last update: 11th February 2024

Session: 2023-24

COMP281 Lecture 5

Principles of C and Memory Management

Phil Jimmieson

UNIVERSITY OF LIVERPOOL

Department of Computer Science

Last Lecture

- C Language Basics
  - Operators
  - Decision Making
  - Loops
  - Functions
- CodeGrade

Last Lecture

- C Language Basics
  - Operators
  - Decision Making
  - Loops
  - Functions
- CodeGrade

Recap

Today

- C Language Basics
  - Operators
  - Decision Making
  - Loops
  - Functions
- CodeGrade

Functions

What is a Function?

Function

“A fragment of code that accepts zero or more ***argument values***, produces a ***result value***, and has zero or more ***side effects***”.

Functions - going back to basics…

- A function can be used to call or execute multiple statements at one time.

We, as a coder, often ask only one thing

The computer goes through each statement one at a time

An example

- How to put an elephant inside a fridge?

I’m the elephant.

I’m the fridge.

An example

- How to put an elephant inside a fridge?

1. Open the fridge
2. Put the elephant in the fridge

An example

- How to put an elephant inside a fridge?

1. Open the fridge → `openFridge( );`
2. Put the elephant in the fridge → `putItIn(elephant);`

An example

- How to put an elephant inside a fridge?

1. Open the fridge
2. Put the elephant in the fridge

```
int putInFridge(elephant)
{
    openFridge( );
    putItIn(elephant);
    return 0;
}
```

What is a function?

- A shorthand to calling a bunch of statements.
  - We could use it again and again

What is a function?

- A shorthand to calling a bunch of statements.
  - We could use it again and again, e.g., put 3 elephants in a fridge:

```
int main(void)
{
    putInFridge(elephantA);    1
    putInFridge(elephantB);    2
    putInFridge(elephantC);    3
    return 0;
}
```

What is a function?

- A shorthand to calling a bunch of s
  - We could use it again and again, e.g.

```
int main(void)
{
    putInFridge(elephantA
    putInFridge(elephantB
    putInFridge(elephantC
    return 0;
}
```

```
int main()
{
    /* if not using a function */
    openFridge( );
    putItIn(elephantA);

    openFridge( );
    putItIn(elephantB);

    openFridge( );
    putItIn(elephantC);

    return 0;
}
```

An example

- How to put an elephant inside a fridge?

What if we want to improve our code?

1. Open the fridge
2. Put the elephant in the fridge

An example

- How to put an elephant inside a fridge?

What if we want to improve our code? -- An improved version

1. Open the fridge
2. Put the elephant in the fridge
3. Close the fridge

```
int main(void)
{
    putInFridge(elephantA);
    putInFridge(elephantB);
    putInFridge(elephantC);
    return 0;
}
```

```
int main(void)
{
    putInFridge(elephantA);
    putInFridge(elephantB);
    putInFridge(elephantC);
    return 0;
}
```

change only in one location

```
/* if using a function */
int putInFridge(elephant)
{
    openFridge( );
    putItIn(elephant);
    closeFridge( );
    return 0;
}
```

```
int main(void)
                                    /* if using a function */
{
    putInFridge(elephantA);
    putInFridge(elephantB);
    putInFridge(elephantC);
    return 0;
}
```

All will be updated

```c
int main(void)
{
    putInFridge(elephantA);
    putInFridge(elephantB);
    putInFridge(elephantC);
    return 0;
}
```

```c
int main()
{
    /* if not using a function */
    openFridge( );
    putItIn(elephantA);
    closeFridge( );

    openFridge( );
    putItIn(elephantB);
    closeFridge( );

    openFridge( );
    putItIn(elephantC);
    closeFridge( );

    return 0;
}
```

Have to change in many locations

```
{
    openFridge( );
    putItIn(elephant);
    closeFridge( );
}
```

→ `putInFridge(elephant);`

Extracting the code as a function

-> then we only have **one copy** to call multiple times

```
{
    openFridge( );
    putItIn(elephant);
    closeFridge( );
}
```

```
putInFridge(elephant);
```

argument(s)

```
int main(void)
{
    putInFridge(elephantA);
    putInFridge(elephantB);
    putInFridge(elephantC);
    return 0;
}
```

```
putInFridge(elephant);
```

passing argument(s) to a function

```
int main(void)
{
    putInFridge(elephantA);
    putInFridge(elephantB);
    putInFridge(elephantC);
    return 0;
}
```

```
putInFridge(elephant);
```

passing argument(s) to a function

Arguments allow us to customise functions

```
putInFridge(elephantA);
putInFridge(elephantB);
…
putInFridge(elephantX);
```

→ Put different elephants into the fridge

The `putInFridge` function does something different, depending upon what argument(s) we pass in.

Arguments allow us to customise functions

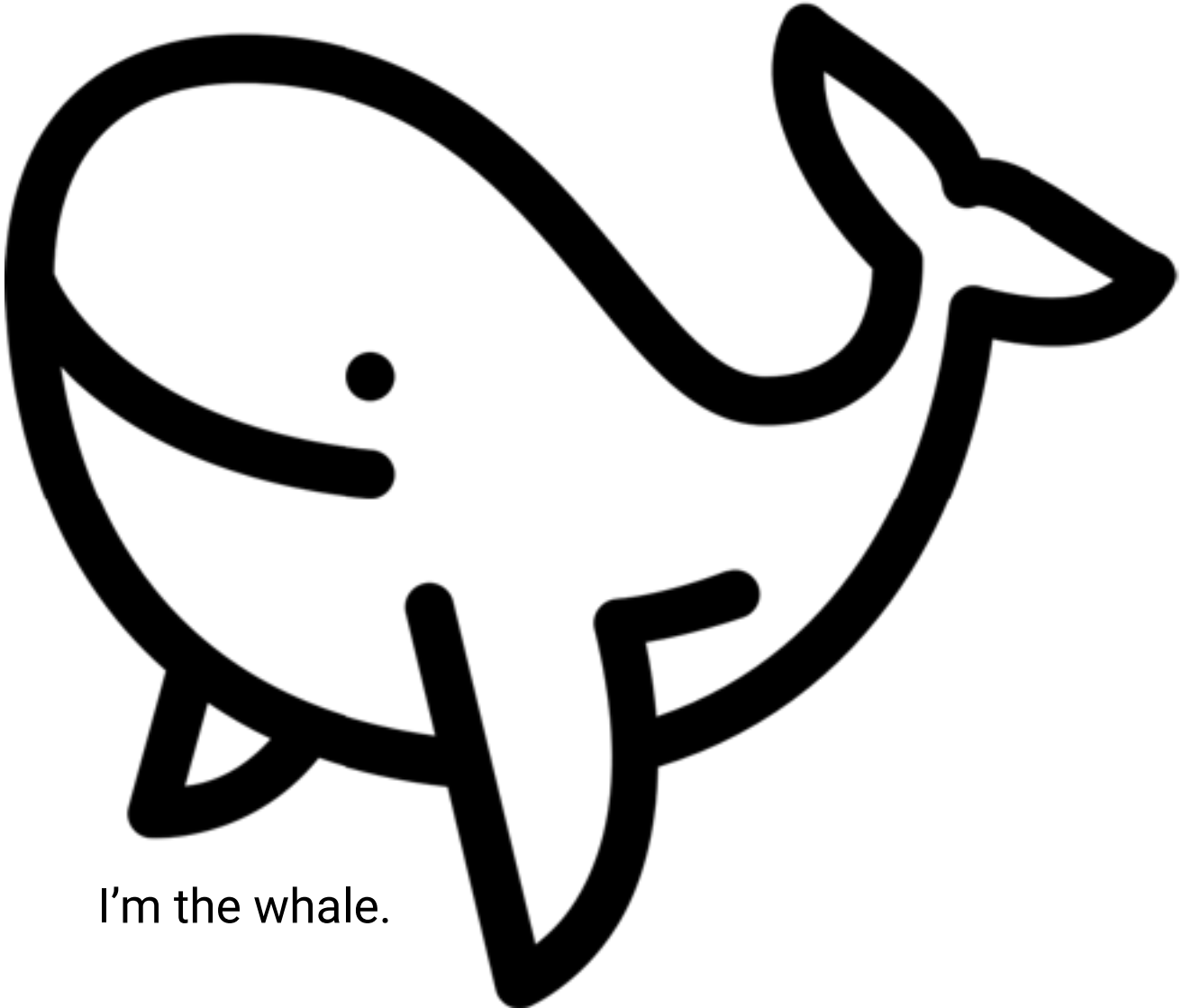
what happens to arguments passed to functions?

```
int main(void)
{
    char elephantA[8] = "Stampy";
    printf("%s\n",elephantA);
    putInFridge(elephantA);
    printf(“%s\n",elephantA);
    return 0;
}
```

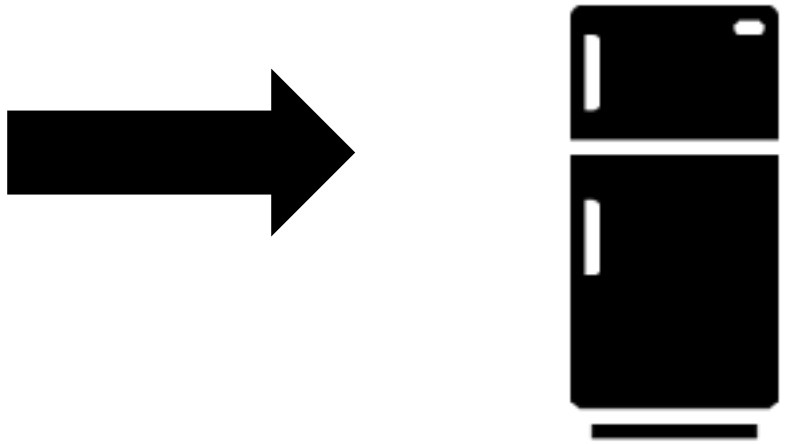
```
int putInFridge(char elephant[ ])
{
    openFridge( );
    putItIn(elephant);
    elephant = "Dumbo";
    return 0;
}
```

Exercise

- How to put a whale inside a fridge?



I’m the whale.



I’m the fridge.

Exercise

- How to put a whale inside a fridge?

```
int putInFridge(animal)
{
    openFridge( );
    putItIn(animal);
    closeFridge( );
    return 0;
}
```

Exercise

- How to put a whale inside a fridge?

```
int putInFridge(animal)
{
    openFridge( );
    putItIn(animal);
    closeFridge( );
    return 0;
}
```

```
int main(void)
{
    putInFridge(theWhale);
    return 0;
}
```

Exercise

- How to put a whale inside a fridge?

```
int putInFridge(animal)
{
    openFridge( );
    putItIn(animal);
    closeFridge( );
    return 0;
}
```

```
int main(void)
{
    putInFridge(theWhale);
    return 0;
}
```

Exercise

- How to put a whale inside a fridge?

```
int putInFridge(animal)
{
    openFridge( );
    putItIn(animal);
    closeFridge( );
    return 0;
}
```

```
int main(void)
{
    takeOutAnElephant( );
    putInFridge(theWhale);
    return 0;
}
```

Exercise

- How to put a whale inside a fridge?

```
int putInFridge(animal)
{
    openFridge( );
    putItIn(animal);
    closeFridge( );
    return 0;
}
```

```
int main(void)
{
    while( hasAnElephant( ) )
    {
        takeOutAnElephant( );
    }
    putInFridge(theWhale);
    return 0;
```

Summary - why Use Functions

- It provides **modularity** to your program's structure.
- It makes your code **reusable**. You just have to call the function by its name to use it, wherever required.
- In case of large programs with thousands of code lines, **debugging and editing** becomes easier if you use functions.
- It makes the program more **readable** and easy to understand.

Function Categories

- Library functions
  - Already defined in C Libraries
- User-defined functions
  - Defined by programmers

Some Common Library Functions

```
#include <math.h>
   - sin(x) // radians
   - cos(x) // radians
   - tan(x) // radians
   - atan(x)
   - atan2(y,x)
   - exp(x) // e^x
   - log(x) // log_e x
   - log10(x) // log_10 x
   - sqrt(x) // x ≥ 0
   - pow(x, y) // x^y
   - ...
```

```
#include <stdio.h>
   - printf()
   - fprintf()
   - scanf()
   - sscanf()
   - ...
#include <string.h>
   - strcpy()
   - strcat()
   - strcmp()
   - strlen()
   - ...
```

Function Definition

Syntax

```
returnType functionName(type1 parameter1,...)

{

    // function body goes here

}
```

Syntax

Function Header → `returnType functionName(type1 parameter1,...)`

Function Body →

```
{
        // function body goes here
}
```

Syntax

Function Header → `returnType functionName(type1 parameter1,...)`

- Return type
- Function name
- Parameter list

Parameter vs **Argument**

- *Formal Parameter:* a declaration of an identifier within the ( ) of a function declaration
  - Used within the body of the function as a *variable* of that function
  - Initialised to the value of the corresponding ***argument***.
- *Argument:* an expression passed when a function is *called*; becomes the initial value of the corresponding **formal parameter**
- Arguments are also sometimes called *actual parameters*

An example

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f(` $\texttt{expr}_1$`,` $\texttt{expr}_2$ `)` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

An example

a parameter

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f(` $\texttt{expr}_1$`,` $\texttt{expr}_2$ `)` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

an argument

An example

another parameter

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f(` $\texttt{expr}_1$`,` $\texttt{expr}_2$ `)` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

an argument

An example

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f( expr`$_1$`, expr`$_2$` )` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

1

The 1st ***argument*** expression is evaluated, converted to `double`, and assigned to ***parameter*** `x`

An example

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f( expr`$_1$`, expr`$_2$` )` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

2

The 2nd ***argument*** expression is evaluated, converted to `int`, and assigned to ***parameter*** `a`

An example

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f( expr`$_1$`, expr`$_2$` )` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

3

Function `f` is executed and ***returns*** a value of type `int`

An example

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f(` $\texttt{expr}_1$`,` $\texttt{expr}_2$ `)` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

4

Result of `f` is added to `d`

An example

- Let `int f( double x, int a ){` be (the beginning of) a **definition** of a function.

- Then `f(` $\texttt{expr}_1$`,` $\texttt{expr}_2$ `)` can be used in *any* expression where a *value* of type `int` can be used, e.g.,

```
N = f( pi * pow( r, 2 ), b + c ) + d;
```

5

Sum is assigned to `N`

Return values

Syntax

Function Header → `returnType functionName(type1 parameter1,...)`

Function Body →

```
{

    // function body goes here

}
```

Syntax

Function Header → `returnType functionName(type1 parameter1,...)`

Function Body →

```
{
    // function body goes here
    return <value of type ReturnType>;
}
```

The main() function - a quick aside

- As we already saw, main() in C is the entry point for your app and is defined as returning an Int result.
- On some operating systems the result from main() is used to determine if an error has occurred in your program (returning a zero, indicates no error)
- C 99 and later versions now implicitly return a zero value from main()
- To me this is bad programming practice. If a function is declared as returning a result, the programmer's code should do that and should be **seen** to do it.
- If you have a function defined as returning a non-void result and do not include the appropriate return statement in your code, it will be judged as being in error.

Function Declaration

Syntax

```
returnType functionName(type1 parameter1,type2 parameter2,...);
```

- **Function declaration** consists of 4 parts:
  - Return type
  - Function name
  - Parameter list
  - *Terminating semicolon*

Syntax

```
returnType functionName(type1 parameter1,type2 parameter2,…);
```

- **Function declaration** informs the compiler about the function name, parameters to accept and its return type.
- A function must be *declared* **before** it's used.
- The actual body of the function can be defined **separately**. This is called **Function Prototyping**.

Function Declaration / Function Prototype

- When might a function be used separately from where it is defined?
  - ***before*** its definition in the same C program
  - In one or more completely **separate** C programs
- This is actually the normal case!

Function Declaration / Function Prototype

- When might a function be used separately from where it is defined?
  - ***before*** its definition in the same C program
  - In one or more completely **separate** C programs
- This is actually the normal case!
- So, we need some way to ***declare*** a function separate from ***defining*** its body
  - called ***Function Prototype***

Function Declaration / Function Prototype

- *Function Prototype* is a language construct of the form:

```
returnType functionName (parameter declarations);
```

- Exactly like the function **definition**, except with a `;` instead of a body in `{ }`

Purpose

- So that compiler knows how to compile calls to that function, i.e.,
  - number and types of arguments
  - type of result
- As part of a “**contract**” between developer and programmer who uses the function
- As part of hiding details of *how* it works and exposing *what* it does.
- A function serves as a “black box.”

Header Files

- In applications with multiple *C* programs, function prototypes are typically provided in *header files*
  - i. e., the `.h` files that programmers include in their code
- Grouped by related functions and features
  - To make it easier for developers to understand
  - To make it easier for team development
  - To make a package that can be used by someone else

Note - #include

```
#include <foo.h>
```

- Search the system's directories

```
#include "foo.h"
```

- Search the directory where the source program is found first, *before* the system directories

Note - Typical C Programming Style

- A lot of small *C* programs, rather than a few large ones
  - Each **.c** file contains closely related functions
  - Usually a small number of functions
- Header files to tie them together
- `Makefiles` to build or rebuild them in an organised way

Note - Typical C Programming Style

- C Functions don’t allow other functions to be declared within them
  - Like C++, Java
  - Unlike Algol, Pascal
- All functions are defined at the “top level” of C programs
  - (Usually) visible to the linker
  - Can be linked by any other program that knows the function prototype

Function Declaration vs **Function Definition: In Summary**

```
int foo();
```

A function **declaration** does not provide a function body, but it tells the *compiler* that it can use this function and to expect that it **will be defined** somewhere else.

```
int foo()
{
    // statements;
    return 0;
}
```

A function definition provides all the necessary information to create that function in its **entirety**.

Function Declaration vs **Function Definition: In Summary**

Once a function is **defined**, that also counts as **declaring** it; so you can often both **declare** and **define** a function, or a variable at the same time. But you don't have to.

```
int foo( );
int main( void )
{
   int x = foo( );
   return 0;
}
int foo( )
{
   return 281;
}
```

- Since the compiler *knows* the **return value** of `foo`, and the number of **arguments** it takes, it can compile the call to `foo` even though it *doesn't yet* have the definition.
- In fact, the definition of the method `foo` could go into another file!

Calling a Function

Calling a Function

- When a program calls a function, the **program control** is *transferred* to the called function.
- A called function performs a *defined task* and when its return statement is executed or when its function-ending closing brace is reached, it *returns* the **program control** back to the main program.

Calling a Function

- Pass the required parameters with the function name, and if the function returns a value, then store the returned value. For example:

```
#include <stdio.h>

/* function declaration */
int max(intnum1,intnum2);

int main(void) {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a,b);

   printf("Max value is : %d\n",ret);
   return 0;
}
```

```
/* function returning the max
between two numbers */
int max(intnum1,intnum2) {
   /* local variable declaration */
   int result;

   if(num1>num2)
      result=num1;
   else
      result = num2;

   return result;
}
```

Output:

```
Max value is : 200
```

Summary

Today

- Function
  - Concept of Function
  - Function Declaration and Prototype
  - Function Definition
    - Syntax
    - Parameter and Argument

Next

- Array
- Debugging