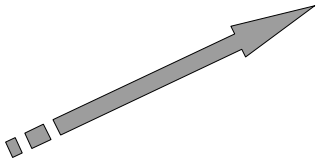
Lecture 4



1. Introduction
2. Binary Representation
3. Hardware and Software
4. High Level Languages
5. Standard input and output
6. Operators, expression and statements
7. Making Decisions
8. Looping
9. Arrays
10. Basics of pointers
11. Strings
12. Basics of functions
13. More about functions
14. Files
14. Data Structures
16. Case study: lottery number generator

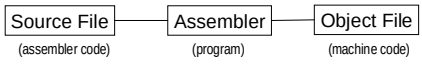
Machine Code

- As we have seen, programs exist as binary machine code. This is inconvenient for humans as we can not naturally interpret the code.
- To overcome this, assembly language was invented, e.g.

```
LDA 1BC9    - load accumulator from address 1BC9
ADD 05      - add 5 to accumulator
STA 1AF7    - store the result at address 1AF7
```

- This is translated more or less directly into machine code by a program called an assembler

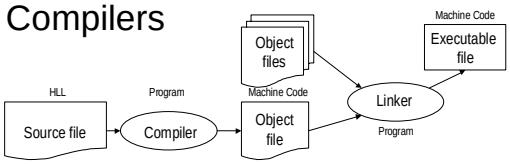
Assembler



- For complex programs we need a greater level of abstraction which is offered by high level languages such as c

e.g.
```
x=newWindow(200,300,x_pos,y_pos);
y=1+sin(theta)-cos(phi)/2;
```

Compilers



- A compiler is used to convert the HLL source code into an object file, which is then possibly combined with others to produce an executable program.
- Other object files can come from other programmers, libraries (e.g. maths, graphics), assembler or other HLLs

Errors

- Several types of error can occur when developing programs. *They are all introduced by the programmer, not the computer!*
  - Compile time errors (reported by the compiler)
    a) Syntax errors, e.g. missing semicolon at end of statement
    b) Semantic errors, e.g. forgetting to #include <stdio.h>, then using printf (which therefore doesn't mean anything)
  - Run time errors
    the program compiles ok but crashes when it is ran, e.g. z=x/y; where y=0
  - Algorithmic errors
    the program compiles and runs ok, but doesn't do what is intended.

Anatomy of a Simple C Program

```
/* Example: anatomy of a simple C program. */

/* It's useful to put comments in your programs to explain
what you're doing. Not too few, not too many: one helpful
comment every 10 to 20 lines of code is probably about
right. This program has far too many! */

/* Note: C doesn't allow nested comments: you can't put one
comment inside another! */

#include <stdio.h>
/* This is a preprocessor directive.
Most C programs need to include the standard header file
<stdio.h> which tells the compiler about input and output facilities,
it declares the function printf, for formatted printing. */

#define ANSWER 42
/* This is another preprocessor directive.
The symbol ANSWER will be replaced by 42. It's useful for
defining constants. */

int a, b;
/* Declared outside any function, these are external
(global) variables, accessible to all functions in this
file. There are good reasons for minimising the use of
globals! */

int mult_and_print(float f, float g);
void add(void);
/* Function prototypes. Their actual definitions follow
main. */

main()
{
  /* This is a function definition. Every C program needs a
     main function, where execution starts. */

  float x, y = 13.7;
  /* Declaring two local variables, visible only within the
     main function */
```

Anatomy of a Simple C Program cont……

```
    x = y * ANSWER + a;  /* This is a statement */

    add();
    b = mult_and_print(x, y);  /* These are function calls */

    return;  /* End of the main function */
}

void add(void)  /* Definition of a function */
{
    a = a + b;
    return;
}

int mult_and_print(float f, float g)
/* definition of a function */
{
    int r;  /* Declaring a local variable */

    r = (int) (f * g);  /* A statement */

    printf("Yesterday I saw %i penguins!\n", r);
    /* A function call; the prototype for printf is in
    <stdio.h> */

    return r;  /* Returning a value to the calling
                  function */
}
```

Simple Data types in C

- C has only a few simple data types:
  - Integer types
    `int, short, long`
  - Floating point types:
    `float, double, long double`
  - Character types
    `char`
- The integer and character types can be either signed or unsigned
- NB. C does not define how many bytes each type uses. The compiler does !!!

Integers

- Consider an `unsigned int` variable. Suppose the compiler uses 2 bytes for this data type.
  It can vary from 0000 to FFFF (hex), i.e. 0 to 65535 (decimal)

  e.g.
  ```
  unsigned int x=65535;
  x=x+1;
  ```
- Whats the value of x now ?
  Its 0000 (hex) - overflow has occurred.

One and Two'2 Complement

- Negative integers are stored in a form known as one and two's complement
- Standard Binary number

128	64	32	16	8	4	2	1
0	0	0	0	1	0	0	1 = 9

- One's Complement, Invert the number

-127	64	32	16	8	4	2	1
1	1	1	1	0	1	1	0 = -9

- Two's Complement, Invert the number and add 1

-128	64	32	16	8	4	2	1
1	1	1	1	0	1	1	1 = -9

Two'2 Complement

- So when overflow occurs +127+1 gives??
- 127=

-128	64	32	16	8	4	2	1
0	1	1	1	1	1	1	1 = 127

- +1

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0 = -128

overflow.c

```
/* Example: demonstrates overflow
of integer variables */

/* Don't worry about how the program works, just
appreciate what happens! */

#include <stdio.h>
#include <limits.h>

void main(void)
{
    unsigned int x = UINT_MAX - 1;
    signed int y = INT_MAX - 1;

    printf("x is an unsigned int, occupying %i bytes.\n\n", sizeof(x));

    printf("The initial value of x is %u\n", x);
    x++;
    printf("Add 1; the new value of x is %u\n", x);
    x++;
    printf("Add 1; the new value of x is %u\n", x);
    x++;
    printf("Add 1; the new value of x is %u\n", x);

    printf("\ny is a signed int, occupying %i bytes.\n\n", sizeof(y));

    printf("The initial value of y is %i\n", y);
    y++;
    printf("Add 1; the new value of y is %i\n", y);
    y++;
    printf("Add 1; the new value of y is %i\n", y);
    y++;
    printf("Add 1; the new value of y is %i\n", y);

    return;
}
```

```
x is an unsigned int, occupying 4 bytes.

The initial value of x is 4294967294
Add 1; the new value of x is 4294967295
Add 1; the new value of x is 0
Add 1; the new value of x is 1

y is a signed int, occupying 4 bytes.

The initial value of y is 2147483646
Add 1; the new value of y is 2147483647
Add 1; the new value of y is -2147483648
Add 1; the new value of y is -2147483647
```

sizes.c

```
/* Example, compile and run this program to find out how your C compiler
treats different data types. The result will depend on the computer and
compiler you use, and possibly on the compiler settings.

*/
#include <stdio.h>
#include <limits.h>
#include <float.h>

void main(void)
{
  puts("With this computer and compiler, the simple C");
  puts("data types have the following characteristics:\n");

  puts("signed char:");
  printf("\tOccupies %i byte (%i bits)\n", sizeof(signed char), CHAR_BIT);
  printf("\tRanges from %i to %i\n\n", CHAR_MIN, CHAR_MAX);

  puts("unsigned char:");
  printf("\tOccupies %i bytes (%i bits)\n", sizeof(unsigned char), CHAR_BIT);
  printf("\tRanges from %i to %i\n\n", 0, UCHAR_MAX);

  puts("signed short:");
  printf("\tOccupies %i bytes\n", sizeof(signed short));
  printf("\tRanges from %i to %i\n\n", SHRT_MIN, SHRT_MAX);

  puts("unsigned short:");
  printf("\tOccupies %i bytes\n", sizeof(unsigned short));
  printf("\tRanges from %i to %i\n\n", 0, USHRT_MAX);

  puts("signed int:");
  printf("\tOccupies %i bytes\n", sizeof(signed int));
  printf("\tRanges from %i to %i\n\n", INT_MIN, INT_MAX);

  puts("unsigned int:");
  printf("\tOccupies %i bytes\n", sizeof(unsigned int));
  printf("\tRanges from %i to %u\n\n", 0, UINT_MAX);

  puts("signed long:");
  printf("\tOccupies %i bytes\n", sizeof(signed long));
  printf("\tRanges from %li to %li\n\n", LONG_MIN, LONG_MAX);

  puts("unsigned long:");
  printf("\tOccupies %i bytes\n", sizeof(unsigned long));
  printf("\tRanges from %i to %lu\n\n", 0, ULONG_MAX);

  puts("float:");
  printf("\tOccupies %i bytes\n", sizeof(float));
  printf("\tRanges from +/-(10e-1 to 10e1) approx.\n", FLT_MIN_10_EXP, FLT_MAX_10_EXP);
  printf("\tPrecision is %i decimal digits approx.\n\n", FLT_DIG);

  puts("double:");
  printf("\tOccupies %i bytes\n", sizeof(double));
  printf("\tRanges from +/-(10e-1 to 10e1) approx.\n", DBL_MIN_10_EXP, DBL_MAX_10_EXP);
  printf("\tPrecision is %i decimal digits approx.\n\n", DBL_DIG);

  puts("long double:");
  printf("\tOccupies %i bytes\n", sizeof(long double));
  printf("\tRanges from +/-(10e-1 to 10e1) approx.\n", LDBL_MIN_10_EXP, LDBL_MAX_10_EXP);
  printf("\tPrecision is %i decimal digits approx.\n\n", LDBL_DIG);

  puts("pointer:");
  printf("\tOccupies %i bytes\n", sizeof(void *));
  puts("");
  return;
}
```

```
With this computer and compiler, the simple C
data types have the following characteristics:

signed char:
        Occupies 1 byte (8 bits)
        Ranges from -128 to 127

unsigned char:
        Occupies 1 bytes (8 bits)
        Ranges from 0 to 255

signed short:
        Occupies 2 bytes
        Ranges from -32768 to 32767

unsigned short:
        Occupies 2 bytes
        Ranges from 0 to 65535

signed int:
        Occupies 4 bytes
        Ranges from -2147483648 to 2147483647

unsigned int:
        Occupies 4 bytes
        Ranges from 0 to 4294967295

signed long:
        Occupies 4 bytes
        Ranges from -2147483648 to 2147483647

unsigned long:
        Occupies 4 bytes
        Ranges from 0 to 4294967295

float:
        Occupies 4 bytes
        Ranges from +/-(1E-37 to 3E+38) approx.
        Precision is 6 decimal digits approx.

double:
        Occupies 8 bytes
        Ranges from +/-(1E-307 to 1E+308) approx.
        Precision is 15 decimal digits approx.

long double:
        Occupies 8 bytes
        Ranges from +/-(1E-307 to 1E+308) approx.
        Precision is 15 decimal digits approx.

pointer:
        Occupies 4 bytes
```

Integer types

- In the examples we used the `sizeof` keyword to determine how many bytes are used to store each type.

  `sizeof(unsigned int)` or `sizeof(x)`

  data type — variable
- The names `short` and `long` can be misleading, on in the last example `int` and `long` were the same size.

  `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`

Floating Point Numbers

- Numbers like 1234.567 can be expressed in "scientific notation" as 1.234567×10^3, or +.1234567 e+4

  mantissa — exponent
- We let the decimal point *float* to the LH end of the mantissa
  - Such as representation is called *floating point*
- We can use a similar method to generate a binary code, typically 4 bytes are used to store floating point N's
  - The majority of bits are used for the mantissa and a few for the exponent.
  - Both parts are stored as 2's complement
- The float type is often poor precision (6 sig figs) far worse than a calculator
  - C also has a *double* precision floating point type , `double`, which typically uses 8 bytes

Characters

- There is only one character type, `char`
- This is defined (in ANSI C) to be a single byte.
- Depending upon the context it can be treated as an integer, so there are signed and unsigned versions
  - `char` (-128 to +127) and
  - `unsigned char` (0 to 255)
- A char variable holds ASCII codes as well as some escape characters
  - `'\n'` = new line, `'\t'` = tab, `'\a'` = alert (beep), `'\0'` = null
- The `null` character has the code 0000000 (binary) and has a special role in strings.

Names rules

- If we wish to create a new variable, say a floating point variable called x, this is done by

  `float x;`
- However there are restrictions on the names that we can use.
  - Can only include letters, digits or underscores '_'
  - First character cannot be a digit
  - Upper and lower case are distinct (All upper used for #define)
  - Must not be the same as any of C's 32 keywords
    - `auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while`
  - Is is also sensible not to use other common C identifiers, e.g. printf

Declaring Variables

- In C, all variables must be declared before they can be used. Typical declarations :
  - `int i,j,k;`
  - `unsigned long very_big_thing=1000000;`
  - `char c;`
  - `char ch='a',NL='\n';`
  - `float x,y=4E-6,z=0.0015f;`
- Note that variables can be initialised (given a value) at their declaration, if desired.

Constants

- It makes sense to use symbols for constants, e.g. π instead of 3.14159…, to save typing, avoid errors and improve readability
- We could define a variable
  `float pi=3.14159;`
  but it is better C style to use a `#define`
  `#define PI 3.14159`
- This means that where ever "PI" appears in the program, the pre-processor substitutes "3.14159"
  - The pre-processor is a program that processes the source file just before the compiler proper starts
- Conventionally such constants are in CAPITALS

define.c

```
/* Example: constant definition using #define */

#include <stdio.h>

#define PI 3.142

/* To change this to a more accurate value,
(e.g. 3.14159265358979323846) throughout the program,
only one line needs to be edited.
*/

main()
{
  double r, c, ac, as, v;

  r = 5.678;
  printf("Radius = %f\n", r);

  c = 2.0 * PI * r;
  printf("Circle's circumference = %f\n", c);

  ac = PI * r * r;
  printf("Circle's area = %f\n", ac);

  as = 4.0 * PI * r * r;
  printf("Sphere's area = %f\n", as);

  v = 4.0/3.0 * PI * r * r * r;
  printf("Sphere's volume = %f\n", v);
}
```