Chapter 5: C Functions

- Subprograms and modularisation — *divide et impera*
- Types and Prototypes
- Side-effects
- Scope, local variables, memory aspects
- `static` variables, storage classes
- Recursion

Subprograms

- A **subprogram** is a (parameterised) fragment of a program.
- A **subprogram call** is an instantiation of a subprogram with *actual parameters.*
  - **Function** calls are expressions
  - **Procedure** calls are statements
- The purpose of introducing subprograms is **modularisation**.
- Modular components are accessed via **interfaces** — ***the interface of a subprogram consists of:***
  - **type:** argument types, result type
  - **specification:** properties, description of effects
- (In programming, the word **module** is usually reserved for components consisting of collections of subprograms and/or data type definitions.)

Subprograms in C

- Every expression can be used as a statement:
  - No procedures necessary — only **functions**
  - Functions with return type `void` are "intended as procedures"
  - Many functions that are often used as procedures have non-`void` return types

    — know and check!
- Types of functions are formally captured in "**prototypes**"
- No further part of function specifications is formally supported by C

Function Types and Prototypes

Mathematics	**C**
$\sin : \mathbb{R} \to \mathbb{R}$	double *sin*(double);
$\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	int *gcd*(int, int);
$pow : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$	double *pow*(double, double);

Prototypes are function **declarations.**

- Prototypes are *implied* by (ANSI-style) function **definitions.**
- The common part is also called **function header**.
- After the prototype has been seen by the compiler, the function name and its type are known.
- Prototypes can be used as "forward-declarations".
- `*.h` files frequently contain `extern` prototypes.

Local Variable — Global Variables

```
int k;

int f(double h)
{
  int n;
  ...
}
```

- *k* is a **global variable**
- *n* is a **local variable**
- *h* is a **formal parameter** — inside the body this is equivalent to a **local variable**

Scope and Instances of Variables in C

- **All variables** are *visible* only **after declaration**
- **Global variables** are *visible* **in the file of their declaration**
- **Local variables** are *visible* **in the block of their declaration**

- For **all variables**, an instance is created when control flow passes their **definition**.
- **Global variables** have only **one** instance
- `static` **(local) variables** have only **one** instance
- **Local variables** have **one instance for each call of the function/block**

Scope and Side-Effects

```
#include <stdio.h>
int x = 0;

int incrX( ) { x++; return x; }
```

What is the type of `incrX`?

- **Prototype:**

  `int incrX( void );`

- **Mathematical:**

  $\mathtt{incrX} : \mathbb{1} \rightarrow \mathtt{int}$

This is not the whole interface to `incrX`!

Scope and Side-Effects — Simulation

```
#include <stdio.h>
int x = 0;

int incrX( ) { x++; return x; }

int main() {
  int x = 10, y;
  y = incrX();
  printf("%d  %d  %d\n", x, y, incrX());
  return 0;
}
```

---

global	main()			Output
x	x	y		
0			init.	
	10		init.	
1			x++;	
		1	y = incrX();	
2			x++;	
				10 1 2

Scope and Side-Effects

```
#include <stdio.h>
int x = 0;

int incrX( ) { x++; return x; }

int main() {
  int x = 10, y;
  y = incrX();
  printf("%d  %d  %d\n", x, y, incrX());
  return 0;
}
```

Locally defined variables **shadow** variables defined in an outer scope.

Side-effects:

– `incrX` changes the value of a variable not mentioned in its formal interface.

– The return value of `incrX` depends on a variable not mentioned in its formal interface.

Pure Functions

Pure functions have no side-effects:

– Return values depend only on the actual parameters

– No global variables are updated

– No I/O is performed

Math library functions are "almost pure":

- In case of error, the global variable `errno` is set.
- Floating-point precision may depend on, e.g., compiler switches.

With **pure functions**, it is **easy** to apply **mathematical reasoning!**

The Abstract Datatype of Stacks

- A **stack** is a very simple and very useful ***abstract datastructure***.
- An **abstract datatype** is described only by its **interface**:
  - ***Signature:*** type names (*sorts*), and function names (*function sysmbols*) with their types
  - ***Specification*** (*laws*): properties that relate the different functions
- **Stack signature**: sorts: Stack and Elem; function symbols:

$$\begin{array}{lcl} \mathsf{emptyStack} & : & \mathsf{Stack} \\ \mathsf{push} & : & \mathsf{Elem} \times \mathsf{Stack} \rightarrow \mathsf{Stack} \\ \mathsf{pop} & : & \mathsf{Stack} \rightarrow \mathsf{Stack} \\ \mathsf{top} & : & \mathsf{Stack} \rightarrow \mathsf{Elem} \end{array}$$

Stack laws: $\mathsf{pop}(\mathsf{push}(x, s)) = s$
$\mathsf{top}(\mathsf{push}(x, s)) = x$

Stack is a ***free*** datatype: no other equations hold.

Function Calls and The Stack

- The run-time environment of C program execution maintains a **stack**
- This stack contains **activation records** for active function calls (also called **stack frame**)
- Each activation record contains all **local variables** for one function call
- Operationally:
  - At program start, there is only one stack frame; it contains all global variables
  - When function f is called, a new activation record is pushed on the top of the stack.

    This activation record contains all local variables of f, including the formal parameters, which are initialised to the values of the actual parameters.
  - When the call to function f returns, the activation record for that call is popped from the stack.

Repeated Function Calls

```
#include <stdio.h>          /* squares.c */

int f(int k) {
  return 2 * k + 1;
}

int main() {
  int s = 0, i;
  for(i = 0; i < 4; i++)
   { s += f(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```


Repeated Function Calls 2

```
#include <stdio.h>          /* squares2.c */

int f(int k) {
  k *= 2;
  return ++k;
}

int main() {
  int s = 0, i;
  for(i = 0; i < 4; i++)
   { s += f(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```


Repeated Function Calls 3

```
#include <stdio.h>          /* squares3.c */

int count=0;

int f(int k) {
  count++;    /* count calls to this function */
  k *= 2;
  return ++k;
}

int main() {
  int s = 0, i;
  for(i = 0; i < 4; i++)
   { s += f(i);
     printf("%d  %d\n", i, s);
   }
  printf("%d  %d  %d\n", i, s, count);
  return 0;
}
```


Alternating Function Calls

```
#include <stdio.h>          /* series1.c */

int f(int k) {
  k += 2;
 return k + 1;
}

int g(int m) { return 2 * m * m - 1; }

int main() {
  int s = 0, i;
  for(i = 0; i < 3; i++)
   { s += f(i);
     s += g(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```

Nested Function Calls 1

```
#include <stdio.h>            /* series2.c */

int f(int k) {
  k += 2;
 return k + 1;
}

int g(int m) { return (m + 1) * f(m); }

int main() {
  int s = 0, i;
  for(i = 0; i < 3; i++)
   { s += g(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```


Nested Function Calls 2

```
#include <stdio.h>            /* series3.c */

int f(int k) {
  k += 2;
 return k + 1;
}

int g(int m) { return (m – 1) * f(m); }

int main() {
  int s = 0, i;
  for(i = 0; i < 3; i++)
   { s += f(i);
     s += g(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```


Recursive Function Calls — Factorial

```
#include <stdio.h>                    /* factorial1.c */

int factorial(int k) {
  if (k < 2)
    return 1;
  else
    return k * factorial(k – 1);
}

int main() {
  printf("%d\n", factorial(5));
  return 0;
}
```

Note:

- **At most one** recursive call per incarnation: **linear recursion**
- Recursive call not in “tail position”: result used for multiplication

Factorial — Tail-Recursive

```
#include <stdio.h>                    /* factorial2.c */

int fact(int n, int k) {
  if (k < 2)
    return n;
  else
    return fact(n * k, k – 1);
}

int main() {
  printf("%d\n", fact(1,5));
  return 0;
}
```

Note:

- All recursive calls are the **last** action before returning: **tail recursion**

Factorial — Tail-Recursion Made More Explicit

```
#include <stdio.h>             /* factorial3.c */

int fact(int n, int k) {
  if (k < 2)
    return n;
  else {
    n *= k;
    k--;
    return fact(n, k);
  }
}

int main() {
  printf("%d\n", fact(1,5)); return 0;
}
```

Note:

- The **tail call** now has the parameter-variables as arguments
- Intermediate step of **mechanical transformation into** while **loop**

Factorial — Tail-Recursion Turned into Repetition

```
#include <stdio.h>             /* factorial4.c */

int fact(int n, int k) {
  while ( ! (k < 2) ) {
    n *= k;
    k--;
  }
  return n;
}

int main() {
  printf("%d\n", fact(1,5));
  return 0;
}
```


`static` Local Variables

```
#include <stdio.h>             /* squares4.c */

int step(int n) {
  static int d = 1;
  static int q = 1;
  int r = n * q;
  d += 2;
  q += d;
  return r;
}

int main() {
  int i;
  for(i = 1; i <4 ; i++)
     printf("%d  %d\n", i, step(i));
  return 0;
}
```

Non-`static` local variables are also called **automatic**.

Recursive Function Call Example

What is the output ot the following C program:

```
#include <stdio.h>             /* myproc.c */

void myprocedure(int n, float s)
{
  static int k=2;
  float r = s / k;
  if (n < 0) return;
  k = k + 1;
  myprocedure(n - 1, (s + r) / 2);
  r = r * k;
  printf("%d %d %.2f %.2f\n",n,k,s,r);
}

int main(void) {
  myprocedure(1, 12.0);   /* myprocedure(3, 144.0) */
  return 0;
}
```

Cascading Recursion — Fibonacci

```
#include <stdio.h>                /* fib1.c */

int fib(int n) {
  if ( n == 0 || n == 1 )
    return n;
  else
    { int f1, f2;
      f1 = fib( n − 1 );
      f2 = fib( n − 2 );
      return f1 + f2;
    }
}

int main() { printf("%d\n", fib(5)); return 0; }
```

Note:

- **More than one** recursive call in some incarnations: **cascading recursion**

Nested Recursion — The Ackermann Function

```
#include <stdio.h>              /* ackermann.c */
#include <stdlib.h>

int ack(int x, int y) {
  if ( x == 0 )
    return y + 1;
  else if ( y == 0 )
    return ack( x − 1, 1 );
  else
    return ack( x − 1, ack( x, y − 1 ));
}

int main(int argc, char * argv[]) { int i = atoi(argv[1]);
  printf("%d\n", ack(i,i)); return 0; }
```

Note:

- A recursive call **as argument of another recursive call**: **nested recursion**
- This function **cannot** be written without recursion or while loops

Fibonacci — Output of Instrumentation

```
fib(5) start
        fib(4) start
                fib(3) start
                        fib(2) start
                                fib(1) start
                                fib(1) = 1
                                fib(0) start
                                fib(0) = 0
                        fib(2) = 1
                        fib(1) start
                        fib(1) = 1
                fib(3) = 2
                fib(2) start
                        fib(1) start
                        fib(1) = 1
                        fib(0) start
                        fib(0) = 0
                fib(2) = 1
        fib(4) = 3
        fib(3) start
                fib(2) start
                        fib(1) start
                        fib(1) = 1
                        fib(0) start
                        fib(0) = 0
                fib(2) = 1
                fib(1) start
                fib(1) = 1
        fib(3) = 2
fib(5) = 5
5  5
```


Different Kinds of Recursion

- **Linear recursion:** in each branch at most one recursive call
  - **Tail recursion (repetitive recursion):**

    The recursive call is the last action in its branch

    Can be mechanically converted into while ***loop!***

- **Non-linear recursion:**
  - **Cascading recursion:**

    several recursive calls "side-by-side" — *fibonacci*

  - **Nested recursion:**

    recursive calls occur as arguments of other recursive calls — *ackermann*