Week 2 Lecture 2
Custom Data Types

Last lecture

- [x] Control flow
- [x] conditions
- [x] if/else if/else
- [x] while loops
- [x] scans

Today

- [ ] Nested loops
- [ ] Custom data types

Live lecture code

https://cgi.cse.unsw.edu.au/~cs1511/24T2/live/week_2/

Link will be pasted in the Stream Chat

if statements recap

```
if(<condition>) {
    do_if_true();
} else if(<second_condition>) {
    do_if_second_true();
} else {
    do_if_both_false();
}
```

- A condition is a true/false value (1/0)
- We can execute an expression to calculate the condition
  - `my_age > drinking_age` -> will evaluate to true/1 if age is greater than `drinking_age`
- Conditions are useful in many places, if statements, while loops, etc.

While loops

```
while(<condition>) {
    do_something_over_and_over();
}
```

- if true, run the body
- at end of body, check condition again
- if true, run the body...

Nested loops

- Simply having a `while` loop within a `while` loop
- Each time the outer loop runs, the inner loop runs an entire set (the inner loop runs *a lot*)

Why are nested loops useful?

Why are nested loops useful?

How can we print 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
```

```c
#include <stdio.h>

#define ROWS 5
#define COLUMNS 5

int main() {
    int i = 0;

    while (i < ROWS) {
        int j = 1;
        while (j <= COLUMNS) {
            printf("%d ", j);
            j++;
        }
        printf("\n");
        i++;
    }

    return 0;
}
```

What about a half-pyramid?

```
1
12
123
1234
12345
```

Week 2 Lecture 2
Custom Data Types

Custom data types

- So far, we have used built-in C data types (int, char, double)
- These store a single item of that type
- What if we want to store a group of related data?

```c
int main(void) {
    int my_age = 20;
    char initial = 'J';
    int UNSW_year = 2;

    return 0;
}
```

^ These three things are related...

We can define our own data types (structures) to store a collection of types

Enter the `struct`

UNSW_student struct

```
struct UNSW_student {
    int age;
    int year_number;
    double WAM;
}
```

To use, we simply say:

```
struct UNSW_student Jake;
```

`struct` (structures)

- Are variables made up of other variable(s)
- They have a single identifier
- Can still access the sub-variables

Defining a struct

```
struct <struct_name> {
    data_type identifier;
    data_type identifier;
}
```

Example

```
struct UNSW_student {
    int age;
    int year_number;
    double WAM;
}
```

Defining a struct

```
struct <struct_name> {
    data_type identifier;
    data_type identifier;
}
```

Example

```
struct UNSW_student {
    int age;
    int year_number;
    double WAM;
}
```

^ Notice, no values... we are only defining.

Full program example

```c
#include <stdio.h>

struct UNSW_student {
    int age;
    int year_number;
    double WAM;
}

int main(void) {
    struct UNSW_student Jake;

    return 0;
}
```

But how do I access the actual data...

the `.` operator

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

```
struct coordinate loc;

loc.x
loc.y
```

DEMO

Another custom data type

The `enum`

Imagine I wanted to store days of the week

```
1. int day_of_week = 1;
2. char day_of_week = 'm';

3. #define MONDAY 1
4. #define TUESDAY 2
```

The problem

- Have to remember that 1 is Monday
- Could accidentally set 8 to `day_of_week`

Enums (the solution)

- Store a range or set of possible values
- Assigns a more meaningful name to state

Syntax

```
enum enum_name { state_1,
state_2, state_3... };
```

Example

```
enum weekdays { Mon, Tue, Wed,
Thu, Fri, Sat, Sun };
```

Using enums

```c
#include <stdio.h>

enum weekdays { Mon, Tue, Wed, Thu,
Fri, Sat, Sun };

int main(void) {
    enum weekdays day;
    day = Sat; // <-- this is why enums
are useful

    return 0;
}
```

Under the hood

```c
#include <stdio.h>

enum weekdays { Mon, Tue, Wed, Thu, Fri,
Sat, Sun }

int main(void) {
    enum weekdays day;
    day = Sat;
    printf("The actual value in day is:
%d\n, day);

    return 0;
}
```

Advantages over other approaches

- We provide limitations on the possible values (has to be defined in the enum)
- We give a nice label to values (Sat)
  - We don't have to remember that 1 is Monday *(or was it 0? 🤔)*
- Could use `#define` but these can clutter our code if we have many

`struct` 🤝 `enum`

```
enum student_status { Enrolled,
Withdrawn, Leave }

struct student {
    enum student_status status;
    int age;
}
```

Feedback