Overview

- identifiers
- variables and constants
- arithmetic and logical expressions
- printf
- scanf
- if (...) then ... else ...

Identifiers

An identifier is a name for a value.
The rules for forming identifiers in C are:

- They must begin with a letter or underscore ("_") character
- They can use any combination of letters, digits, and underscore
- They must not contain any other characters
- Note that upper case and lower case letters are considered to be different.
  - eg *firstNum*, *firstNUM* and *firstnum* are all different identifiers.

Identifiers

In this course we must follow the Style Guide `https://wiki.cse.unsw.edu.au/info/CoreCourses/StyleGuide` , which is more restrictive.
The rules for forming identifiers in the Course Style Guide are that:

- They must be valid C identifiers AND
- They must begin with a lower case letter
- They must not use any underscore characters
- single letter variables should be avoided unless they are loop counters
- identifier names should be meaningful
- where identifier names are composed of several words, the first word should be in lower case and the first letter of each subsequent word should be in upper case
  - eg *myFirstVariable*

Identifiers

For example, all of date and temperature and mean and big_long_name and numItems and __funny are valid identifiers.
However big_long_name and __funny do not conform to our style guide.

Use meaningful identifiers, so that names reflect the quantities being stored and manipulated.

What is wrong with: fast-food, and 76trombones, and #_of_words?

Identifiers

Some words are reserved, because they have special significance and are keywords in the C language.
For example, int while return and if
There are about two dozen such special words in C.

Constants and Variables

Constants are fixed values that do never change during the execution of the program.
They are introduced using the "#define" facility:
#define PI 3.1415
#define SEC_PER_MIN 60
#define MIN_PER_HOUR 60
#define EXAM_PC 55
#define PRAC_MARK_PC (100-EXAM_PC)
#define KM_PER_MILE 1.609
Different names should be used for different concepts, even if they have the same value.
Make sure you follow the Style Guide when naming your constants.
Constant names should be all uppercase with underscores between words.

Constants and Variables

Make sure you follow the Style Guide and use all uppercase with underscores, separating words for your constants.

Use symbolic constants for all fixed values manipulated in your programs, to improve readability, and to facilitate subsequent modification.

Magic Numbers

You will lose style marks if you fail to use defined constants and instead use magic numbers everywhere.

Constants and Variables

Variables can change their value during program execution.
Assignment statements are used to assign variables the values generated by evaluating expressions. The assignment sum=sum+next causes the current value of the expresssion sum+next to be assigned to the variable sum.
This is different to mathematics, where $s = s + n$ implies that $n = 0$.

Constants and Variables

More examples:

```
nitems = nitems+1;
miles = km/KM_PER_MILE;
totSeconds = (hours * MIN_PER_HOUR + min) * SEC_PER_MIN + secs;
finalMark = (examMark * EXAM_PC + pracMark * PRAC_PC) / 100;
```

Exercise

Using suitable identifiers, write assignment statements to compute
(a) the total surface area and
(b) the total edge length, of a rectangular prism of edge lengths a, b, and c.

Constants and Variables

Variables must be declared before they are used.
The declaration specifies a type for that variable.

Each variable in a program has a type associated with it.

Declaring a variable to have a certain type does not assign a value, and use of uninitialised variables is a common programming error. An uninitialised variable will contain a random value.

Variables must be assigned values before they can be used in expressions.

Constants and Variables

The simplest type is int. Variables of type int store integer-valued numbers in a constrained range, often -2^{31} to $2^{31}-1$, i.e. -2,147,483,648 to +2,147,483,647
These bounds are a consequence of 4 bytes (32 bits) being used to store the variable

Constants and Variables - overflow

Hard question: What is the output by this program fragment?

```
int big, bigp1, bigt2, bigp1t2;

big = 2147483647;
bigp1 = big + 1;
bigt2 = big * 2;
bigp1t2 = bp1 * 2;
printf("big=%d bigp1=%d\n", big, bigp1);
printf("bigt2=%d ", bigt2);
printf("bigp1t2=%d\n", bigp1t2);
```

Constants and Variables

Not what you think!
big=2147483647 bigp1=-2147483648
bigt2=-2 bigp1t2=0
The computation has exceeded the ability of the computer to represent integers.

Beware. Most C implementation don't check for integer overflow leading to incorrect values propagating through the remainder of any computation without amy warning message.

Possible solution

Use variables of type long long. On lab machines 8 bytes (64 bits). TThis is suffcieient to avoid overflow for many applications. If this is insufficient, you can use a floating point types (double). More about it later.

I/O

`printf` and `scanf`

- `printf`(*control string*, *expression list*)
- `scanf`(*control string*, *address list*)

The control string for `printf` may contain plain text together with conversion specifiers and/or escape sequences (`\n`, `\t`, etc.).

The control string is followed by a number of expressions (these are often just variables) equal to the number of conversion specifiers in the control string.

The control string for `scanf` often contains only conversion specifiers. It is followed by a number of variable addresses equal to the number of conversion specifiers in the control string.

Don't forget to use the the address-of operator for scanf, e.g., the address of `x` is `&x`.

I/O

`scanf` Conversion Specifiers

`%d`	corresponds to the `int` type
`%ld`	corresponds to the `long int` type
`%lld`	corresponds to the `long long int` type
`%f`	corresponds to the `float` type
`%lf`	corresponds to the `double` type
`%c`	corresponds to the `char` type

In addition, most conversion specifiers can be optioned for finer control, e.g., `%.3f` instructs `printf` to use a precision of three.

The Unix `man` command

To find out more check out your textbook and/or use the `man` command in a Unix terminal, e.g., `man printf`.

Arithmetic Operators

C supports the standard mathematical operations in the form of the binary operators: $*$, $/$, %, $+$, $-$. Operators act on *operands*.

The $*$ operator represents multiplication (since there is no $\times$ on the keyboard) and % is the modulus (remainder) operator.

What is the value of the following expression?

```
1 + 2 * 3 - 2 / 2
```

Not sure, because we don't know what the order of evaluation is. But it turns out that C supports operator precedence and the result is what we would expect from maths.

Arithmetic Operator Precedence

Operators `*`, `/`, `%` have equal and higher precedence than `+`, `-`, which also have equal precedence.

Arithmetic Operators

Remember that C is a typed language. Arithmetic operations are valid for all numeric types, with the exception of `%` which is only valid for integer types. If both operands are of the same type then the result of the operation will be of that type.

What is the value of the expression: `1 / 2` Answer: `0`!

Integer Division

The result of integer division is the truncated integer quotient!

Negation (Unary -)

The negation operator works as expected, it changes the sign of its argument. It has the highest precedence of all arithmetic operators, i.e., higher than multiplication, etc.

```
int x = 100;
int y = -x;
```

Exercise

Discuss with your neighbour

What are the values of the following expressions?
6* 7 - 8 * 9/10
2*3*4+5*6
5*6/4
1.0/2.0
1/2.0

Relational Operators

Relational operators allow us to answer questions.

"Is x greater than y?" – They return a Boolean (true/false or yes/no) answer.

Specifically, the C relational operators behave as follows:

- they return an `int` value
- the value 0 for false
- the value 1 for true

Boolean values in C

Many languages have a special Boolean type, however C makes do with integer values, with the convention that 0 stands for false and every other value stands for true.

Relational Operators

The C relational operators by example:

`5 > 4`	$\mapsto$	`1`	greater than
`5 >= 4`	$\mapsto$	`1`	greater than or equal to
`5 < 4`	$\mapsto$	`0`	less than
`5 <= 4`	$\mapsto$	`0`	less than or equal to
`5 != 4`	$\mapsto$	`1`	not equal to
`5 == 4`	$\mapsto$	`0`	equal to

Don't confuse equality (==) with assignment (=)!

Be careful when using them. Something like: `16 > 4 > 2` will compile (albeit with a compiler warning), but what does it mean?

Precedence

The operators >, >=, <, <= have higher precedence than !=, ==.

Logical Operators

Logical operators allow us to combine Boolean expressions (e.g., comparisons, etc.). We use them to answer questions like "Is x greater than y and less than z?"

The logical operators are:

- and (`&&`) true if both operands are true
- or (`||`) true if either operand is true
- not (`!`) true if its operand is false

The complete truth tables are found in your textbook (Table 3.2).

Here are some examples:

`(2 > 0) && (2 < 10)`	$\mapsto$	`1 && 1`	$\mapsto$	`1`	and
`(0 > 1) || (2 < 10)`	$\mapsto$	`0 || 1`	$\mapsto$	`1`	or
`!(0 > 1)`	$\mapsto$	`!0`	$\mapsto$	`1`	not

Logical Operators

Precedence

The operator `!` has higher precedence than `&&` which in turn has higher precedence than `||`.

Short-circuit evaluation

This is an important concept, the operators `&&` and `||` evaluate their left-hand-side operand first and only evaluate their right-hand-side operand if necessary.

Operator `&&` only evaluates its RHS if the LHS is *true*.

Operator `||` only evaluates its RHS if the LHS is *false*.

This is very useful because we can safely write:

```
(x != 0) && (y / x > 10)
```

Precedence

A list of all operators in order of precedence, from high to low:

- `!x, -x`
- `x * y, x / y, x % y`
- `x + y, x - y`
- `x < y, x <= y, x > y, x >= y`
- `x == y, x != y`
- `x && y` (short-circuit left to right)
- `x || y` (short-circuit left to right)
- `x = y`

Explicit Order

The evaluation order can be changed and/or made explicit via parentheses, e.g., `7 * (4 + 3)`.

Associativity

Associativity

Associativity dictates the order of evaluation for (binary) operators with the same precedence. Assignment (=) is right-associative, all others are left-associative.

Consider the expression:

$$7 - 4 + 3$$

Left-associative evaluation (what we expect):

$$(7 - 4) + 3 \Rightarrow 6$$

Right-associative evaluation:

$$7 - (4 + 3) \Rightarrow 0$$

Control Flow

Our C toolset has grown considerably, however we still don't know how to solve a simple problem like: "read two integers and print the larger one."

- we can read two integers
- we can compare them
- but how do we print the larger one?

What we need is a way of making choices in our programs. This functionality is known as control flow or branching and is provided by the `if` statement.

The `switch` statement

You can read about it in your textbook but we will not cover it in this course and you should refrain from using it.

The `if` Statement

This is the structure of the `if` statement:

```
if (EXPR) {
  STMTS1
} else {
  STMTS2
}
```

If *EXPR* evaluates to a non-zero value, *STMTS1* are executed, otherwise *STMTS2* are executed.

The `else`-branch is optional:

```
if (EXPR) {
  STATEMENTS1
}
```

The `if` Statement

Multiple `if` statements can be chained together:

```
int a, b;

printf("Please enter two numbers, a and b: ");
scanf("%d %d", &a, &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");
}
```

The `if` Statement

This syntax is also valid:

```
if (a == 0)
    printf("a is zero\n");
a = 1; // this does not belong to if-block
```

If the braces ({}) are not supplied then the `if` statement controls only the statement that immediately follows.

Always use braces!

Doing this will ensure that you avoid bugs and ambiguity. The style guide requires it.

Summary

We looked at:

- some stuff from last week
- arithmetic operators
- relational operators
- logical operators
- the `if` statement
- programming!

More information about these topics can be found in Chapters 2 - 3.4 of the text book.

cp

Copy files and directories

- `cp` *sourceFile destination*
  - Note: If the destination is an existing file, the file is overwritten; if the destination is an existing directory, the file is copied into the directory
- To copy a directory use `cp -r` *sourceDir destination*

mv

Moves or renames a file.

- `mv` *source destination*
  - Note: If the destination is an existing file, the file is overwritten; if the destination is an existing directory, the file is moved into the directory.

rm

Removes/deletes a file. (Note: This is permanent. It does not move files to a recyle bin)

Be careful with this command

- `rm` *filename*
- `rm -r` *directoryName*
  - This will delete a whole directory.
  - Be extra careful with this command