
UNSW
SYDNEY
Australia's
Global
University
COMP1911 - Computing 1A

7. Functions

In this lecture we will cover:

- Why we need functions
- Type of functions
- How to use functions
- Scope
- How to write a function

Functions

Have you seen C functions before?

Have you seen C functions before in this course?

Total Results: 0

Functions

Have you seen C functions before?

Yes! We've been working with the main function from the very beginning, as well as library functions such as, printf, scanf, etc., which are also functions.

Definition

A function is 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

Using Functions

To use a function, we don't need to see all the code for it. We just need to know the name of the function and how to use it.
To do this we need to know the function's **prototype**
Here are some examples from the math.h library

```
double sqrt(double x);
```

```
//returns x to the power of y
double pow(double x, double y);
```

There are many more!

And the stdlib.h library

```
int abs(int x);
```

The Type of Functions

Parameters and Return Value

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

The type of the function is the type of the value they return

Consider:

```
double pow(double x, double y);
```

This is called a function prototype and it gives us (and the compiler) information about how we can use the function!

This function :

- has the name pow
- expects two arguments (x and y), both of type double
- has a return value of type double

Using Functions

Here is a simple program using two library functions:

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

int main(void) {
  int a = -199;
  int result = abs(a);
  printf("Absolute Value of %d is %d\n",a,result);
  printf("The square root of 2 is %lf\n",sqrt(2));
  return 0;
}
```

Note: We do not need to see the actual code written to implement the function use it. Our code is easier to read because we can read the function name such as sqrt and understand it. If we saw the actual code for sqrt it would be much more confusing!!!

Using Functions

scanf has a return value that we have been ignoring. It tells us how many items were successfully read in!

This could be 0 if a matching failure occurred or -1 (EOF) if there was no more input to read such as when we reach the End Of a File or interactively when we type Ctrl^ d

```c
#include <stdio.h>
int main(void) {
    int x,y,numbersRead;

    numbersRead = scanf("%d %d",&x,&y);
    if(numbersRead != 2){
        printf("Invalid input\n");
    }
    return 0;
}
```

Using Functions

We can use this return value to keep reading while we get integer input

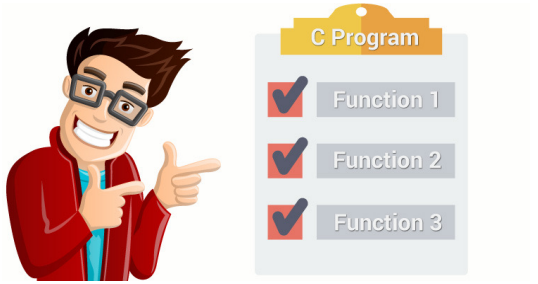
```c
#include <stdio.h>

int main(void) {
    int x;

    while(scanf("%d", &x ) == 1){
        printf("I read 1 integer %d\n", x);
    }
    return 0;
}
```

User Defined Functions

We can also define our own functions!



- Functions allow code reuse.
  Define a function once, call it many times.
  Makes program much easier to change!
- Functions allow code modularisation.
  Interaction with the rest of program is explicit and limited
  We can consider the code in isolation
  Much easier to read, test and debug!
- Breaking up your code into function is essential to reducing the complexity of your programs

Function Prototypes and Definitions

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

Function Prototypes are needed above the main function, because the compiler processes the program code sequentially. It lets the compiler know that the definition (implementation) for these functions can be found somewhere else.

A compile error occurs if a function call is encountered before the function prototype.

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

Example: Function Prototype and Definition

```c
#include <stdio.h>

//function prototype
double answer(int x);

int main(void) {
    //Calling(using) the function
    printf("The answer is %lf\n", answer(2));
    return 0;
}

//function definition/implementation
double answer(int x) {
    return x * 0.5;
}
```

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;
- Reaching the last statement of the function or reaching a **return** statement stops execution of a function
- When the function completes, execution resumes at a point after the function call.

Function Variables

- function have their own variables created when function called and destroyed when function returns
- function's variables are not accessible outside the function

Structure of a Function

1. Return type
2. Function name
3. Parameters (inside brackets, comma separated)
4. Variables (only accessible within function)
5. Return statement

Click on Return type.

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

Total Results: 0

Click on Variables.

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

Total Results: 0

Structure of a Function

1. Return type
2. Function name
3. Parameters (inside brackets, comma separated)
4. Variables (only accessible within function)
5. Return statement

```
int addNumbers(int num1, int num2) { // 1, 2, 3
    int sum; // 4
    sum = num1 + num2;
    return sum; // 5
}
```

Different Types 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:

```
double foo1(int i, double d);
int    foo2(double x, double y, double z);
int    foo3(void);
void   foo4(int);
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.

The return Statement

The return statement 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.

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

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.

Functions with No Return Value

Some functions do not compute a value.

- They have a return type of void
- They are useful for "**side-effects**" such as output

```
void printAsterisks(int n) {
    int i = 0;
    while (i < n) {
        printf("*");
        i = i + 1;
    }
    printf("\n");
}
```

These functions do not need to return a value. The use of return to terminate the function is optional but not recommended.

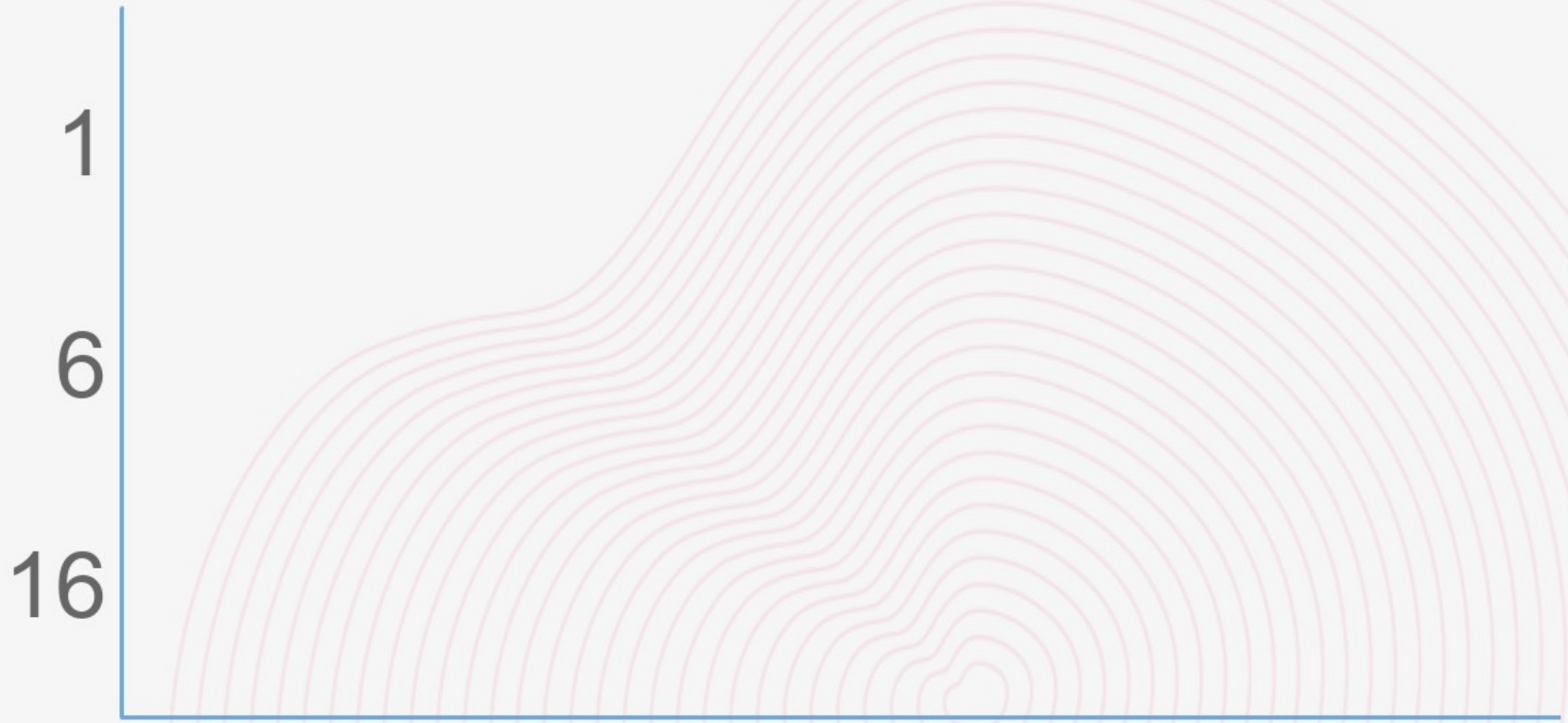
Return Value

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

```
int main (void) {
      int n = 1;
      n = f(n) + 10;
      printf("%d\n", n); // what is printed?
}

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

What is printed?



Total Results: 0

Return Value

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

```
int main (void) {
      int n = 1;
      n = f(n) + 10;
      printf("%d\n", n); // what is printed?
}

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

printf() 16, explain?

f(n) + 10 → f(1) + 10 → 6 + 10 → 16.

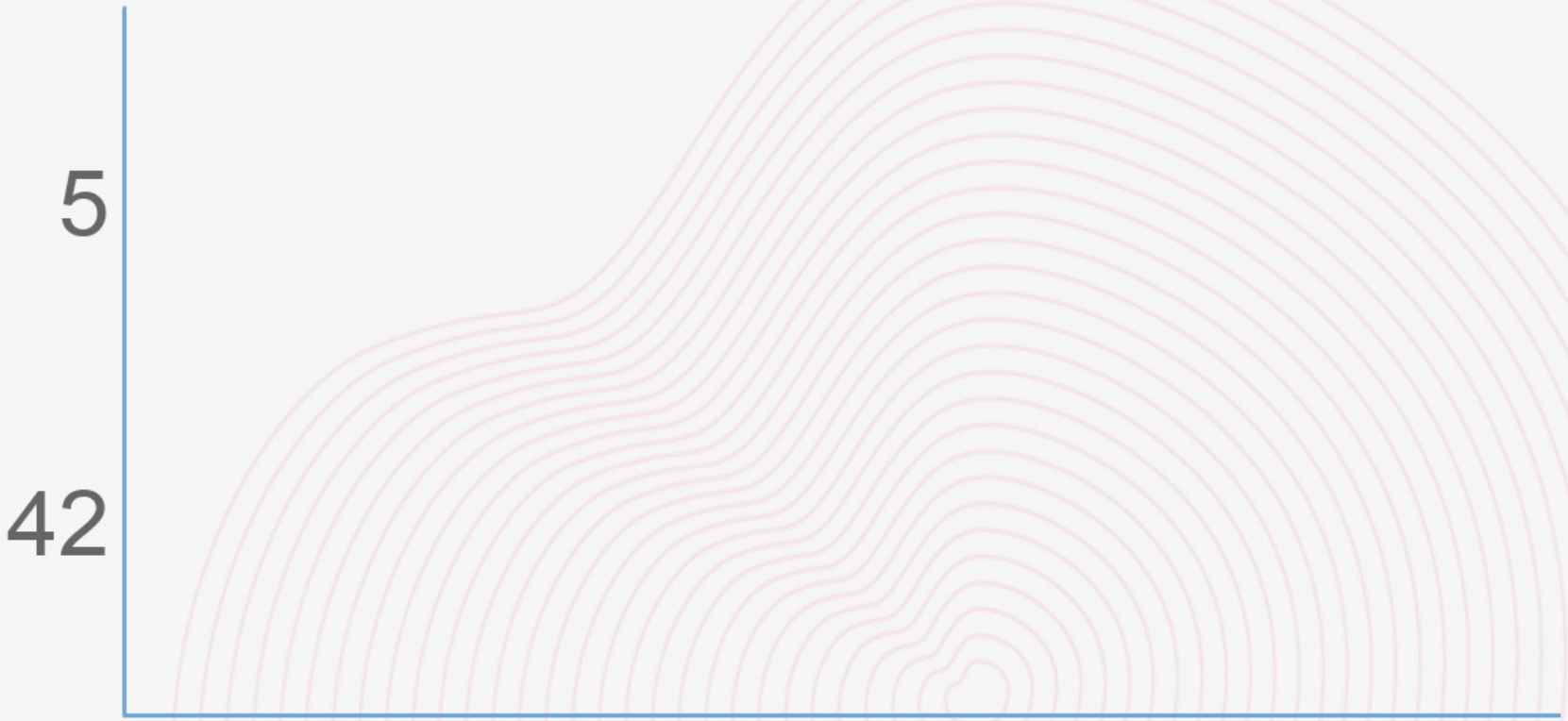
Argument Passing: Standard Variables

Most (all types that we have seen so far) arguments in C are passed by value. Consider:

```
int main(void) {
      int n = 5;
      f(n);                  // pass n as argument
      printf("%d\n", n); // what is printed?
}

void f(int n) {
       n = 42
}
```

What is printed?



Total Results: 0

Argument Passing: Standard Variables

Most (all types that we have seen so far) arguments in C are passed by value. Consider:

```
int main(void) {
      int n = 5;
      f(n);                  // pass n as argument
      printf("%d\n", n); // what is printed?
}

void f(int n) {
       n = 42
}
```

printf() → 5. Why?

- A copy of the value of n is passed to the f()
- f() is modifying its local copy and the original n is not altered.

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 of 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.

Scope - example

```c
int main(void){

    int i = 0;
    while (i < 10) {
        int j = 0;
        if (i > 5) j = 100;
        printf("i is %d and j is %d\n", i, j);
        i = i + 1;
    }
    int j = 10;
    printf("i is %d and j is %d\n", i, j);
/*
    i = 0;
    while (i < 10) {
        int j = 0;
        if (i > 5) j = 100;
        printf("i is %d and j is %d\n", i, j);
        i = i + 1;
    }*/
    return 0;
}
```

Find errors.

```c
int n = 5;
int bar(int n);

int main(void) {
 int i = j;
 int j = n;
 foo();
 return 0;
}
void foo(void) {
 int n = 10;
 int j = i;
 bar(n);
}
int bar(int n) {
 return (n + 5);
}
```

Total Results: 0

Scope Example

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

int main(void) {
 int i = j;       // illegal, j not in scope
 int j = n;       // OK, uses global n
 foo();           // illegal, what is foo?
 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);
}
```

Design and Abstraction

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 apply top-down design and abstraction to a simple problem:
Read two integers and print out the smallest.

What are the logical tasks?

- read two integers
- 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:

- find the minimum of two integers

We already abstract over the details of reading two integers by using scanf. If we wanted to do error checking or more complex input we could consider putting this into a function.

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

So let's assume we had a function to find the minimum of two integers.

What prototype would it need?

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

So how do we use this?

Calling a function - Example

Well, let's write our program by using our function!

```c
#include <stdio.h>

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

int main(void) {
  int n, m, min;
  printf("Enter two integers: ");
  scanf("%d %d",&n,&m);

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

  return 0;
}
```

Defining a Function - Example

Now we can actually write our function

```
// Find and return the minimum of two integers
int minInt(int n1, int n2){
    int result;
    if( n1 <= n2){
        result = n1;
    } else {
        result = n2;
    }
    return result;
}
```

It is good style to define your main function at the top of the file and other user defined functions underneath it.

Questions