Functions

Have you seen C functions before? **Yes!** We’ve been working with the `main` function from the very beginning, as well as, `printf`, `scanf`, etc., which are also functions.

**Definition**
A function is a relatively independent block of code, within a larger program, that performs a particular task.

Functions allow us to:
- better structure our programs
- easily modify and extend programs
- easily test for and isolate bugs
- minimise code duplication and maximise reuse

Here is a simple program with two functions:

```c
#include <stdio.h>

int max(int n, int m); // function prototype

int main(int argc, char *argv[]) {
    printf("The larger of \%d and \%d is \%d.\n", 3, 4, max(3, 4));
    return 0; // -------- function call
}

int max(int n, int m) { // function definition
    int maximum;
    if (n > m) {
        maximum = n; // this simple function
    } else {
        maximum = m; // returns the value of the largest of its arguments, n and m
    }
    return maximum;
}
```

Design and Abstraction

Functions are powerful structuring and abstracting tool in programming languages.

**What is abstraction?**
The process of creating self-contained units of software that allow the solutions to tasks to be parametrised, and hence made general purpose. [Moffat, 2003]

The idea is that we can break down our program into tasks and then reason about the interaction between the tasks while abstracting/hiding their exact implementation details.

**Top-Down Design**
A design approach that starts with the high-level task and then progressively breaks it down into smaller and smaller sub-tasks until the desired level of detail is achieved.

We often use the top-down approach for problem solving, stopping when the level of detail roughly corresponds to individual functions.

**Remember**
Each function should perform a single logical task!

**Problem**
Let’s revisit a simple problem and apply top-down design and abstraction: Read two integers and print out the smallest.

What are the logical tasks?
- read an integer (×2)
- find the smaller of the two
- print out the result
Design and Abstraction

What is the abstraction? We assume that we have some parts already and abstract over their implementation, in particular:

- read an integer
- find the minimum of two integers

We already abstract over the details of printing by using printf.

So let’s assume we have this function for reading an integer:

```c
int readInt(void);
```

And this function for finding the minimum of two integers:

```c
int minInt(int n1, int n2);
```

So how do we use this?

Andrew Taylor

Well, let’s write our program by using these functions!

```c
#include <stdio.h>

int readInt(void); // function prototypes
int minInt(int n1, int n2);

int main(int argc, char *argv[]) {
    int n, m, min;
    printf("Enter two integers: ");

    // use function to read numbers!
    n = readInt(); // collect return value
    m = readInt(); // in n & m

    // use function to find minimum
    min = minInt(n, m); // pass n & m as args
    printf("The smaller number is %d.\n", min);
    return 0;
}
```

Andrew Taylor

The Type of Functions

Arguments and Return Value

Functions are parametrised via zero or more arguments and they may (optionally) return a single value.

Consider:

```c
int minInt(int n1, int n2);
```

This function has:

- two arguments (n1 and n2), both of type int
- a return value of type int

The Type of Functions

When defining a function, you have control over:

- the number, order and type of arguments; and
- type of the return argument (if any)

Example:

```c
double foo1(int i, char s[], double d);
double foo2(float f, double dd[], int ii[]);
void foo3(char s1[], char s2[][SIZE]);
int foo4(void);
void foo5(void);
```

The void Type

The special type void is used to indicate that a function takes no arguments and/or returns no value.
**Function Prototypes**

**Definition**

A **function prototype** provides the compiler with the **type information** for a particular function.

**Function Prototypes** are needed because the compiler processes the program code **sequentially**. It lets the compiler know that the implementation for these functions can be **found somewhere else**. A **compile error** occurs if a function call is encountered **before** it is defined.

**Function prototypes** provide the compiler with **type information** about our functions you want to use.

**Type Mismatch**

The types of arguments and **return value** must match the types expected by the **actual** function!

**Example**

Here is the function call for **minInt**:

```c
min = minInt(n, m);
```

A function is called by its **name** (e.g., `minInt`) and with the correct number of **parameters** (e.g., `n` and `m`), of appropriate types, to **match its arguments**.

If a function **returns a value**, that value may be collected by assigning it to a variable (e.g., `min`). If the **value is not collected** then it is lost.

**Function Calls**

**Execution Flow**

The code of a function is only executed when requested via a **function call**.

- Current code execution is **halted**;
- Execution of the function body **begins**;
- When the function **completes**, execution **resumes** at a point after the function call.

**Example**

Let’s now write our two functions:

```c
int readInt(void) {  // this must match prototype
    int n;
    scanf("%d", &n);  // <--- could and should do some error checking here
    return n;
}

int minInt(int n1, int n2) {  // n1 & n2 behave like local variables, initialised
    // with values supplied to the function when called
    int min;
    if (n1 < n2) {  // find minimum value
        min = n1;
    } else {
        min = n2;
    }
    return min;  // return minimum value
}
```
Argument Passing: Standard Variables

Most arguments in C are passed by value. Consider:

```c
void foo(void) {
    int n = 5;
    bar(n); // pass n as argument
    printf("%d\n", n); // what is printed?
}
void bar(int n) {
    n = n + 5; // increment argument
}
```

printf() \(\Rightarrow\) 5. Why?

- \(n\) in `foo()` and `bar()` are a different entities—they have different scopes.
- The value of \(n\) is passed by value to the `bar()` and the original \(n\) cannot be altered in `bar()`.

Argument Passing: Array Variables

Arrays are always passed by reference. This means that modifying an array passed as an argument will reflect back in the calling code.

```c
void foo(void) {
    int num[1] = {0};
    bar(num); // pass num as argument
    printf("%d\n", num[0]); // what is printed?
}
void bar(int num[1]) {
    num[0] = 42; // modify argument
}
```

printf() \(\Rightarrow\) 42. Why?

Because a reference to the array `num` is passed to `bar()`, any changes to the referenced `num` in `bar()` is reflected in the original `num`.

The return Statement

The `return` statement, as its name implies, terminates function execution and returns control to the caller.

Functions that specify a return value

These functions are required to supply a value of correct type to the `return` statement.

```c
int readInput(void) {
    int x;
    ...
    return x; // return value matches declared return type
}
```

Functions with a void return type

These functions do not need to return a value and the use of `return` is optional.

```c
void initVar(int x) {
    int counter;
    ...
    counter = x;
}
```

Style Note

You can specify multiple `return` statements within the same function. However, this is generally considered bad style; try to avoid doing so as much as possible.
Return Value

When a function completes, the instance of that function is destroyed including all its local variables.

If the data that the function returns is important, you must capture the information as it completes otherwise it is lost forever.

```c
void foo(void) {
    int n = 5;
    n = bar(n) + 5;
    printf("%d\n", n); // what is printed?
}

int bar(int n) {
    return (n + 5);
}
```

`printf()` ⇒ 15, explain?

`bar(n)` + 5 ↦→ `bar(5)` + 5 ↦→ 10 + 5 ↦→ 15.

Andrew Taylor

COMP1911 Computing IA

Return Value

It might be tempting to try returning an array from a function:

```c
int [] foo(void) {
    int nums[] = {1,2,3};
    return nums;
}
```

This looks good but fails spectacularly!

**Careful**

Arrays are passed by reference, so a pointer to the local array `str` is returned. The array is destroyed immediately after the return statement. Using it in the caller then becomes a run-time error!

It is possible to return dynamically allocated arrays, which you will only learn in COMP1921.

Andrew Taylor

COMP1911 Computing IA

Scope

Scope is an important concept in programming and it deals with the visibility and lifetime of program entities.

Blocks of code in C are delimited by a pair if braces `{}`. The body of a function forms such a block.

Generally the scope of a C entity (usually a variable) is between the point of declaration and the end of the block within which the declaration is found.

**Global Scope**

Entities declared outside a block (e.g., functions) have global scope, they are visible in all blocks.

```c
int n = 5; // global variable, do not use those!
int bar(int n); // prototype for bar

int main(int argc, char *argv[]) {
    int i = j; // illegal, j not in scope
    int j = n; // OK, uses global n
    foo(); // illegal, foo not in scope
    return 0;
}

void foo(void) {
    int n = 10; // OK, but hides global n
    int j = i; // illegal, i not in scope
    bar(n); // OK because of prototype
}

int bar(int n) { // argument n hides global n
    return (n + 5);
}
```

Andrew Taylor

COMP1911 Computing IA
Arrays as Function Arguments

Recall that arrays are passed to functions by reference.

Examples of how they are declared:

```c
void foo1(double ff[]);
void foo2(double ff[][SIZE]);
void foo3(double ff[][SIZE1][SIZE2]);
```

Notice that the size of the first (dominant) dimension may be left unspecified.

**Consequence:** it is up to the programmer to manage the number of elements in its array argument.

Options:

- by using a size constant
- by specifying an additional array size argument (see next slide)

---

Recursion

The concept of recursion is often seen in mathematics:

\[
\text{factorial } n = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times \text{factorial } (n - 1) & \text{if } n > 0
\end{cases}
\]

Note that we have a **base case** \((n = 0)\) and a **recursive case**. The base case is necessary for the recursion to terminate.

When we talk about recursion in computer science we often refer to **recursive functions**.

**Definition**

A recursive function is one which, as part of its definition, makes reference to (i.e., calls) itself.

Recursive functions are often simple and easy to write.

---

Arrays as Function Arguments

Consider the following:

```c
#define SIZE 10
int sum1(int nums[]);
int sum2(int nums[], int size);
int main(void) {
    int nums[10] = {1, 2, 3};
    sum1(nums);
    sum2(nums, 3);
    return 0;
}
```

The function `sum1` uses `SIZE` to iterate through its array argument, while `sum2` uses the supplied `size` argument. **Why is sum2 better?**

Consider the factorial function, iteratively:

```c
int factorial(int n) {
    int i;
    int fact = 1;
    for (i = 1; i <= n; i = i + 1) {
        fact = fact * i;
    }
    return fact;
}
```

And, recursively:

```c
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else { 
        return (n * factorial(n - 1));
    }
}
```
A function may call itself more than once, in fact there are no restrictions. Consider the Fibonacci sequence:

\[
\text{fib } n = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{fib } (n - 1) + \text{fib } (n - 2) & \text{if } n \geq 2
\end{cases}
\]

We can implement it directly from the formula:

```c
// Note: This function is very inefficient
int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return (fib(n - 1) + fib(n - 2));
    }
}
```