COMP1511/1911 Programming Fundamentals

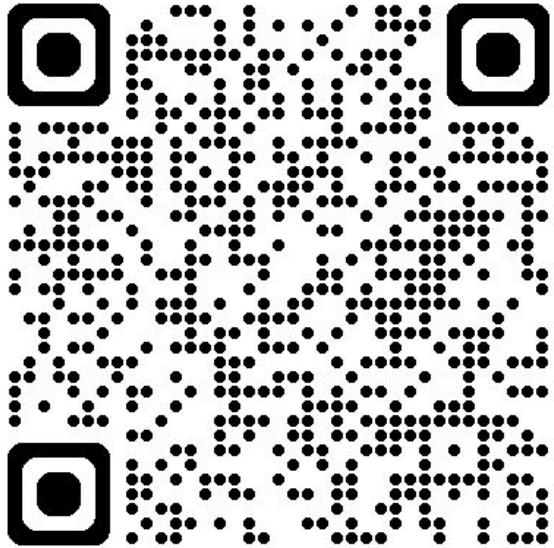
Week 2 Lecture 2

Loops
Custom Data Types

Revision Videos

https://cgi.cse.unsw.edu.au/~cs1511/current/resources/revision_videos.html

- We've created a series of short-form videos to help you quickly revise important topics from the course!
- These videos provide concise, easy-to-understand explanations of core concepts in the course, perfect for refreshing your memory!
- Let us know if there are any topics you'd like us to cover next!



Help Sessions

Starting next week!

Schedule out soon!

Yesterday's Lecture

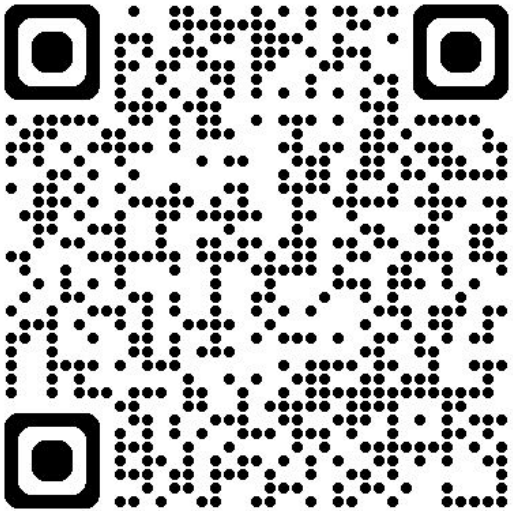
- Conditions and if statements
  - Relational Operators, Logical Operators
  - if-else, chaining if-else, nested if statements
- While loops
  - Infinite loops
  - Intro to counting loops

Today's Lecture

- More single while loops
- Nested While Loops
- Custom data types
  - structs
  - enums

Link to Week 2 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/25T3/code/week_2/



Recap scanf return

scanf_return.c

3 Ways of Controlling while loops

- counting loops
  - The number of iterations is known
  - Use a variable as a counter to control how many times a loop runs
- conditional loops
  - We may not know how many times we will need to loop
  - Conditions terminate the loop based on calculations or user input
- sentinel loops
  - Special case of conditional loops
  - A sentinel loop continues to execute until a special value (the sentinel value) is encountered.

Counting while loops

- Use a loop control variable (“loop counter”) to count loop repetitions.
  - We stop when the loop reaches a certain limit.
- Useful when we know how many iterations we want.

```
// 1. Initialise loop counter before the loop
int counter = 0;
while (counter < 5) { // 2. check loop counter condition
   printf("Here we go loop de loop!\n");
   counter = counter + 1; // 3. update loop counter
}
```

Conditional Loops

- Iterate as long as your condition is still true
- Used when we don't know how many times we need to loop

```
// 1. Initialise the loop control variable
int total_kombucha_ml = 0;
int kombucha_ml;
while (total_kombucha_ml < MAX_KOMBUCHA) { // 2. Test the loop condition
    printf("Please enter the ml of kombucha: ");
    scanf("%d", &kombucha_ml);
    // 3. Update loop control variable
    total_kombucha_ml = total_kombucha_ml + kombucha_ml;
}
printf("Warning! You have had %dml today!!\n", total_kombucha_ml);
```

Sentinel Loops

- Process data until reaching a special value (sentinel value)
  - Special case of conditional loop

```c
int number = 0;
int end_loop = 0;        // 1. Initialise the loop control variable
while (end_loop == 0) { // 2. Test the loop condition
    scanf("%d", &number);
    if (number < 0) {    // We want a negative value to end the loop
        end_loop = 1;    // 3. Update the loop control variable
    } else {
        printf("You entered %d\n", number);
    }
}
```

Code Demo

`while_count.c`

`while_condition.c`

`while_sentinel.c`

Write a program that reads integers from the user and sums them until a non-integer input is encountered

`while_scanf_sum.c`

Nested While Loops

- A loop in a loop
- If we put a loop inside a loop . . .
- Each time a loop runs
  - It runs the other loop
- The inside loop ends up running a LOT of times
  - How many times does the second hand go around the clock for every minute? For every hour?

Why are nested while loops useful?

How could we print out something like this?

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Or this?

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Code Demo Nested While Loop

```
grid.c
pyramid.c
clock.c (if we have time)
```

Quick Break. Back Soon…

Custom Data Types

Organising related data

Is there a better way of storing related data?

```
char my_first_initial = 'A';
char my_last_initial = 'F';
int my_age = 23;
double my_lab_mark = 2.4;
char brianna_first_initial = 'B';
char brianna_last_initial = 'K';
int brianna_age = 21;
double brianna_lab_mark = 9.9;
```

Organising related data

Is there a better way of storing related data?

```
char my_first_initial = 'A';
char my_last_initial = 'F';
int my_age = 23;
double my_lab_mark = 2.4;
```

```
char brianna_first_initial = 'B';
char brianna_last_initial = 'K';
int brianna_age = 21;
double brianna_lab_mark = 9.9;
```

We could group the data related to a person

Organising related data

Is there a better way of storing related data?

```
int x1 = 0;
int y1 = 0;
int z1 = 0;
int x2 = 10;
int y2 = -5;
int z2 = 5;
```

Organising related data

Is there a better way of storing related data?

```
int x1 = 0;
int y1 = 0;
int z1 = 0;
int x2 = 10;
int y2 = -5;
int z2 = 5;
```

We could group the data related to a coordinate

User defined Data Type: struct

- So far, we have used built-in C data types (int, char, double)
- These store a single item of that type
- `structs` allow us to define our own data types (structures) to store a collection of types
- Before we can create struct variables, we need to define the struct (outside the main)
  - Note this does not create a variable or set aside any memory.
  - It just defines the type.
- Then we declare and use struct variable/s

1. Defining a struct

- We define our structs before our main function.
- `structs` are types that we design, made up of data elements that we decide belong together
  - we call these elements **members** or **fields**
  - we need to define a type and name for each member

```
struct student {
    char first_initial;
    char last_initial;
    int age;
    double lab_mark;
};
```

2. Declaring a struct variable

- Creating variables using your custom **`struct`** type

```c
struct student {
    char first_initial;
    char last_initial;
    int age;
    double lab_mark;
};
```

```c
int main(void) {
    // Declare a variable
    // of type struct student
    struct student brianna;
```

3. Initialising struct data

- We access a member of a struct by using the dot operator `.`

```c
struct student {
    char first_initial;
    char last_initial;
    int age;
    double lab_mark;
};
```

```c
int main(void) {
    // Declare a variable
    // of type struct student
    struct student brianna;
    // Initialise the members of
    // your struct variable
    brianna.first_initial = 'B';
    brianna.last_initial = 'K';
    brianna.age = 21;
    brianna.lab_mark = 9.9;
```

Exercise: Using structs

- Increment the age field
- Read in updated lab mark from the user.
- Print out struct data

```c
struct student {
    char first_initial;
    char last_initial;
    int age;
    double lab_mark;
};
```

```c
int main(void) {
    // Declare a variable
    // of type struct student
    struct student brianna;
    // Initialise the members of
    // your struct variable
    brianna.first_initial = 'B';
    brianna.last_initial = 'K';
    brianna.age = 21;
    brianna.lab_mark = 9.9;
```

Exercise: Using structs

- Enter data for a point
- Print out the point struct

```c
struct coordinate {
    int x;
    int y;
    int z;
};
```

```c
int main(void) {
    // Declare 2 variables of
    // type struct coordinate
    struct coordinate point_1;
    struct coordinate point_2;
```

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

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/F56gV5WHM7

What did we learn today?

- While loops
  - `while_count.c, while_conditional.c, while_sentinel.c, while_scanf_sum.c`
- Nested while loops
  - `grid.c, pyramid.c, clock.c`
- structs
  - `struct_student.c, struct_points.c`
- enums
  - `enum_weekdays.c`

Next week

- Functions
- Style

Reach Out

Content Related Questions:
Forum

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