedef struct {
  float x;
  float y;
} coordinate;
int main() {
  coordinate p = {0, 0};
  printf("p = (%.0f, %.0f)\n", p.x, p.y);
  coordinate *ptr = &p;
  (*ptr).x = 3;
  ptr->y = 3;
  printf("p = (%.0f, %.0f)\n", p.x, p.y);
  return 0;
}
```

Records and pointers: Putting everything together

- The output of the program is
  ```
  p = (0, 0)
  p = (3, 3)
  ```

Records and pointers: Putting everything together /cont.

```c
typedef struct {
  float x;
  float y;
} coordinate;

coordinate p = {0, 0};
```

Type: coordinate
Name: p
x = 0;
y = 0;

```c
coordinate *ptr = &p;
```

ptr →

Type: coordinate
Name: p
x = 0;
y = 0;

```c
(*ptr).x = 3;
ptr->y = 3;
```

ptr →

Type: coordinate
Name: p
x = 3;
y = 3;

Records and arrays

▶ `line[]` is an array of type `coordinate`, itself defined as a record.
▶ The elements of the array are initialized at the time of declaration.
▶ We use the dot (.) operator to access fields of individual records:
  `line[0].x` accesses the x field of the first element (record) of `line`.

```
#include<stdio.h>
typedef struct {
  float x;
  float y;
} coordinate;
```

Records and arrays /cont.

```
int main() {
  coordinate line[2] = {
    {0, 0},
    {11, 19}
  };
  printf("Line points: (%.0f, %.0f), and (%.0f, %.0f).\n",
         line[0].x, line[0].y, line[1].x, line[1].y );
}
```

Records and arrays /cont.

- The output of the program is
  ```
  Line points: (0, 0), and (11, 19).
  ```

Memory management: An example

- We have already seen that when declaring an array, we have to specify not just the type of its elements but also the size of the array.
- This allows the system to allocate the appropriate amount of memory.
- Once specified, we cannot change the size of the array dynamically, i.e. during the execution of the program.
- Through one of its standard libraries, the C language offers a number of functions that allow us to circumvent this problem and manage memory dynamically.

Memory management: An example /cont.

- Consider the program below:

```c
#include<stdio.h>
#include <stdlib.h>
int main() {
  int *array = malloc(3 * sizeof(int));
  if (array == NULL) {
    printf("ERROR: Out of memory.\n");
    return 1;
  }
  *array = 1;
  *(array + 1) = 3;
  *(array + 2) = 5;
  printf("%d\n", *array);
  printf("%d\n", *(array + 1));
  printf("%d\n", *(array + 2));
  free(array);
  return 0;
}
```

Memory management: An example /cont.

- In the following statement

  `int *array = malloc(3 * sizeof(int));`

  we request the allocation of enough memory for an array of three elements of type `int`.

- We stress the fact that this is merely a request and the allocation of memory is not guaranteed to succeed.

- If successfull, function `malloc` returns a pointer to a block of memory.

- If not successfull, `malloc` will return the special value `NULL` to indicate that for some reason the memory has not been allocated.

Memory management: An example /cont.

- As a result, to indicate success we now have to verify that our array
  pointer is not NULL.

  ```
  if (array == NULL) {...}
  ```

- We then proceed to assign values to the elements of the array and
  subsequently display them.

- Once we no longer need the array, we have to release the allocated
  memory back to the system.

- We do this with function free:

  ```
  free(array);
  ```

Memory management: An example /cont.

- ▶ Memory that is no more needed but it is not deallocated cannot be reused by the system.
- ▶ This waste of resources can accumulate and can lead to allocation failures when resources are needed but have been exhausted.
- ▶ Even though memory not released with `free` is automatically released once the program terminates, it is a good practice to ensure that we explicitly release memory once it is not needed.
- ▶ The output of the program is

  1
  3
  5

Memory management functions

- malloc: Allocates the specified number of bytes.
- realloc: Increases or decreases the size of the specified block of memory.
- calloc: Allocates the specified number of bytes and initializes them to zero.
- free: Releases the specified block of memory back to the system.

The linked list data structure

- ▶ The linked list is among the most common data structures.
- ▶ It can be used to implement several common abstract data types, including stacks, and queues.
- ▶ Among the different variants, the singly linked list is the simplest: It represents a chain of elements, called *nodes*, where each node contains a minimum of two *fields*: the *data field* (or *value field*) and the *next link* (or *next pointer*) that points to the next node in the chain.
- ▶ Additionally, the *head* of a list is the list's first node and the *tail* either points to the rest of the list (thus following the corresponding mathematical structure), or it can sometimes point to the last node in the list.

The linked list data structure: An example

- In this example, we will construct a linked list with two nodes. A node is represented as a record:
```
struct node {
  int data;
  struct node *next;
};
```
- Initially the list is empty, thus the head of the list points to NULL:
```
struct node *head = NULL;
```

The linked list data structure: An example /cont.

- We are now ready to request memory for the head of the list:
  ```
  head = malloc(sizeof(struct node));
  if (head == NULL) {
    printf("ERROR: Out of memory.\n");
    return 1;
  }
  ```
- Once memory has been allocated we need to a) have the head's next field point to null and b) assign some value to the data field:
  ```
  head->data = 5;
  head->next = NULL;
  ```

The linked list data structure: An example /cont.

▶ We follow the same procedure with the second node of the list, but at the end we need to make sure that a) the next field of the new item points to the node currently pointed to by head and b) the new item becomes the new head, i.e. the head pointer is updated to point to the new node:

```
new->next = head;
head = new;
```

```
#include<stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
```

The linked list data structure: An example /cont.
Putting everything together

```c
int main() {
  struct node *head = NULL;
  struct node *new;
  head = malloc(sizeof(struct node));
  if (head == NULL) {...}
  head->data = 5;
  head->next = NULL;
  new = malloc(sizeof(struct node));
  if (new == NULL) {...}
  new->data = 11;
  new->next = head;
  head = new;
  printf("%d ", head->data);
  printf("%d ", (head->next)->data);
  return 0;
}
```

The linked list data structure: An example /cont.
Putting everything together /cont.

```
node *head = NULL;
```
head ● →NULL

```
head = malloc(sizeof(struct node));
```
head ● → | data | next |

```
head->next = NULL;
```
head ● → | data | next | → NULL

```
head->data = 5;
```
head ● → | data: 5 | next | → NULL

```
new = malloc(sizeof(struct node));
new->data = 11;
```
new ● → | data: 11 | next |

head ● → | data: 5 | next | → NULL

```
new->next = head;
head = new;
```
head ● → | data: 11 | next | → | data: 5 | next | → NULL