

CS 137 Week 1

Introduction to C Programming

September 8th, 2017

Introducing Team

- Dr. Carmen Bruni (more on me later)
- Chantelle Gellert (Our ISC)
- Alexander Rowaan (Our ISA)

About Me

Carmen Bruni, DC 3142 (later 3119), cbruni@uwaterloo.ca

Websites:

- www.student.cs.uwaterloo.ca/~cs137
- <https://piazza.com/>

Office Hours: Mondays 1:00-1:55 and Tuesdays 8:30-10:30 in my office.

Office hours for Alex: 4:00-5:30 Tuesdays and Thursdays in MC 4065.

Introduction

On your index card, write:

- Your actual name
- Your preferred name (and tips for pronunciation!)
- Something interesting about you
- Someone famous that shares the same name (first or last) or birthday as you.

Words about the Course

Provides an introduction to fundamental programming principles for first-year Software Engineering students. Topics include: procedures and parameter passing, arrays and structures, recursion, sorting, pointers and simple dynamic structures, space and time analysis of designs, and design methodologies. The course will be taught using the C programming language.

Course Times

- Section 001, 3:30-4:20M, 1:30-2:30 WF, MC 1085 and the following Fridays: September 22nd, October 6th, October 20th, November 3rd, November 17th 3:30-4:30 MC 1085
- Lab: Tuesday September 12th at 8:30-9:20 or 9:30-10:20 MC 2062
- Tutorials: Tuesdays (From September 19th on) at 8:30-9:20 or 9:30-10:20 MC 4041

Grading Scheme

- 8% Assignments (10 total - Must do A0 but worth 0%, take best 8 of remaining 9)
- 35% Midterm on Friday October 20th from 9:30-11:30.
- Max of (57%,0%) (50%,7%) for final exam and clickers

Marmoset

- Assignments will be submitted on Marmoset (online submission tool)
- At least 50% of your grade on every assignment will be public tests.
- More on this in Lab 1.

Clickers

- A great way to encourage active learning.
- Note: Not all weeks will have the same number of clicker questions! Some weeks will have substantially less and some weeks will have substantially more depending on the type of content.



Outline

In this course, we will use Marmoset, Piazza and a public website.

- All content related to the course lectures will be on the public website stated above.
- All content relating to grading (assignments, tests, etc.) will be on the website (password protected).
- All non-personal questions should be posted on Piazza.
- All homework submissions will be done on Marmoset.
- Textbook (Optional): K.N. King, C Programming: A Modern Approach

Tips for Surviving 1A

- Eat well, sleep right and exercise. Surviving is harder when you are unhealthy!
- Fix holes in your knowledge today. You will never have a better opportunity to correct mistakes and overcome fears than today.
- Start early! (Assignments, studying, etc.) Don't fall behind!
- Go to class!
- Brace yourself for failure (not in a course necessarily, but expect problems you will have difficulty solving).
- Swimming analogy
- Remember since you have made it this far, it means that you are fully capable and can succeed here.
- Find help (More on next slide).

Options for Help

- Your ISA's office hours.
- My office hours.
- Friends! (Discuss cheating and how to avoid it.)
- Piazza, our online forum www.piazza.com
- Math Tutorial Centre
<https://uwaterloo.ca/math/current-undergraduates/mathematics-tutorial-centre> (MC 4066 8:30-5:30 starting either Next Monday or the one after)
- The course webpage <https://www.student.cs.uwaterloo.ca/~cs137/index.shtml>.

Course Outline

From the textbook:

1. Basic C Programming, including variables, integers, characters, expressions, conditionals, loops
2. Functions, parameters, arguments, recursion
3. Arrays and pointers
4. Structures

Not in the textbook:

1. Sorting, searching, time and space complexity
2. Other fun stuff

Core CS Sequence

- 1A - CS 137 - Programming Principles (in C)
- 1B - CS 138 - Data Abstraction (in C++)
- 2A - CS 241 - Sequential Programs (in C++/Scheme)
- 2B - CS 247 - Abstraction and Specification (in C++)

and SE 101, ECE 124, SE 212, ECE 222, CS 240, etc.

First C Program

```
#include <stdio.h>
int main(void) {
    printf("Hello world\n");
    return 0;
}
```

Note - The f in printf stands for “formatted”.

First C Program Explanation

```
//This is a standard i/o header file  
#include <stdio.h>  
//int below is the return type  
//void below means an empty parameter list  
int main(void) {  
    // the \n below is a newline character  
    printf("Hello world\n");  
    //return status to caller  
     //(usually the shell on an OS)  
    return 0;  
}
```


Returning Success

By convention, return 0 is a success and anything non-zero is a failure. There are alternatives to the above code:

```
#include <stdlib.h>
int main(void) {
    return EXIT_SUCCESS; //which is 0
}
//or return EXIT_FAILURE; which is 1
```

You can also omit the return statement (which in C99 defaults to returning 0 but in C89, is undefined).

How to Run Code

```
% nano hello.c  
% gcc hello.c  
% ./a.out  
% gcc-o hello hello.c  
% ./hello
```

With Explanation

```
#Create hello.c  
% nano hello.c  
#Compile into executable (a.out)  
% gcc hello.c  
# ./ refers to the current directory;  
# after this is the executable name  
% ./a.out  
# This sets the output file  
# to the first parameter  
% gcc-o hello hello.c  
% ./hello
```

IDE Options

There are some options for an IDE if you like GUI based learning. I've heard good things about Eclipse (and CodeBlocks but I've had bug problems with CodeBlocks). You can also just go online and use `repl.it` which works great for single file programs. You are free to use whatever editor you would like.

Homework For Day 1

- If you have the textbook, read chapters 1,2, 4 and 5 (and skim through chapter 3)
- Change your CS account password and do A0
- Familiarize yourself with UNIX and some text editor (or get Eclipse running!)
- Linux and MacOS users should learn the basics of SSH (or Fugu for a GUI option)
- Windows users should learn putty, or bash (Cygwin) or SSH terminal client for a GUI option.
- Get a Watcard!!!

Our First Algorithm - The Euclidean Algorithm

- How do we reduce $\frac{1080}{1920}$?

Our First Algorithm - The Euclidean Algorithm

- How do we reduce $\frac{1080}{1920}$?
- Similarly, how to we compute the greatest common divisor of 1080 and 1920, denoted by $\text{gcd}(1080, 1920)$?

Our First Algorithm - The Euclidean Algorithm

- How do we reduce $\frac{1080}{1920}$?
- Similarly, how to we compute the greatest common divisor of 1080 and 1920, denoted by $\gcd(1080, 1920)$?
- Turns out geometrically this has a neat interpretation.

Youtube Video

<https://www.youtube.com/watch?v=AVrtH6m2wcU>

How Does It Work?

- What the video shows is that at each step, we are dividing the larger number by the smaller number using the division algorithm (more on this in MATH 135).
- When Euclid performed this, he used repeated subtraction (which is basically division).
- In fact, we have

$$1920 = 1080(1) + 840$$

$$1080 = 840(1) + 240$$

$$840 = 240(3) + 120$$

$$240 = 120(2) + 0$$

Mod operator

- At each step, the remainder when dividing a number by another positive integer seems to be the most important value
- In C, we have the modulus operator (sometimes called the remainder operator) given by `%` and defined over the integers with $a \geq 0$ by

$a \% b$ is the remainder when a is divided by $|b|$ (nonzero)

- For example, $18 \% 4 = 2$ and $23 \% (-5) = 3$. Also, $1920 \% 1080 = 840$.
- For $a < 0$, the above is the remainder subtracted by b (so that the sign of the answer is the same as the sign of a).

Repeating Structure

- Notice that at each step, we want to basically do the same procedure, that is, divide a value by another value and keep track of the remainder.
- In C, this can be accomplished using a while loop:

```
while(expr_is_true){  
    //do something  
}
```

- Let's see an example:

While Loop Example

The following code counts down from 5 down to 0.

```
#include <stdio.h>
int main(void) {
    int i = 5;
    while(i != 0){
        i = i - 1;
    }
    return 0;
}
```

Notice on each pass the value of *i* gets smaller and smaller until it reaches 0 when the loop expression is false and we break the loop.

Rules for Variable Names

- The command `int i = 5;` initializes an integer variable (think that this code creates a box called `i` and store the value 5 in this box).
- Variable names...
 - Must begin with a letter or an underscore.
 - After the first letter, can be letters, numbers or underscores.
 - Case sensitive.
 - Cannot be keywords (eg. `int`, `while` etc.)

Another Problem

What about printing? Specifically what about printing variables?

```
printf(string, argument(s));
```

To print variables, in our string we can include "%d" meaning that the first instance will correspond to the first argument. As an example,

```
#include <stdio.h>
int main(void){
    int a = 3;
    printf("1 + 2 = %d and %d + %d = 9 \n",
           a, a, 2*a);
    return 0;
}
```

prints **1 + 2 = 3 and 3 + 3 = 9** to the screen. The %d literally means "signed integer". There are others like %c for character, %u for unsigned integer and so on. We will introduce as needed.

Revisit While Loop

The following code counts down from 5 down to 0.

```
#include <stdio.h>
int main(void) {
    int i = 5;
    while(i >= 0){
        printf("The value of i is %d", i);
        i = i - 1;
    }
    return 0;
}
```

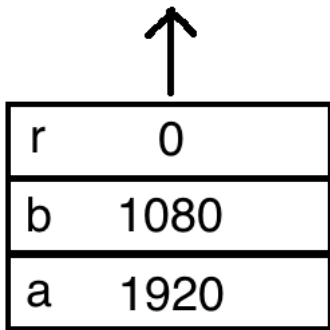
Notice on each pass the value of *i* gets smaller and smaller until it reaches 0 when the loop expression is false and we break the loop.

Let's Code the Euclidean Algorithm!

Let's Code the Euclidean Algorithm!

```
#include <stdio.h>
int main(void) {
    int a = 1920;
    int b = 1080;
    int r = 0;
    while (b != 0){
        r = a%b;
        a = b;
        b = r;
    }
    printf("%d\n",a);
    return 0;
}
// Note: Could also
// use while(b){ }
```

Call stack



r	0
b	1080
a	1920

Code Running

Iterations of the while loop:

Iteration	a	b	r
0	1920	1080	0
1	1080	840	840 (=1920%1080)
2	840	240	240 (=1080%840)
3	240	120	120 (=840%240)
4	120	0	0 (=240%120)

Making the Code User Friendly

- Currently, we can only do the above example with $a = 1920$ and $b = 1080$.
- We could go into the code to change these numbers but this seems inefficient.
- We would like to make this more user friendly by asking the user for input for the a and b values after the file compiles.

Input

```
scanf ("%d", &a);
```

- The command `scanf` looks to the standard input (`stdin`) to read an integer.
- The integer read will be stored into `a`.
- If the user doesn't enter an integer, the previous value for `a` remains and the `scanf` fails.
- The `&` refers to the memory address of `a`. We'll talk a lot more about this later so for now accept that you need the `&` symbol to make the `scanf` command to work.
- `scanf` ignores all whitespace/newline characters. (Caveat see next page)

Other Features

```
scanf ("%d/%d" , &num ,&denom ) ;
```

- The above code will look for user input of fractions of the form *num/denom* and store the numerator and denominator in the appropriate variables.
- This idea will work not just for / but for any special type of inputted format.
- The inputted format however must match exactly in order to work.
- This includes white spaces in the matching.

Euclidean Algorithm With User Input

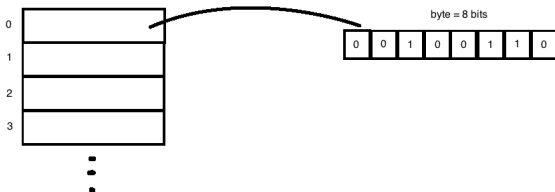
```
#include <stdio.h>
int main(void) {
    int a = 0;
    int b = 0;
    int r = 0;
    scanf("%d", &a);
    scanf("%d", &b);
    while (b != 0){
        r = a%b;
        a = b;
        b = r;
    }
    printf("%d\n", a);
    return 0;
}
```

Euclidean Algorithm With User Input

```
#include <stdio.h>
int main(void) {
    int a = 0, b = 0, r = 0;
    scanf("%d%d", &a,&b);
    while (b != 0){
        r = a%b; a = b; b = r;
    }
    printf("%d\n",a);
    return 0;
}
```


Computer Memory

Table of bytes (byte addressable)



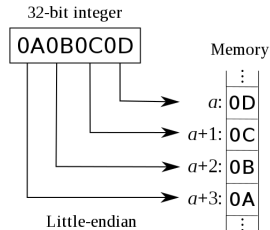
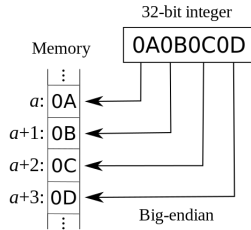
In the above example, $(00100110)_2 = 2^5 + 2^2 + 2^1 = 38_{10}$.
(Usually we drop the subscript if the base is clear).

Recall **1 byte = 8 bits**.

Endianness (Images Courtesy Wikipedia)

Integers are usually 4 bytes. There are two ways that an integer could be stored. Either they could be **big endian** (like in MIPS) where the most significant byte is at the lowest memory address [usual left-right reading; store from least to largest].

or it could be **little endian** (like in x86), namely the most significant byte is at the highest memory address. In C, the implementation is computer/compiler dependent and usually irrelevant for us.



Endianness

```
short int a = 1;
```

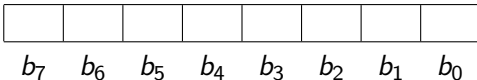
Memory Address	Big Endian	Little Endian
1000	0000 0000	0000 0001
1001	0000 0001	0000 0000

Table of Variable Values

Type	Storage Size	Value Range	Numeric
char	1 byte	$[-128, 127]$	$[-2^7, 2^7 - 1]$
unsigned char	1 byte	$[0, 255]$	$[0, 2^8 - 1]$
int	2 bytes	$[-32768, 32767]$	$[-2^{15}, 2^{15} - 1]$
int	4 bytes	$[-2147483648, 2147483647]$	$[-2^{31}, 2^{31} - 1]$
unsigned int	4 bytes	$[0, 4294967295]$	$[0, 2^{32} - 1]$
short int	2 bytes	$[-32768, 32767]$	$[-2^{15}, 2^{15} - 1]$
unsigned short int	2 bytes	$[0, 65535]$	$[0, 2^{16} - 1]$
long int	4 bytes	$[-2147483648, 2147483647]$	$[-2^{31}, 2^{31} - 1]$
long int	8 bytes	$[9.22 \cdot 10^{18}, 9.22 \cdot 10^{18} - 1]$	$[-2^{63}, 2^{63} - 1]$
long long int	8 bytes	$[9.22 \cdot 10^{18}, 9.22 \cdot 10^{18} - 1]$	$[-2^{63}, 2^{63} - 1]$
unsigned long long int	8 bytes	$[0, 1.84 \cdot 10^{19} - 1]$	$[0, 2^{64} - 1]$

Unsigned Integers

This is a positional number system that works like a normal binary system.



The value of a number stored in this system is the binary sum, that is

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0$$

For example,

$$01010101 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85_{10}$$

or

$$\begin{aligned} 11111111 &= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 255_{10} \end{aligned}$$

Converting to Binary

- Question: Write 38 and in binary.
- One way: Take the largest power of 2 less than 38, subtract and repeat.
- For example, 32 is the largest power of two less than 38, subtracting gives 6. Next, 4 is the largest power of two less than 6 and subtracting gives 2. This is a power of two hence $38 = 32 + 4 + 2 = (100110)_2$.
- Another way is to constantly divide by 2:

Number	Quotient	Remainder
38	19	0
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1

- ..and in binary (reading bottom to top) this is $(100110)_2$.

Signed Integers

- These are more complicated - they are represented in something called **Two's complement form**
- To negate a value:
 1. Take the complement of all bits
 2. Add 1
- This will ultimately mean that the first bit is a sign bit (0 if positive 1 if negative)

An Example

Let's compute -38_{10} using this notation. First, write 38 in binary:

$$38_{10} = 00100110$$

Next, take the complement of all the bits

$$11011001$$

Finally, add 1:

$$11011010$$

This last value is -38_{10} .

Short Cut

- A slightly faster way is to locate the rightmost 1 bit and flip all the bits to the left of it.
- For example:

11011010

Negating

00100110

How Signed Integers Work

- The idea is that an integer k is represented in memory as $2^n - k$ where n is the size of the data type.
- Arithmetic works naturally except that any final carry overs are ignored (see the two examples below).
- Watch out for overflow errors!
- For a few examples, to add 4 and -3 on the left in a 4 bit system or adding -4 and -3 on the right, we have

$$\begin{array}{rcl} & 1 & \\ 0100 & (+4) & \\ + \underline{1101} & (-3) & \\ \hline 0001 & (+1) & \end{array}$$

$$\begin{array}{rcl} & 1 & \\ 1100 & (-4) & \\ + \underline{1101} & (-3) & \\ \hline 1001 & (-7) & \end{array}$$

Overflow

- <http://www.youtubecutter.com/watch/26be1342/>
- <https://youtu.be/jm4wqRj4Lm8?t=3h17m12s>

What happened?

- The boss has 20000 health points (HP)
- SNES was a 16 bit console so this value was stored essentially in a `short int`.
- Thus, the boss' health is **010011100 0100000** in binary
- The Elixir is a full heal item which adds the original maximum health to the current health then checks if the health is greater than max and adjusts current health to max health.
- Thus, the final bosses HP becomes **10011100 01000000** which as a signed integer is **-25536**.
- This is less than 0 hence the boss is dead.

Basic Operations

- In the next few slides, we'll introduce some of the basic operations (you've seen many already!)
- Arithmetic Operators (follow BEDMAS rules):
(), %, *, /, +, - in order from left to right if there is a tie (for example, with modulus, division and multiplication or with the last two operations).
- Note that the division above for integers is integer division (the decimal is truncated).
- These are left associative.
- Note: There is no exponentiation operator in C (need to use repeated multiplication or a special library)

Assignment Operators

- Basically done last in order of operations.
- Right associative, that is, evaluate everything to the right first then assign the value.
- Syntax: `value = expr`, for example `a = 3`.
- There are also as assignment and operator operations
- Syntax: `value operation= expr` where the operation can be `%`, `*`, `/`, `+`, `-` or others which we will see later.
- For example, `a += 2`; 'roughly' means `a = a + 2`. I say 'roughly' because of weird situations for example `a *= b + c` which is not `a = a*b +c` (missing brackets) or if `a` is an object with some other side effect these won't be exactly the same (but for us this is unlikely to come up).
- Note: Order is important! The commands `a += 2`; and `a =+ 2`; are different!

Incremental Operators

- Two types: prefix, applied before the variable is used, and suffix, applied after the variable is used.
- As examples,
 - ++a; would increment a by one and then possibly use a.
 - --a; would decrement a by one and then possibly use a.
 - a++; would use a first (if applicable) then increment a by one.
 - a--; would use a first (if applicable) then decrement a by one.
- Combining:

```
#include <stdio.h>
int main(void) {
    int a = 3, b = 1;
    b += a+++2;
    return 0;
}
```

The value of a is 4 and the value of b is 6 (remember we add a and 2 first then add this to the value of b!) Notice that this can be confusing without spaces or brackets!

Incremental Operator Example

```
#include <stdio.h>
int main(void) {
    int a=10, b=15;
    a += ++b;
    printf("%d\n",a);
    a=10, b=15;
    a += b++;
    printf("%d\n",a);
    a=10, b=15;
    a += --b;
    printf("%d\n",a);
}
```

This will print 26, 25 and 24 on three separate lines.

An Example of Associativity

- Left associative: $i\%j*k$ is equivalent to $(i\%j)*k$
- Right associative: $a = b += c$ is equivalent to $a = (b += c)$
(which adds c to b and then stores a with the value of b)

```
#include <stdio.h>
int main(void) {
    int a = 1, b = 2, c = 3;
    a = b += c;
    printf("%d", a);
}
```

will print out 5.

Relational and Logical Operators

- Relational operators $<$, $>$, $<=$, $>=$ take lower precedence than arithmetic and are left associative
- Equality operators $==$, $!=$ (note the double equal sign!) take lower precedence than relational operators and are left associative.
- Logical Operators
 - $!$ (negation) is right associative and has the same precedence as unary $+$ or $-$.
 - $||$ and $\&\&$ are logical bitwise **or** and **and** respectively, are left associative and are lower than equality.

Relational and Equality Operators

- Logical operators return **1** for true and **0** for false.
- The operators `&&` and `||` are short-circuited evaluators; that is, they only evaluate right hand expressions as necessary. For example:

```
int a=0; a != 0 && 4/0 > 2;
```

Since *a* is 0, the first condition after the semi colon is false and hence C will not evaluate the second condition (that is, this will compile and run without error even though the second condition would normally give an error if on its own!)

Precedence Chart

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	= = !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13 ^(note 1)	?:	Ternary conditional ^(note 2)	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

De Morgan's Theorem

De Morgan's Theorem in C

Let P and Q be logical statements. Then

$$\!(P \&\& Q) == \!P \|\! \!Q$$

$$\!(P \|\! \!Q) == \!P \&\&\!Q$$

Proof:

P	Q	P && Q	!(P && Q)	!P	!Q	!P !Q
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Bit-Wise Operators

Suppose we have `char a=5, b=3;`. Note that this is valid as C will interpret 5 as the character corresponding to 00000101 and similar for 3 with the bit string 00000011 (recall also that a char is one byte).

- Bitwise not \sim , for example `c = \sim a;` gives `c = 11111010`
- Bitwise and $\&$, for example `c = a&b;` gives `c = 00000001`
- Bitwise or $|$, for example `c = a|b;` gives `c = 00000111`
- Bitwise exclusive or \wedge , for example `c = a ^ b;` gives `c = 00000110`
- Bitwise shift right or left $>>$ and $<<$, for example `c = a >> 2;` gives `c = 00000001` and `c = a << 3;` gives `c = 00101000`.
- These can even be combined with the assignment operator!

Selection Statements

- What do we do if we want to execute code only when some condition is true?
- To handle this, we can use one of a few different types of control statements.

If Statements

```
if (expression) {  
    //statement  
}  
else //optional  
{  
    //statement when expression is false  
}
```

Braces are optional if there is only one statement (brackets are not optional!) Can also write this on one line:

```
if (expression) statement else statement
```


First If Example

```
#include <stdio.h>
int main(void) {
    int a=5, b;
    if (a == 5){
        b = 10;
    } else {
        b = 4;
    }
    printf("%d\n",b);
    return 0;
}
```

Cascaded If Statements

```
if (expression){  
    //statement  
} else if (expression2) {  
    //statement  
} else if (expression3) {  
    //statement  
} //...  
} else {  
    //final statement  
}
```

If Example

```
#include <stdio.h>
int main(void) {
    int age;
    scanf("%d",&age);
    if (age < 5)
        printf("Free movie ticket!");
    else if (age < 18)
        printf("Youth movie ticket.");
    else
        printf("Regular admission movie ticket.");
    return 0;
}
```

Nested If Statements

Conditional statements can also be nested.

```
#include <stdio.h>
int main(void) {
    int x;
    scanf("%d", &x);
    if (x < 10){
        if (x >5)
            printf("Small");
        else
            printf("Very small");
    } else {
        printf("Big");
    }
    return 0;
}
```

Exercise: Rewrite the above code without nested if statements.

More Conditionals

Ternary Conditional Operator

`expr1 ? expr2 : expr3`

- If `expr1` is true, this conditional statement returns the value of `expr2`. Otherwise it returns `expr3`.
- For example assuming we have integer *i* and *j*,

```
printf("%d\n", i > j ? i : j);
```

returns the maximum of *i* and *j*.

Switch Statements

Syntax:

```
switch(expr){  
    case const_expr: statements break;  
    ...  
    case const_expr: statements break;  
    default: statements  
}
```

- The `break;` command lets C know that this case is finished.
- You are not permitted to have multiple equal cases.

Example

```
#include <stdio.h>
int main(void) {
    int i;
    printf("Enter an integer between 1 and 3.");
    scanf("%d",&i);
    switch (i){
        case 1:
            printf(" 1 is the loneliest number\n");
            break;
        case 2: printf(", 2 is an even prime\n"); break;
        case 3: printf(" 3 is a crowd\n"); break;
        default: printf(" illegal entry\n"); break;
    }
    return 0;
}
```

Without Breaks

What happens if you don't put in breaks?

```
#include <stdio.h>
int main(void) {
    int i;
    printf("Enter an integer between 1 and 3.");
    scanf("%d",&i);
    switch (i){
        case 1:
            printf(" 1 is the loneliest number\n");
            break;
        case 2: printf(" 2 is an even prime\n");
        case 3: printf(" 3 is a crowd\n"); break;
        default: printf(" illegal entry\n"); break;
    }
    return 0;
}
```


Previous Slide Explanation

- On user input of 2, we will print “is an even number” and “is a crowd”. If this is the behaviour you want (called a fall through), you probably want to let the programmer know with a comment.
- Note you can also put multiple cases on the same line if they do the same thing

Your Turn!

Write a program that requests any integer and prints the sum of the digits to the screen. For example, if the user enters 1234 then the answer 10 ($=1+2+3+4$) is printed to the screen.

Special Cases

What special cases do you have to consider for your code?

- Negative numbers?
- Non-integer values?
- Zero?
- Integer capacity? (Probably want less than 9 digits).