COMP1511/1911 Programming Fundamentals

Week 3 Lecture 1

Functions and Style

Last Week

- if statements
- scanf returns!
- while loops
- nested while loops
- structs

This Week

- Lab 2 due tonight 8pm.
- Lab 3 due next week
- Help Session Schedule
- Assignment 1 out early next week!

Today’s Lecture

- Recap of nested while loops, structs,
- enums
- structs and enums
- Functions
- Style

Most students start to find things are getting hard this week

Be patient and keep practicing.

Link to Week 3 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/24T3/live/week_3/

A Brief Recap

- Nested While Loops

  `pattern.c`

  `clock.c` (solution for you to look at once you have tried to implement it yourself)
- Structs

  `struct_student.c`

  `struct_points.c`

Enumerations

- Data types that allow you to assign names to integer constants to make it easier to read and maintain your code
    - By default the enumerated constants will have int values 0, 1, 2, …
    - Note you can’t have two enums with the same constant names

```
// Example of the syntax used to define an enum
enum enum_name {STATE0, STATE1, STATE2, ...};

// E.g. define an enum for day of the week
enum weekdays {MON, TUE, WED, THU, FRI, SAT, SUN};

// E.g. define an enum with specified int values
enum status_code {OK = 200, NOT_FOUND = 404};
```

enum code example

```c
// Define an enum with days of the week
// make sure it is outside and before the main function
// MON will have value 0, TUE 1, WED 2, etc
enum weekdays {MON, TUE, WED, THU, FRI, SAT, SUN};

int main (void) {
    enum weekdays day;
    day = SAT;
    // This will print out 5
    printf("The day number is %d\n", day);
    return 0;
}
```

enum vs #define

- enums are useful when we want to define a specific fixed set of constants
- The advantages of using enums over #defines
    - Enumerations are automatically assigned values, which makes the code easier to read
        - Think of the case where you have a large number of related constants
- #define are useful for other contexts such as constants that are not integers or stand alone constant values

struct with enum members!

- We can have enum members in our structs!

```c
enum student_status {
    ENROLLED, WITHDRAWN, LEAVE
};

struct student {
    enum student_status status;
    double wam;
};
```

```c
struct student z123456;
z123456.status = ENROLLED;
z123456.wam = 95.9;
```

Coding Example:

`pokemon.c`

- We can have enum members in our structs!
- Create a enum for pokemon types FAIRY, WATER, FIRE etc
- Create a struct called pokemon with a field for the type and some other relevant fields
- Make a pokemon variable and set it with data

Have you seen functions before?

Functions

- Yes you have seen functions before!
- You have been writing main functions
- You have also used functions
    - printf and scanf
- But what is a **function**?
- And will we finally find out what **`void`** and **`return`** really mean?
- And can we start writing our own functions now instead, instead of writing all of our code in the main function?

What are Functions. Why do we use them?

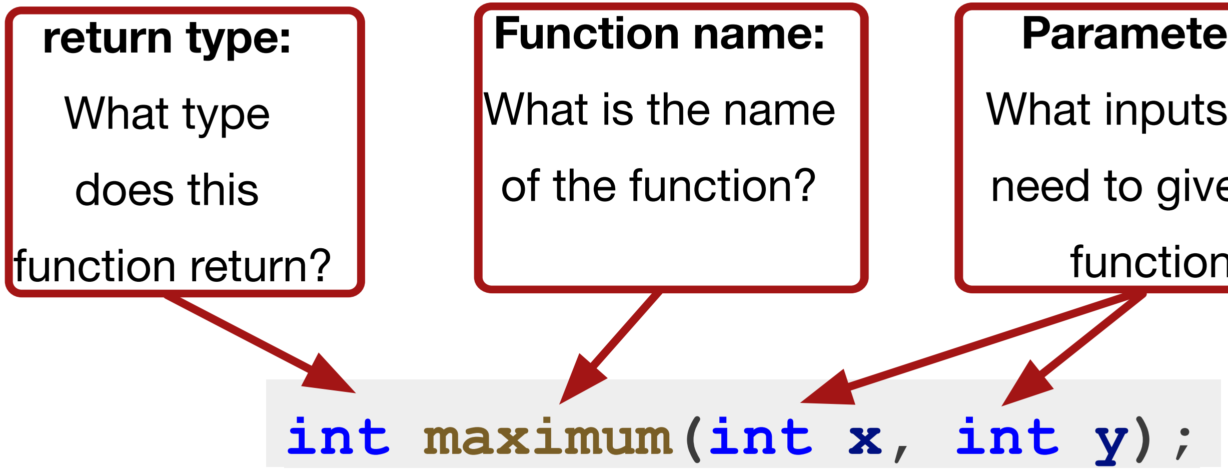
- A function is an independent, reusable block of code that performs a specific task
- The benefits of functions are:
  - Modularity: Breaks complex programs into simpler, manageable pieces, easier to read and understand
  - Reusability: Avoids code duplication, as you can reuse the functions
  - Abstraction: Hides the implementation details and allows you to focus on higher-level logic.
  - Allow us to test and debug smaller chunks of code in isolation

Functions

- Functions have **parameters**
    - Parameters define what type of arguments (inputs) the functions need
    - Functions with `void` in its parameter list needs no arguments (inputs)
- Functions may **return** a single value
    - The type of the function is the type that it returns
    - A return type of `void` means the function does not return a value
        - It can still use `return` to end the function without giving it a return value

Function Prototypes

- This is a function prototype
  - it gives programmers and the compiler information about how the function can be used


return type:
What type does this function return?
Function name:
What is the name of the function?
Parameters:
What inputs do I need to give my function

```c
int maximum(int x, int y);
```

Using Functions

- We do not need to see the implementation code of a function to use it
  - For example you have not seen the implementation of printf or scanf but you know how to use them.
- When we want to use a function, we do a **function call**
  - We must pass in the correct sequence of arguments of the correct type in the correct order
  - If our function has a return value we may wish to use or store it

Functions calls

Examples of calling functions with various prototypes:

```c
int maximum(int x, int y);
void print_stars(int number_of_stars);
void print_warning(void);

int main(void) {
    int num = 7;
    // Store the return value in a variable to use later
    int max = maximum(10, num);
    print_stars(max);
    print_warning();
    return 0;
}
```

Function Definition

You will also need to implement your functions

```c
// This function returns a value of type int
int maximum(int x, int y) {
    int max = x;
    if (x < y) {
        max = y;
    }
    // returns an int value
    return max;
}
```

Function Definitions

```c
// This function does not need a return
// statement since its return type is void
void print_stars(int number_of_stars) {
    int i = 0;
    while (i < number_of_stars) {
        printf("*");
        i = i + 1;
    }
    printf("\n");
}
```

Function Definitions

```c
// This function does not need a
// return statement since its return type
// is void
// It takes no inputs as the parameter list
// is also void
void print_warning(void) {
    printf("#########################\n");
    printf("Warning: Don't plagiarise\n");
    printf("#########################\n");
}
```

Function Calls and Execution Flow

- The code of a function is only executed when requested via a function call
- When a function is called
  - 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 the instruction after the function call.

Prototypes and Style

- It is good style to have
  - main function at the top of the file
  - implement additional user defined functions below it.
- To do this we need to write prototypes above main function
  - the compiler processes the program code top-down
  - This 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.

Function Comments and Style

- Every function must have a comment placed before the function implementation describing
  - the purpose of the function
  - any side-effects the function has
- As always, choose meaningful names for your functions

Quick Break

Code demo

area_triangle.c

print_pokemon.c

Memory and Scope

- Blocks of code in C are delimited by a pair if braces {}.
    - The body of a function is a common example of a block.
- Generally the scope of a variable is
    - Between where the variable is declared
    - The end of the block it was declared in
- Variables declared inside functions are called local variables.

Functions and Local Variables

- Local variables are created when the function called and destroyed when function returns
- A function's variables are not accessible outside the function

```
double add_numbers(double x, double y) {
    // sum is a local variable
    double sum;
    sum = x + y;
    return sum;
}
```

Global Variables

- Variables declared outside a function have global scope
    - Do NOT use these!

```
// result is a global variable BAD DO NOT USE IN COMP1511
int result;
int main(void) {
    // answer is a local variable GOOD
    int answer;
    return 0;
}
```

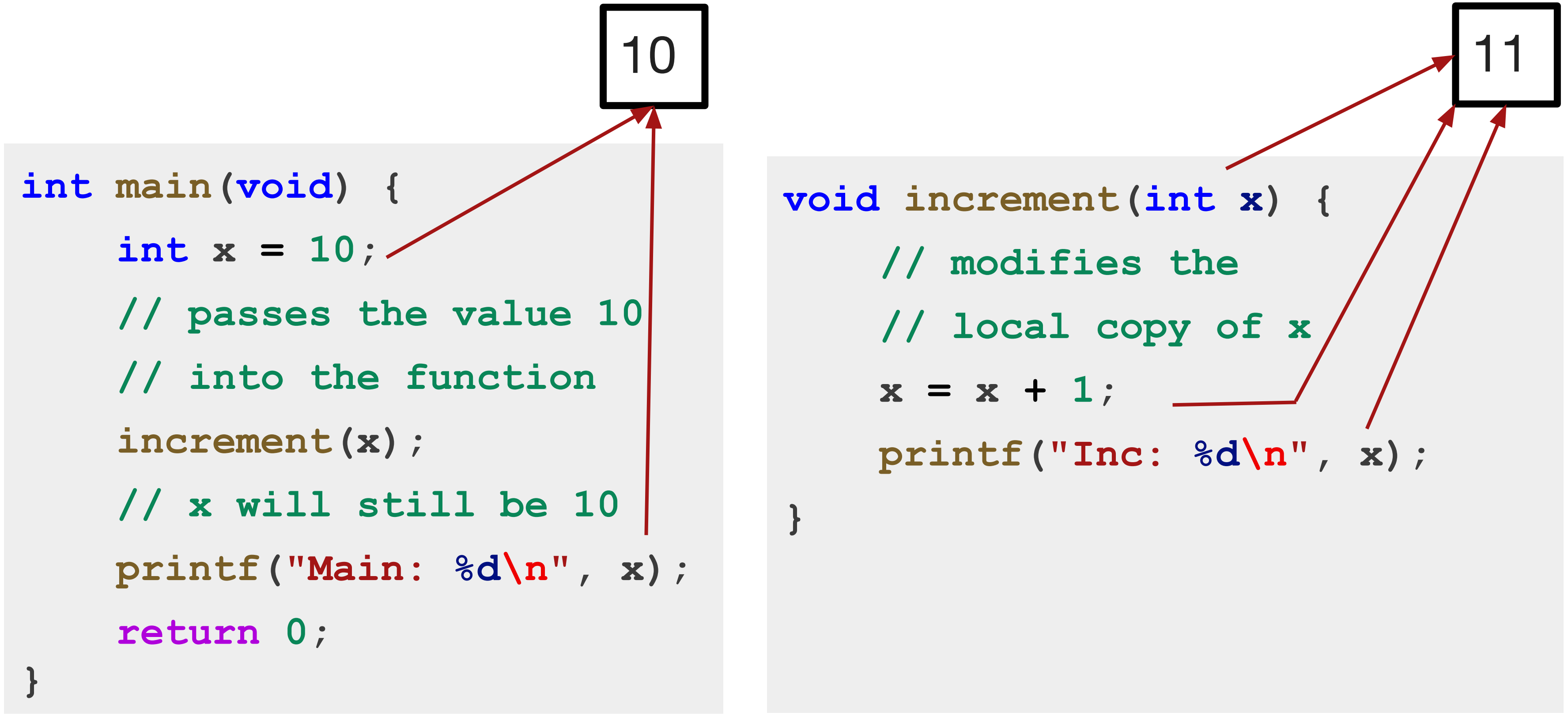
Passing by Value

- Primitive types such as int, char, double and also enum and structs are passed by value
    - A copy of the value of the variable is passed into the function

E.g. This increment function is just modifying its own copy of x

```
void increment(int x) {
    // modifies the
    // local copy of x
    x = x + 1;
}
```

Passing by Value



```c
int main(void) {
    int x = 10;
    // passes the value 10
    // into the function
    increment(x);
    // x will still be 10
    printf("Main: %d\n", x);
    return 0;
}
```

```c
void increment(int x) {
    // modifies the
    // local copy of x
    x = x + 1;
    printf("Inc: %d\n", x);
}
```

Using Functions in Conditions

One way to check that scanf() successfully scanned data is to do something like :

```
int scanf_return;
scanf_return = scanf("%d", &n);
while (scanf_return == 1) {
    ...
    scanf_return = scanf("%d", &n);
}
```

You can call functions inside your if statements or your while loops like this:

```
while (scanf("%d", &n) == 1) {
    ...
}
```

Note: You can't do this with functions that have void return types

Style

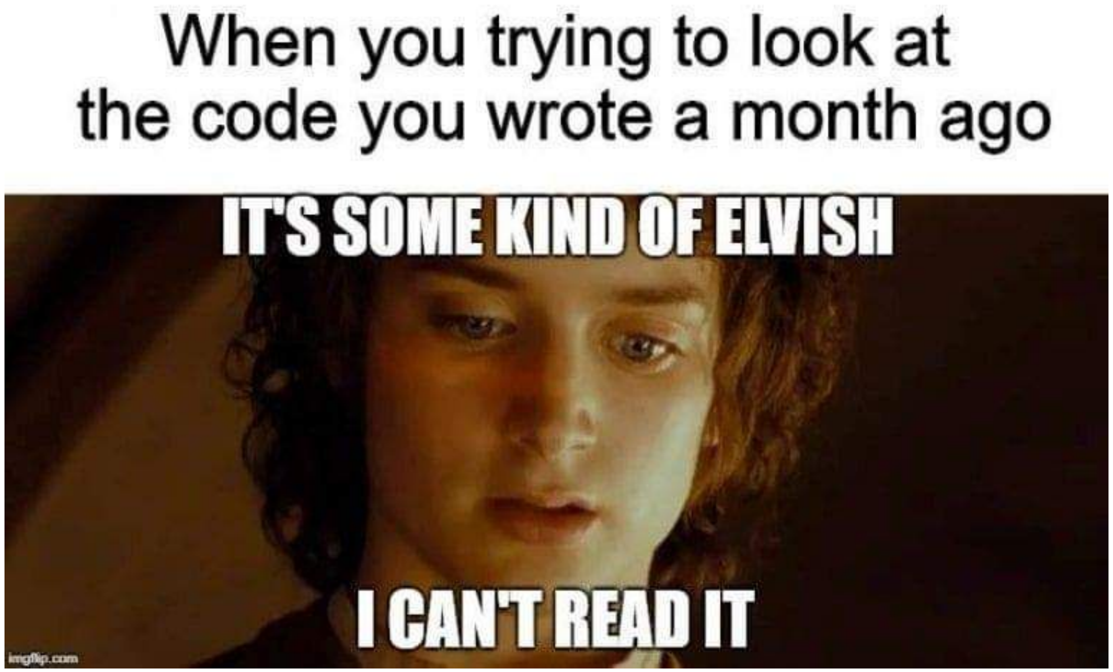
What is Style? Why Style?

- The code we write is for human eyes
- We want to make our code:
    - easier to read
    - easier to understand
- Coding should always be done in style - it is worth it…
    - ensures less possibility for mistakes
    - ensures faster development time
    - You also get marks for style in assignments
    - If we need to mark your code in the final manually it is good if it is not a dog’s breakfast

What is Good Style?

- Indentation and Bracketing
- Names of variables and functions
- Structuring your code
- Nesting
- Repetition
- Comments
- Consistency

Bad Style Demo



Let's look at bad_style.c

- What are some things we should fix?

Clean as you go

- Write comments where they are needed
- Name your variables based on what that variable is there to do
- In your block of code surrounded by {}:
    - Indent 4 spaces
    - Vertically align closing bracket with statement that opened it
- One expression per line
- Consistency in spacing
- Watch your code width (<= 80 characters)
- Watch the nesting of IFs - can it be done more efficiently?
- Break code into functions

Style Guide

- Often different organisations you work for, will have their own style guides, however, the basics remain the same across
- Your assignment will have style marks attached to it
- We have a style guide in 1511 that we encourage you to use to establish good coding practices early:
- https://cgi.cse.unsw.edu.au/~cs1511/24T1/resources/style_guide.html

Things are getting harder...

- If you do not understand something, do not panic!
- It is perfectly normal to not understand a concept the first time it is explained to you
    - ask questions in lectures
    - try and read over the information again
    - rewatch lectures
    - ask questions in the tutorial and the lab
    - ask questions on the forum
    - go to help sessions
    - go to revision sessions

Things are getting harder...

- If you can't solve a problem
  - break down the problem into smaller and smaller steps until there is something that you can do
  - ask us lots of questions!
- Remember learning is hard and takes time
- Solving problems is hard and takes practice
- We are here to help you!!!

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/2GGPJaSE37

What did we learn today?

- Recap of while loops, nested while loops
  - grid.c (clock.c left as an exercise)
- Recap of structs
  - struct_student.c struct_points.c
- Enums
  - enum_weekdays.c
- Enums and structs
  - pokemon.c

What did we learn today?

- Functions
  - simple_functions.c area_triangle.c print_pokemon.c memory_scope.c pass_by_value.c scanf_loop.c
- Style
  - bad_style.c

Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1511@unsw.edu.au