Access live lecture example code here as its written:
https://cgi.cse.unsw.edu.au/~cs1511/21T2/live/

Making Decisions

• Sometimes we want to make decisions based on what information we have at the time
• We can let our program branch between sets of instructions
  • also called control flow, or conditional execution
• In C this is the if statement
• First we ask a question
• If we get the right answer, we run some code

```c
if (expression) {
  // this runs if the expression results in anything other than 0 (true)
}
```

else - Adding to if statements

We can expand beyond the simple if by adding the else statement

```c
if (expression) {
  // this runs if the expression results in anything other than 0 (true)
} else {
  // this runs if the earlier expression results in 0 (false)
}
```

Let’s create a program using if

This program will help us in our games of “Catacombs and Large Reptiles”

A user will roll two dice and tell us the result of each die

Our program will add them together and check them against a target number that only the program knows.

It will then report back whether the total of the dice was higher, equal or lower than the secret number.
What does our program need?

- All recipes need ingredients
- A way to “talk” to our user
  - We know about `printf`
- A way to receive input
  - We learnt about `scanf` today
- A way to compare numbers . . .
  - Relational Operators
  - . . .Against a secret number
  - A variable or a constant
- A way to run different code depending on the number comparisons
  - If and else conditional statements

Describe our program

We’ll write some comments and include stdio.h for `printf` and `scanf`

```
// Alan Turing, September 1938
// This small example will ask the user to input the
// result of rolling two dice.
// It will then check the sum of them against its
// secret number.
// It will report back:
// success (higher or equal)
// or failure (lower)

#include <stdio.h>
```

Using a `#define` in our program

`#define` is nice for things that we know aren’t going to change. Note the use of all caps to signify a constant and underscores to show different words

This will go after our `#include`, but before our main

```
#define SECRET_TARGET 7
```

Add a main function

We always need a main function for C to know where our program starts

```
int main(void) {
    // our code goes here
    return 0;
}
```
Setting up some variables

We know we’re going to be getting dice values from our user, so we can start by declaring them

```c
// set up some dice variables so we can store numbers
int die_one;
int die_two;
```

Reading in the Dice Throws

Using printf and scanf, we can print words to the screen and read numbers

We store the two die rolls in the variables we set up earlier

```c
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then read in a number they type in the terminal
scanf("%d", &die_one);
// repeat for the second die
printf("Please enter your second die roll: ");
scanf("%d", &die_two);
```

Calculation

Using some basic arithmetic, we calculate our total

We then save that value in a “total” integer variable

```c
// calculate the total and report it
int total = die_one + die_two;
printf("Your total roll is: %d\n", total);
```

If Statement

• We now use an if statement to test for success
  • `>=` is a comparison operator
    • its value is 1 if the comparison is true
    • its value is 0 otherwise

```c
// Now test against the secret number
if (total >= SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
}
```
What about the failure?

The other option, which we actually don’t have to test for, because it’s all that’s left.

```c
// Now test against the secret number
if (total >= SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
} else {
    // the same as total < SECRET_TARGET
    // but we don’t have to test it because
    // we’ve already checked all other
    // possibilities
    printf("Skill roll failed!");
}
```

We have a dice check program!

```c
#include <stdio.h>
#define SECRET_TARGET 7
int main(void) {
    int die_one;
    int die_two;
    // we start by asking the user for their dice rolls
    printf("Please enter your first die roll: ");
    // then read in a number they type in the terminal
    scanf("%d", &die_one);
    // repeat for the second die
    printf("Please enter your second die roll: ");
    scanf("%d", &die_two);
    // calculate the total and report it
    int total = die_one + die_two;
    printf("Your total roll is: \200\201%d\201\n ", total);
    // Now test against the secret number
    if (total >= SECRET_TARGET) {
        // success
        printf("Skill roll succeeded!");
    } else {
        // the same as total < SECRET_TARGET
        // but we don’t have to test it because
        // we’ve already checked all other
        // possibilities
        printf("Skill roll failed!");
    }
    return 0;
}
```

source code for dice_check.c

Challenges

Challenges:
Can you modify this code to:
Detect exact ties as well as success and failure?

What about the idea of a “critical double”? Can you detect when the player rolled the same number on both dice and report it as a “critical” success, tie or failure?

What about randomisation? (This is much harder and might need some things from later in the term)

More Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
</tbody>
</table>

- all the operators have the value 1 if the comparison is true, and 0 otherwise
- Important we use = to change a variable
- Important we use == to check if two value are equal
Chaining Questions Together

- C has logical operators: `&&` `||` `!`
- Logical operators allow us to combine comparisons, eg:
  - `mark > 0 && mark < 100`
- `&&` is the **and** operator
  - 1 if both operands are true, 0 otherwise
  - `2 > 0 && 2 < 9` has value 1
  - `2 > 0 && 2 < 1` has value 0
- `||` is the **or** operator - true if either operand is true
  - `24 > 42 || 2 < 9` has value 1
  - `24 > 42 || 2 < 1` has value 0
- `!` is the **not** operator - true iff its operands is false
  - 1 if its operand is false, 0 otherwise
  - `!(24 > 42)` has value 1
  - `!(24 > 12)` has value 0

A process for problem solving

- **We can develop a way to approach all problems**
- Figure out what’s wrong (or what we need to solve)
- Find out what our options are (what code could we write or change?)
- Assess those options
  - How well do they solve the problem?
  - Can we make them work?
- Pick an option to try
- Did it work?
  - If it didn’t or even if it did, we can get more information for our next attempt

Problems and Solutions

Figure 1:

What’s our problem?
This is always a good question!
Spending some time figuring out exactly what we aim to do (or what’s stopping us from getting there) is important
Keeping the problem in mind keeps you focused on a solution

Back to the Dice Checker

We created a program that:
- Asked the user to input their dice values
- Reported back whether the total was above or below a target value

Let’s look at one problem that might occur
- What if the user enters incorrect values?
  - Too high or too low?
Testing our Input

Let’s assume we have this input code:

```c
// Setup dice variables
int die_one;
int die_two;
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &die_one);
```

A six sided die has a specific range of inputs

We will only accept inputs in this range

![Six-Sided Dice](image)

But ints have a much wider range!

Testing Input Range in Code

```c
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &die_one);

// test for proper input range
if (1 <= die_one && die_one <= 6) {
  // if this succeeds, we are in the right range
} else {
  // number was outside the die’s range
}
```

Using Constants in code

The 1 and 6 are the minimum and maximum values of the dice

We could use something like this at the start of our program:

```c
#define MIN_VALUE 1
#define MAX_VALUE 6
```

- Makes the program much easier to modify for different dice sizes
- Also makes things much more readable by using explanations instead of numbers

Dice

- The Egyptians were using flat sticks to randomise movement in Senet that dates games with randomisation back past 3000BC
- Six sided dice have been excavated in Iran from 2800-2500BC Nowadays we usually use dice ranging from 4 to 20 sides
- Random Number Generation (for letting the computer roll the dice) uses libraries which we’ll show later in the course!

What are our options?

If we know we have incorrect input, what do we do?

We have several options . . .

- PANIC!!!!!
- Reject the input, end the program
- Let the user know what the correct input is
- Correct the input
  - Ask for new input
Reject the input

We can just end the program if the input is incorrect

```c
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &die_one);
// test for proper input range
if (MIN_VALUE <= die_one && die_one <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die’s range
    return 1;
}
```

- `return` causes a function execution to stop
- much about this in 2 weeks

Assessing This Option

Is it a good idea to have the program just end?
- What’s a good way for the program to reject incorrect input?
- If we’re testing or using the program, what do we want to see?

25

Reporting Failure

Information from the program helps the user

```c
// test for proper input range
if (MIN_VALUE <= die_one && die_one <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die’s range
    printf("Input for first die, %d was out of range. Program will exit now. 
", die_one);
    return 1;
}
```

Can we do better?

Exploring other options

Let’s give the user information that helps them correct the input issues

```c
// test for proper input range
if (MIN_VALUE <= die_one && die_one <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die’s range
    printf("Input for first die, %d", die_one);
    printf(" was out of the range %d - %d", MIN_VALUE, MAX_VALUE);
    printf(" Program will exit now.
");
    return 1;
}
```
If we want the program to finish executing even with bad input

- Imperfect, but sometimes we want the program to finish

What are our options?

- **Clamping** - anything outside the range gets “pushed” back into the range
- **Modulus** - a possibly elegant solution

Any Issues with Clamping?

- Definitely end up with input that works
- But is it correct?
- What are the issues with correcting data without the user knowing?

A reminder of what it is

% - A maths operator that gives us the remainder of a division

How can we use it?

Any number “mod” 6 will give us a value from 0 to 5

If we use a 6 instead of the 0, we get the range 1 to 6

This means the user could type in completely random numbers and be given a 1-6 dice roll result
Using Modulus in code

```c
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &die_one);
// mod gives us a result within 0-MAX_VALUE
die_one = die_one % MAX_VALUE;
// make any value < MIN_VALUE into MAX_VALUE
if (die_one < MIN_VALUE) {
    die_one = MAX_VALUE;
}
```

Pros and Cons of using Modulus for dice

**Pros**
- We guarantee a number between 1 and 6 (or whatever the max value is)
- We don’t shut down unexpectedly due to incorrect input
- We give a very dice-like randomish result (as opposed to clamping)

**Cons**
- We might accept incorrect input silently
- We might make a change that affects the user’s expectations

A Range of Solutions

Which one to use?
- No single answer
- The original purpose of the program can help us decide
- What’s our priority?
- Exact correctness?
- Failure on any kind of incorrect data?
- Usability and randomisation over correctness?

The Upgraded Dice Checker

- The programmer can set the size of the dice in `#define` constants
- The user can enter any number and it will produce a valid roll
- The program will still report back success or failure
- Starting from our previous Dice Checker program, we can make some modifications to give it some new capabilities
Our Solution . . . description of the program

// The Dice Checker v2
// Alan Turing, July 1938
// Allows the user to set dice size
// Tests the rolls of two dice against a target number
// Able to deal with user reported rolls outside the range
// Will report back Success, Tie or Failure

Our Solution . . . constants

#include <stdio.h>
#define MIN_VALUE 1
#define MAX_VALUE 6

// The secret target number
#define SECRET_TARGET 7

Our Solution . . . main

• main starts the same as before

int main(void) {
    int die_one;
    int die_two;

    Our Solution . . . input of dice rolls

    Two rolls will be taken as input (only one is shown here)
    
    // Process the first die roll
    printf("Please enter your first die roll: ");
    scanf("%d", &die_one);

    // Check and fix the die roll
    if (die_one < MIN_VALUE || die_one > MAX_VALUE) {
        // die_one is invalid
        printf("%d is not a valid roll for a D%d.\n", die_one, MAX_VALUE);
        die_one = die_one % MAX_VALUE;
        // make any value < MIN_VALUE into MAX_VALUE
        if (die_one < MIN_VALUE) {
            die_one = MAX_VALUE;
        }
    }
}
Calculate and report the total

This is identical to last week's code

```c
// calculate the total and report it
int total = die_one + die_two;
printf("Your total roll is: %d\n", total);
// Now test against the secret number
if (total > SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
} else {
    // failure
    printf("Skill roll failed!\n");
}
```

We've added:
- Some measures against user mistakes
- Some modifiability

We made some decisions:
- We will report any user errors
- But we're also delivering a die roll regardless

source code for dice_check_v2.c

if example - avoiding division by zero

```c
int x, y;
printf("Enter x: ");
scanf("%d", &x);
printf("Enter y: ");
scanf("%d", &y);
if (y != 0) {
    printf("%d/%d=%d\n", x, y, x/y);
} else {
    printf("Can't divide by zero sorry\n");
}
```

source code for divide.c

if example - absolute value

```c
int x, absoluteValue;
printf("Enter number: ");
scanf("%d", &x);
absoluteValue = x;
if (x < 0) {
    absoluteValue = -1 * x;
}
printf("The absolute value of %d is %d\n", x, absoluteValue);
```

source code for absolute.c
```c
double mass, rest_mass;
double velocity;
double ratio;
printf("Enter rest mass: ");
scanf("%lf", &rest_mass);
printf("Enter velocity in metres/second: ");
scanf("%lf", &velocity);
// compute velocity as a fraction of speed of light
ratio = velocity / SPEED_OF_LIGHT;
if (ratio >= 1.0) {
    printf("Error: velocity exceeds speed of light.\n");
} else {
    // compute observed mass using Einstein's equation
    mass = rest_mass / sqrt(1.0 - ratio*ratio);
    printf("Observed mass = %1.6f\n", mass);
}
```

### Chained if example #1

```c
int a;
int b;
printf("Enter a: ");
scanf("%d", &a);
printf("Enter b: ");
scanf("%d", &b);
if (a > b) {
    printf("a is greater than b\n");
} else if (a < b) {
    printf("a is less than b\n");
} else {
    printf("a is equal to b\n");
}
```

### Chained if example #2 - printing more information

```c
int a;
int b;
printf("Enter a: ");
scanf("%d", &a);
printf("Enter b: ");
scanf("%d", &b);
if (a > b) {
    printf("%d is greater than %d\n", a, b);
} else if (a < b) {
    printf("%d is less than %d\n", a, b);
} else {
    printf("%d is equal to %d\n", a, b);
}
```
chained if example #2 - what to wear

```c
int temperature_in_celsius;
printf("Enter temperature in celsius: ");
scanf("%d", &temperature_in_celsius);
if (temperature_in_celsius < 0) {
  printf("Its freezing - wear a down jacket\n");
} else if (temperature_in_celsius < 20) {
  printf("Its cool - wear a hoodie\n");
} else {
  printf("Its warm - wear a t-shirt\n");
}
```

source code for what_to_wear.c

logical operators example - how many digits does a number have

```c
int x;
printf("Enter x: ");
scanf("%d", &x);
printf("%d has ", x);
if (x < 10 && x > -10) {
  printf("1 digit");
} else if ((x >= 10 && x < 100) || (x <= -10 && x > -100)) {
  printf("2 digits");
} else if (x >= 100 || x <= -100) {
  printf("more than 2 digits");
}
printf("\n");
```

source code for digits.c

nesting if statements example

```c
int a;
printf("Enter a: ");
scanf("%d", &a);
printf("%d is a ", a);
if (a < 0) {
  if (a < -100) {
    printf("big");
  } else {
    printf("small");
  }
  printf(" negative");
} else {
  if (a > 100) {
    printf("big");
  } else {
    printf("small");
  }
  printf(" positive");
}
printf(" number.\n");
```

source code for nested_if.c

Logical Operators - Conditional evaluation - Advanced Topic

- `&&` || always evaluate their left-hand side first.
- They only evaluate their right-hand side if needed.
- `&&` will not evaluate its right-hand side if left-hand side is false
- `||` will not evaluate right-hand side if left-hand side is true
- useful if evaluating the righthand operand first could produce an error, e.g.;
- allows us to write code like this

```c
if (x != 0 && y/x > 2) {
```

without risking division by zero.
• Advanced Topic: remember doubles are approximations.
  • comparing doubles for equality using == or != is dangerous
  • much safer to check to check if they are close to same value

```c
double theta, identity;
printf("Enter theta: ");
scanf("%lf", &theta);
identity = 1 - (sin(theta) * sin(theta) + cos(theta) * cos(theta));
if (identity == 0.0) {
    printf("Pythagorean identity true for %lf\n", theta);
} else {
    printf("Pythagorean wrong by %g for %lf\n", identity, theta);
}
```

source code for pythagorean_identity.c

53