Complete (but elementary) C programs

- operators and assignment
- sequence, selection and repetition
- functions
- pointers - introduction
- getting input
- arrays and array parameters

announcements:

- Assignment 1 (part 1) up; also answers to week 2 Tute
- confirm assessment scheme?
Operators

● arithmetic:

<table>
<thead>
<tr>
<th>operator</th>
<th>integer operands</th>
<th>floating point operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ addition</td>
<td>23 + 10 = 33</td>
<td>22.5 + 10.3 = 32.8</td>
</tr>
<tr>
<td>− subtraction</td>
<td>23 − 10 = 13</td>
<td>22.5 − 10.3 = 12.2</td>
</tr>
<tr>
<td>* multiplication</td>
<td>23 * 10 = 230</td>
<td>22.5 * 10.3 = 231.75</td>
</tr>
<tr>
<td>/ division</td>
<td>23 / 10 = 2</td>
<td>22.5 / 10.3 = 2.18446(···)</td>
</tr>
<tr>
<td>% remainder</td>
<td>23 % 10 = 3</td>
<td>N/A</td>
</tr>
</tbody>
</table>

● classifications:

■ unary operator: one operand, e.g. + (6), − (10.43), ~112
■ binary operator: two operands, e.g. 23 + 10, 23 % 10, 17 ~ 114, 0x8 | 0x1
■ ternary operator: can you find one?

● relational operators:

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>x &gt; 100</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal to</td>
<td>x &gt;= 20</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>x &lt; 100</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less or equal to</td>
<td>x &lt;= 20</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
<td>x == 100</td>
</tr>
<tr>
<td>!=</td>
<td>non-equality</td>
<td>x != 100</td>
</tr>
</tbody>
</table>
Assignment Statements

- lvalue = rvalue  e.g. year = 1900 + 99;
  - value of righthand side is stored in lefthand side
  - rvalue is a variable, constant or expression
  - lvalue is a variable

- an assignment statement has a value (that of the righthand side)

  year2 = year = 1968;

- minimal type checking! e.g. with int year; float length;

  year = 1968.29;  /* year becomes 1968 */
  length = 2 / 3;  /* length becomes 0.0 */
  length = 2.0 / 3;  /* length becomes 0.66... */

- = is different to the equality operator ==

  if (year2 == 1968) ... /*possibly false */
  if (year2 = 2001) ... /*always true */
  if (year2 = 0) ...  /* always false */
String Assignments and Operations

- string operations require library calls: `month = "Dec" + "ember"` does not work – no in-built string operators!
- we have to use the string library `string.h` (see `man string` on student system)
  ```c
  #include <string.h>
  ...
  char month[9];
  ...
  strcpy(month, "Dec");
  strcat(month, "ember");
  if (strcmp(month, "December") != 0)
    printf("Doh!\n");
  ```
- in C, strings are implemented as arrays of `char`
  - individual elements of the string variable `month`:
    ```c
    month[0], month[1], ..., month[8]
    ```
    can be accessed the same way as `char` variables, e.g. `month[0] = 'd';`
Precedence, Type Hierarchy and Shorthand Notation

- quiz from lecture C1!
- logical operators ([King, table 5.3]); bitwise operators ([King, table 20.1-2])
- arithmetic precedence: [King, sect 4.1]
- general operator precedence: [King, table 4.2]
- data type conversion hierarchy: [King, sect 7.5]
- shorthand operators: e.g. \( i += 1; \)
- increment and decrement operators: e.g. \( i++; \)
Sequencing and Selection

- statements are separated by semicolons ";" and are executed in the order they appear in the function (program code)

- a statement may be a (or include) call to a function and the statements of that function are then executed first. e.g.

```
int doit() {
    int z;
    ...
    return z;
}
```

- selection: alternative instruction (sequences) according to some condition(s)

```
if (i == 1) {
    /* statement sequence 1 */
} else if (i > 45) {
    /* statement sequence 2 */
} else if (i < -42) {
    /* statement sequence 3 */
} else {
    /* statement sequence 4 */
}
```
Repetition

- three loop constructs in C:
  - `while (condition) { ... }
  - `for (start; condition; inc/decrement) { ... }
  - `do { ... } while (condition);

- use `break` to exit loops in special cases
- use `continue` to ‘skip’ to the bottom of loop
- never use the `goto` statement!
- e.g. `while` and `for` loops

```c
int main(void) {
    int i;
    i = 0;
    while (i < 10) {
        printf("%d
", i);
        i = i+1;
    }
    return 0;
}

int main(void) {
    int i;
    for (i = 0; i < 10; i = i+1) {
        printf("%d
", i);
    }
    return 0;
}
```
Functions

- functions must be declared before they are used
  - e.g. `#include <stdio.h>` for `printf()` definition
  - the definition (i.e. implementation) of the function can come later in the program (or even in a different module)

- a function declaration or ‘header’ gives its ‘type signature’; it consists of:
  
  ```c
  return_type function_name (parameter_list);
  ```

  e.g.:
  ```c
  int add(int x, int y);
  int main(void);
  void display(float x, float y);
  ```

- this gives sufficient information for compiler to tell if function is called properly
  - hence header files are sufficient to use external libraries

- parameters are passed-by-value, so they won’t be changed
  - are effectively local variables to the function, except they are initialised when it is called
Example: function.c

```c
#include <stdio.h>
int nextYear(int y);

int main(void) {
    int a, b;
    a = 1981;
    b = nextYear(a);

    if (b == 1982) {
        printf("1982\n");
    } else {
        printf("Help!\n");
    }
    printf("Final value of a=%d\n", a);

    return 0;
}
int nextYear(int y) {
    int newYear;

    y = y+1;
    /* change y, but not a! */
    newYear = y;
    return newYear;
```
Introduction to Pointers

● a pointer is something which may or may not exist
  – Anonymous

● variables have 5 attributes: name, type, size, value and address
  ■ size is determined by the type
  ■ given (in bytes) by the `sizeof()` operator; e.g. `sizeof(i), sizeof(int)

● a pointer is a variable containing a memory address;
  usually this is the address of another (non-pointer) variable

● we can use the memory address to read or modify the value of the variable the
  pointer refers to. e.g.

```c
int y = 1; int *x;
x = &y;  *x = *x + 1;
```

- `int` * type for pointer to `int`
- `&y` address of `y`
- `*x` variable pointed to by `x`

● why are these not valid?

```c
&100  &’A’  &(k+2)
```
Getting Input – scanf()

- The **stdio** library provides functions to read values from keyboard and other sources (like files)

  ```c
  int scanf(const char *format, ...);
  ```

- The **format** string contains conversion specifications indicating how many, and what type of values to read

- The subsequent parameters are pointers to the variables in which `scanf()` should store the values

  ```c
  e.g. scanf("%d", &year);
  ```

- `scanf()` uses whitespace characters (return, tab and space) to decide how to divide the input into separate fields

- E.g. `scanf.c`

  ```c
  int a, b;
  scanf("%d-%d", &a, &b);
  printf("sum=\%d\n", a+b);
  ```
Arrays

- a group of $n$ variables of the same type (e.g. a vector), stored sequentially in memory
- a single name applies to the whole array; individual elements accessed using an index in the range $[0, n-1]$, e.g.

```c
int a[3]; /* declares array of length 3 */
int x = 0;
a[0] = 42;
a[1] = 68;
a[2] = a[0] + 2*a[1]; /* this statement is OK */
a[3] = 1;       /* index is too big! what will happen? */
```

- array names can be used as pointers
  - the name is actually a pointer to the first element of the array
  - since array elements are stored contiguously in memory, subsequent elements can be accessed by doing pointer arithmetic, e.g.

```c
int a[3];
*a = 42;    /* same as a[0] = 42; */
*(a+1) = 68; /* same as a[1] = 68; */
```

- address of $(a+1) = a + 1 \times \text{sizeof(int)}$
Array as Function Parameters: uppercase.c

- while an array parameter (pointer) is passed by value, the memory locations of the array elements are thus effectively passed by reference
- i.e. the function can change the array's elements

```c
#include <stdio.h>
#include <string.h>

void uppercase(char string[]);

int main(void) {
    char vowels[6];
    strcpy(vowels, "aeiou");
    printf("%s\n", vowels);
    uppercase(vowels);
    printf("%s\n", vowels);
    return 0;
}
```

```c
// alt.: declare as char *string
void uppercase(char string[]){
    int i = 0;
    while (string[i] != '\0') {
        string[i] = string[i]
            - 'a' + 'A';
        i++;
    }
}
```
For Next Lecture: Look at string.c

```c
#include <stdio.h>
int main(){
    char name[100];
    printf("Enter your name\n");
    scanf("%s", name);
    printf("Your name is %s\n", name);
    return 0;
}
```

- why is there no & in this `scanf()`?
- run the program and input both your first and lastname separated by a space.
  
  What is printed out and why?