Functions in C

The concept of a function is a fundamental idea in mathematics. For example, the mathematical notation

$$f(x) = x^2$$

is used to describe the simple mathematical function of squaring a value. In this example f is the name of the function, and the variable x that appears inside the parentheses is called an **argument** or **parameter** of the function.

Mathematically: *x=2, f(x)=4; x=9, f(x)=81;* and so on.

Functions take in arguments and pass back a **return value** (here: 4, 81).

The return value of a function is determined by its definition and the values of its parameters.

C functions are like mathematical functions in that **they can accept arguments** and **they can return values**.

Functions can be either defined by the programmer or pre-defined as part of a system library. First, we look at examples from the C system libraries. Then: how to define your own functions.

Programming with Libraries

- No-one wants to reinvent the wheel. Examples: compute square root, capitalize a word, fetch a URL...
  - ➢ **How to re-use bits of software** – a major challenge for computer science and software engineering.
- These software "components" are stored in a **software library** from which they can be retrieved and incorporated into new programs as needed.
- A library has two parts:
  1. An **interface**, which specifies which items are stored in the library and how they are used. This is stored in a **header file** with filename extension ".h"

     ```
     #include <math.h>
     ```
  2. An **implementation**, which contains definitions of the items in the library (i.e. the code). The library user doesn't need to see this!
- C provides a large set of libraries for the use of the programmer: **`ctype.h`**, **`string.h`**, **`stdlib.h`**, **`stdio.h`**, **`math.h`**, **`stdarg.h`**, **`setjmp.h`**, **`signal.h`**, **`assert.h`**, **`time.h`**, **`locale.h`**…

Examples of Library Functions

- The **`math.h`** library contains many functions relating to mathematical operations – if you use these, they can save you a lot of work.
- Recall the program to calculate the volume of a sphere. The formula is:

$$v = \frac{4}{3}\pi r^3$$

- The relevant line of code to carry out this calculation looked like

```
float volume = 4.0/3*PI*radius*radius*radius;
```

- Another example could be

```
float energy = mass*SPEEDOFLIGHT*SPEEDOFLIGHT;
```

- Need a general-purpose "raise to a power" function…

Examples of Library Functions (contd.)

- Use the **`pow`** function from the **`math.h`** library:

$$pow(x, y) = x^y$$

- ***x*** and ***y*** are its arguments and it returns ***x*** raised to the power of ***y***.
- So we can say:

```
float volume = 4.0/3*PI*pow(radius,3);
float energy = mass*pow(SPEEDOFLIGHT,2);
```

- In order to use the **`pow`** function we need (at the start of the program)

```
#include “math.h”
```

- Important: you don't need to know/care how **`pow`** actually does its job -- you can just use it. (Of course, it's just a piece of C code…)

Examples of Library Functions (contd.)

- There are many other functions in **`math.h`**, e.g.

```
#include “math.h”
   . . .
int num = 100;
float result = sqrt(num);
```

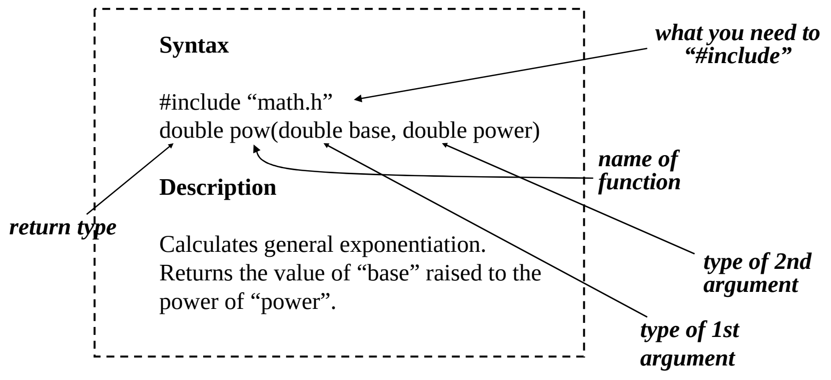
$$f(x) = \sqrt{x}$$

- Other examples:

`ceil(x)`	smallest integer not less than **`x`**
`floor(x)`	largest integer not greater than **`x`**
`fabs(x)`	absolute value of **`x`**
`sin(x)`	trig. sine of **`x`**, where **`x`** is in radians
`acos(x)`	trig. arc cosine of **`x`**, expressed in radians in (0,π)
`exp(x)`	exponential of **`x`** (i.e. **`pow(e,x)`**, e=2.71828…)
`log(x)`	logarithm of **`x`** (base e)
`log10(x)`	logarithm of **`x`** (base 10)

How to use library functions

- When using functions it is important to know what parameters they expect (if any) and what value they return (if any).
- Specifically, you must know the **number, order,** and **type of the parameters**, and the **type of the return value**.
- To find this out, use your programming environment’s “help” facility:



Syntax

#include “math.h”
double pow(double base, double power)

Description

Calculates general exponentiation.
Returns the value of “base” raised to the power of “power”.

Actual and Formal Parameters

- The variables/values used in the function call are called **actual** parameters
- The variables/values used in the function itself are called **formal** parameters
- Actual variable names used are irrelevant – their role is determined solely by their **position in the list**. For example:

```
double power = 3;
double base = 10;
double res = pow(power,base); /* calculates 3^{10}, not 10^{3} */
```

- If formal and actual parameters have different types, the values are converted:

```
float base = 3;
float power = 10;
float res = pow(base,power);
```

*when executed, computer automatically assigns to formal parameters the **`doubles`** to which **`base`** and **`power`** correspond*

*then, the return value **`double`** is automatically converted back to a **`float`***

Programming with Libraries (contd.)

- not just for mathematical functions! For example:
  - manipulating date and time values
  - text string processing
  - accessing files and low-level system resources

  etc.

- But: sometimes you have to write your own functions, because (e.g.)
  - precise functionality you need isn’t in standard library
  - your functions should reflect your algorithm’s refinement structure

- key ideas: a function you write has to be
  - **defined** (specifies what exactly the function does)
  - **declared** (so that it is accessible to other functions that use it)
  - **called** (when another function needs the processing provided by your function)

A simple example: write a function to compute integer powers: result = base^{exponent}

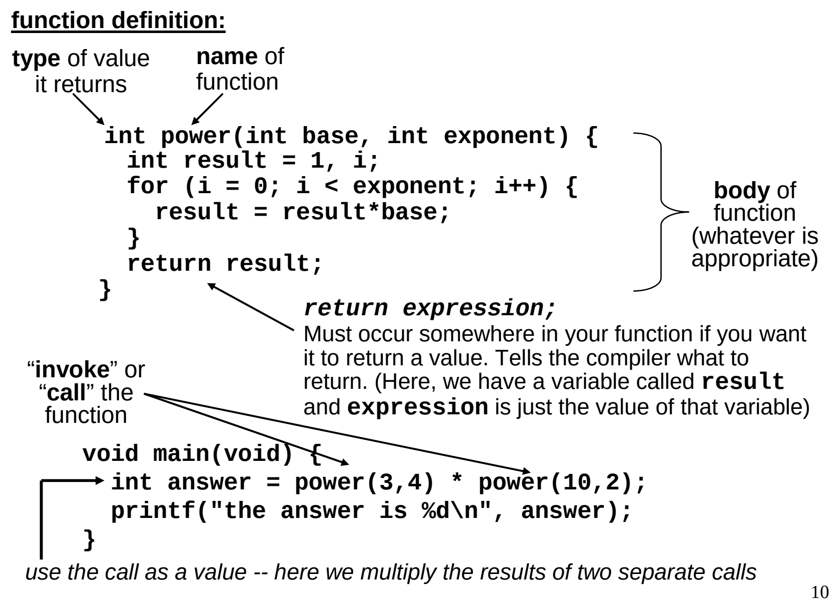
```
#include <stdio.h>

int power(int base, int exponent) {
  int result = 1, i;
  for (i = 0; i < exponent; i++) {
    result = result*base;
  }
  return result;
}

void main(void) {
  int answer = power(3,4) * power(10,2);
  printf("the answer is %d\n", answer);
}
```

produces the screen output: `the answer is 8100`

A simple example: write a function to compute integer powers: result = base^{exponent}



function definition:

type of value it returns

name of function

```
int power(int base, int exponent) {
  int result = 1, i;
  for (i = 0; i < exponent; i++) {
    result = result*base;
  }
  return result;
}
```

body of function (whatever is appropriate)

`return expression;`
Must occur somewhere in your function if you want it to return a value. Tells the compiler what to return. (Here, we have a variable called **`result`** and **`expression`** is just the value of that variable)

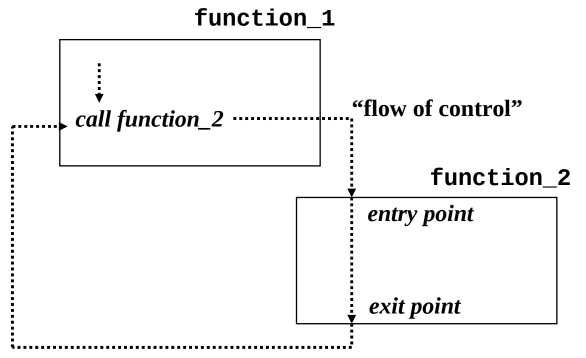
“**invoke**” or “**call**” the function

```
void main(void) {
  int answer = power(3,4) * power(10,2);
  printf("the answer is %d\n", answer);
}
```

use the call as a value -- here we multiply the results of two separate calls

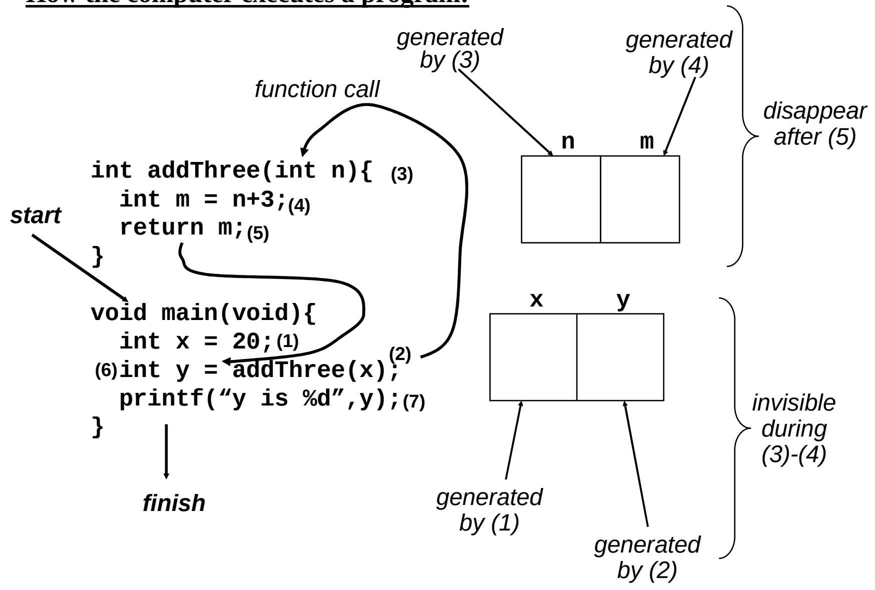
Function call mechanism

When one function calls another function in C, the calling function is “suspended” until the called function “returns control” to it:



Note: calling function **may or may not** send data to called function;
called function **may or may not** return data to calling function when it returns.

How the computer executes a program:



```
int addThree(int n){ (3)
  int m = n+3;(4)
  return m;(5)
}

void main(void){
  int x = 20;(1)
(6)int y = addThree(x);(2)
  printf(“y is %d”,y);(7)
}
```

General “template” for function definition

```
return-type function-name (parameter-list) {
   declarations & statements (as usual),
   optionally with return somewhere in here
}
```

If the function returns an integer, use return-type **`int`**
If the function returns a double (e.g. **`pow`**), use return-type **`double`**
etc.
Not all functions return a value (e.g. **`main`**) -- use special type **`void`**
If no return-type specified: default is **`int`** (not **`void`**).

Like variable names, the **`function-name`** assigns a unique name for the function. You should name a function according to its purpose (i.e. meaningfully).

The **`parameter-list`** is used to specify what arguments the function expects. The general format is:

```
type variable, type variable, type variable, …
```

If function doesn’t take any arguments: use **`(void)`** or **`()`**

General “template” for function definition (contd.)

In C, function definitions CANNOT be “nested” – each function must be defined outside any other function definition.

If the calling function wants to “capture” a returned value from a called function, possible ways include **assignment** and **using it in an expression**:

```
int y = addThree(x);
int answer = power(3,4) * power(10,2);
```

A returned value can also be **ignored** by the calling function. For example, so far we haven’t been concerned with the return value from a call to **`printf()`**, although **`printf()`** does provide one – we’ve just discarded it every time.

In a function definition, when the closing **`}`** is reached, control is returned to the point in the calling function where this function was called. This is referred to as an **implicit return**. The **`return`** statement, on the other hand, is an **explicit return** since its effect is to ***immediately*** return control to the calling function. If the **`return`** statement has an associated **`expression`**, the value of that **`expression`** is also returned to the calling function as the “returned value”.

Note: **`return expression;`**
is equivalent to **`return (expression);`**

Multiple `return` statements

Functions need not have just one `return` statement.
If function returns something other than **`void`**, then every possible “execution path” through the function body should end up at a **`return`** statement:

```
#include "stdio.h"
char convert_mark(int mark) {
   if (mark < 40) {return 'C';}
   else if (mark >= 70) {return 'A';}
   else {return 'B';}
}

void main(void) {
  int mark;
  char grade;
  printf("enter the mark: ");
  scanf("%d", &mark);
  grade = convert_mark(mark);
  printf("You got a %c\n", grade);
}
```

[Q: what if some “execution path” doesn’t end up at a **`return`** statement?]

More on `return` statement

A function which does not return a value can still have `return` statement(s):

```
#include <stdio.h>
void printer(int a, int b){
  if (a == b) {return;}
  else if (a > b) {printf("yes\n"); return;}
  else {printf("no\n");} /* no explicit return */
} /* implicit return for printer() */

void main(void){
  int x,y;
  printf("enter value for x: ");
  scanf("%d", &x);
  printf("enter value for y: ");
  scanf("%d", &y);
  printer(x,y);
}
```

More on `return` statement (contd.)

The type of the expression returned must match the called function's return value, or be capable of being converted to it using C's datatype hierarchy (as seen before):

```
#include <stdio.h>
float function1 (void) {
  return 1; /* returned as 1.000000 */
}
int function2 (void) {
  return 3.1416; /* returned as 3 */
}
void main(void) {
  int x;
  float y;
  y = function1(); /* must include the () */
  printf("value of y is %.2f\n", y);
  x = function2(); /* must include the () */
  printf("value of x is %d\n", x);
}
```

Produces the screen output:

```
value of y is 1.00
value of x is 3
```

Function parameters

A **function call** specifies the ***actual parameters*** to be used when the function is executed. These actual parameters are evaluated and used one-by-one as initial values for the function's formal parameters for this execution of the function.

An actual parameter in a function call can be:
- a constant
- a variable
- an expression
- the return value from another function

In all these cases, the actual parameter is evaluated and this value is copied into the corresponding formal parameter of the function being called.

Very very very important – the **value** of the actual parameter is ***unaffected*** by the execution of the called function. The value of the corresponding formal parameter is NOT copied back to the actual parameter when the called function returns (even if they have the same name!).

Function parameters – example

```
#include <stdio.h>

void fn_1(int q){ /* this q is a formal param. for fn_1 */
  q = 100;
  printf("fn_1 just set parameter to %d\n", q);
}

void main(void){
  int q = 3;
  printf("value of parameter is %d\n", q);
  fn_1 (q); /* this q is the actual parameter here */
  printf("now the value of parameter is %d\n", q);
}
```

Produces the screen output:

```
value of parameter is 3
fn_1 just set parameter to 100
now the value of parameter is 3
```

Function parameters (contd.)

If a function has more than 1 parameter, the formal and actual parameters must match in **number**, **type**, and **order**. If not, in general you have problems…

If the types don't match, the value of the actual parameter(s) is converted according to C's datatype hierarchy (as seen before).

```
#include <stdio.h>
int max (int a, int b){
  if (a >= b) {return a;}
  else {return b;}
}
void main(void){
  int x=6, y=3, result1, result2;
  float f1=2.8, f2=4.7;
  result1 = max(x,y);
  printf("result of comparing the integers is %d\n", result1);
  result2 = max(f1,f2);
  printf("result of comparing the floats is %d\n", result2);
}
```

When `max()` is called with actual parameters `x` and `y`, the initial value of `a` is 6 and the initial value of `b` is 3

When `max()` is called with actual parameters `f1` and `f2`, the initial value of `a` is 2 and the initial value of `b` is 4 (due to integer truncation)

Produces the screen output:

```
result of comparing the integers is 6
result of comparing the floats is 4
```

Function prototypes

(another name for “function declarations”)

If you look back at all the programs we’ve examined in the previous 12 slides, you’ll notice that in every case, the functions (other than **`main()`**) were defined ***before*** the definition of **`main()`**. What if this were not the case (since in C, function definitions can occur **in any order** in a program file)? For example:

```
#include <stdio.h>
void main(void) {
  float y;
  y = function1(); /* first mention of function1() */
  printf("value of y is %.2f\n", y);
}
float function1 (void) {
  return 1; /* returned as 1.000000 */
}
```

Note: in C, the default return-type for a function is **`int`**. So when the C compiler first encounters a mention of **`function1()`**, it assumes that its definition is of the form **`int function1(void){…}`** – (this is called an “implicit declaration” of **`function1()`**). ***So what happens when you compile/run this program?***

Function prototypes (contd.)

```
#include <stdio.h>
void main(void) {
  float y;
  y = function1(); /* first mention of function1() */
  printf("value of y is %.2f\n", y);
}
float function1 (void) {
  return 1; /* returned as 1.000000 */
}
```

My C compiler issues the following warning:

```
At top level:
Line 7: warning: type mismatch with previous implicit declaration
Line 4: warning: previous implicit declaration of `function1'
Line 7: warning: `function1' was previously implicitly declared
to return `int‘
```

But it does compile the program. However, the screen output is not as expected:

```
value of y is 0.00
```

Something has gone wrong! The solution – function prototype for `function1()`

Function prototypes (contd.)

Each C function should be declared ***before*** it is called:

```
#include <stdio.h>
void main(void) {
  float y;
  float function1(void); /* prototype for function1() */
  y = function1();
  printf("value of y is %.2f\n", y);
}
float function1 (void) {
  return 1; /* returned as 1.000000 */
}
```

Now the C compiler knows what the definition for **`function1()`** looks like, before it has encountered it. Correct screen output produced: **`value of y is 1.00`**

A function prototype declares what **type of value the function returns**, and what **type(s) of parameters it accepts**. If no return value or parameters – use **`void`**

Optionally, a function prototype may use parameter names in the argument list, e.g.

```
void printer(int x, int y); /* function prototype */
```

As with the formal parameters in a function definition, these names are for documentation only and do not have to match the names of the actual parameters in the function call.

Function prototypes: another example

```
#include <stdio.h>

void main(void) {
int power(int a, int b);
/* function prototype with "dummy" parameters a and b */
int answer = power(3,4) * power(10,2);
printf("the answer is %d\n", answer);
}

int power(int base, int exponent) {
  int result = 1, i;
  for (i = 0; i < exponent; i++) {
    result = result*base;
  }
  return result;
}
```

Produces the correct screen output: **`the answer is 8100`**

Storage class and Scope

Key question: what variables can a particular function access?

Recall example program from slide 12: if we try EITHER of the following changes –

```
#include <stdio.h>
int addThree(int n){
 int m = x+3; /* ILLEGAL! Won’t compile */
 return m;
}

void main(void){
  int x = 20;
  int y = addThree(x);
  printf(“y is %d”,m); /* ILLEGAL! Won’t compile */
}
```

The general rule: A function has access only to variables declared in its function body *(local variables)* and variables in its parameter list.

Storage class and Scope (contd.)

A variable’s ***scope*** refers to the region of a program in which the variable is accessible. Variables defined within a function (including its formal parameters) are **local** to that function, and their scope is that function’s body.

One consequence of this is that different functions can have local variables with the same name – such variables are unrelated, and any operation performed on one such variable has ***no effect*** on other variables of the same name in other functions. See slide 19 for an example of this!

It is possible to define a variable outside the body of any function – such a variable is called a **global variable**, and its scope is from the point of declaration of the variable to the end of the program file.

Since C functions are defined outside of any other function, they are treated as “global identifiers” and are accessible to other functions that come after them in the program file. To make them accessible to all functions in the program – use a function prototype.

Storage class and Scope (contd.)

```
#include <stdio.h>
int count=0; /* global variable */
void main(void) {
  int x, y; /* x and y are local to main() */
  void function1(int, int); /* function prototype */
  x = count + 2;
  printf("value of x in main is %d\n", x);
  count++; /* increase global variable "count" by 1 */
  y = count;
  printf("value of y in main is %d\n", y);
  function1(x,y);
  y = count;
  printf("now value of y in main is %d\n", y);
}
void function1(int a, int b) { /* a and b are local to function1() */
  int x=4; /* this x is a local variable in function1() */
  printf("value of x in function1 is %d\n", x);
  printf("value of a in function1 is %d\n", a);
  printf("value of b in function1 is %d\n", b);
   count--; /* decrease global variable "count" by 1 */
}
```

Produces the screen output:

```
value of x in main is 2
value of y in main is 1
value of x in function1 is 4
value of a in function1 is 2
value of b in function1 is 1
now value of y in main is 0
```

Storage class and Scope (contd.)

Storage class of a variable refers to how long their value is held by the computer.

- Local variables have **automatic** storage class – their value is lost when the function in which they are defined exits.
- Global variables have **external** storage class – their value is held until the program terminates.
- Since C functions are defined outside any other function, they have **external** storage class.

Sometimes you may want a local variable to hold its value until the program terminates. This will be the case if the variable’s declaration is preceded by the keyword **`static`**. Local **`static`** variables hold their value across multiple function calls; any initialisation is done only the first time the function is executed.

Local **`static`** variables are often used to count how many times the function in which they are defined has been executed during the lifetime of the program…

Storage class and Scope (contd.)

```
#include <stdio.h>
void main(void) {
  int i;
  void fn(void); /* function prototype */
  for (i=0; i<5; i++) {
    fn(); /* call to function fn() */
  }
}
void fn(void) {
  static int var_s = 0; /* done once */
  int var_a = 0; /* done every time fn() is entered, */
               /* since var_a has automatic storage */
  printf("var_a = %d, var_s = %d\n", var_a, var_s);
  var_a++; /* increase var_a by 1 */
  var_s++; /* increase var_s by 1 */
}
```

Produces the screen output:

```
var_a = 0, var_s = 0
var_a = 0, var_s = 1
var_a = 0, var_s = 2
var_a = 0, var_s = 3
var_a = 0, var_s = 4
```

A problem on functions

What will be output when the following code fragment is executed ?

```
#include "stdio.h"
int f1(int, int); /* function prototype for f1() */
int f1(int i, int j){
        return (i+j);
}
int f2(int i){
        int j = f1(i, i-1);
        return j;
}
void main(void){
int f2(int); /* function prototype for f2() */
printf("result of function call is %d\n", f1(4,f2(4)) );
}
```