Overview

• identifiers
• variables and constants
• types
• arithmetic expressions
• printf
• scanf

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.

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

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.

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 expression sum+next to be assigned to the variable sum.

So if sum was 2 and next was 1 the expression sum+next would evaluate to 3. And now this value would be stored in sum.

This is different to mathematics, where \( s = s + n \) implies that \( n = 0 \).
Variables and Types

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.

Types

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.

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

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

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 any warning message.
Overflow and Other Types

Possible solution if int is not sufficient

- Use variables of type long long. On lab machines these are 8 bytes (64 bits). This is sufficient to avoid overflow for many applications.
- If this is insufficient you could try an unsigned long long if you were working with large positive numbers (Why?)
- If this is still insufficient, you can use a floating point types (double).

Types for Real Numbers

- If we want to store real numbers then we could use a float or a double
- By default we recommend you use double rather than float as double has better precision.
- Note: There will still be precision errors even with double. Be careful with these types
- If you are interested in exactly how these are stored google IEEE 754 Standard. This is not eximnab in this course.

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
- `%llu` corresponds to the unsigned 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 3 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 × on the keyboard) and % is the modulus (remainder) operator.

What is the value of the following expression?

\[ 1 + 2 \times 3 - 2 / 2 \]

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.

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
- 3/2
- 1.0/2.0
- 1/2.0

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:

\[ 7 - (4 + 3) \Rightarrow 0 \]

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.

```c
int x = 100;
int y = -x;
```
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 $c$.

**Exercise 1**

```java
// Exercise 1 code

double totalSurfaceArea;
double totalEdgeLength;

// Assignment statements for exercise 1

```

**Exercise 2**

```java
// Exercise 2 code

double totalSurfaceArea;
double totalEdgeLength;

// Assignment statements for exercise 2

```