ICT106
Fundamentals of Computer Systems- Topic 2

REPRESENTATION AND STORAGE OF INFORMATION

Reading: Linux Assembly Programming Language, Ch 2.4-2.9 and 3.6-3.8

Final Labs and Tutors

- **Venue and time**
  - South Street: PS 1.16F
    - Steven Van Der Werf: Wed (Gp 1) 9:30-11:30; Wed (Gp 2) 11:30-13:30
    - Eric Li: Wed (Gp 3) 13:30-15:30
  - South Street: ECL 2.046
    - James Mei: Fri (Gp 5) 8:30 – 10:30, Fri (Gp 6) 14:30-16:30
    - Eric Li: Fri (Gp 6A) 14:30-16:30

Objectives

- Understand how information is represented inside a computer
- Understand the binary, octal and hexadecimal notations
- Understand the number representation schemes
- Understand how character and string data are represented
- Know how instructions are represented
- Know the conversion between decimal, binary, octal and hexadecimal notations
- Know how to read and display decimal, octal and hexadecimal integers in C

Basic Concepts

- Computer memory is organised into *bytes/words*
- Computers can store many *types* (or formats) of data/information
- HLLs can have different *data types*: integer, real, character,...
- Inside a computer, data/information can be stored as collections of discrete representation in voltage, electrical charges or magnetic levels. In a binary digital logic system, two distinct levels are required. For example, high or low voltage levels.
- Different levels result essentially from electronic switching circuits which are either on or off (eg. capacitors are charged/discharged)
Basic Concepts

• Conceptually: **binary** information is represented by only **two states**,
  - on or off
  - true or false
  - 1 or 0
• Information is represented as collections of 0's and 1's (**bits**)
  - 8 bits make a **byte**
  - 2 or more bytes make a **word**
  - The no. of bits in the word is called its **wordlength** (word size)

Representing Information as Bit Strings

• Series of 0's and 1's are called bits or bit strings (binary digit - base 2 numbers) eg base 2
  - 101011₂
• Information is stored as collections or sequences or strings of bits
• Number of these bits defines a particular representation - eg. 8 bit, 16 bit, 32 bit

Decimal System

• In the decimal system, we know a number can be viewed as a sum of its products, eg
  \[325 = 3 \times 10^2 + 2 \times 10^1 + 5 \times 10^0\]
• Here each digit position has a certain weight, i.e., from right to left \(10^0, 10^1, 10^2\)
  that multiplies the digit value
• To represent a decimal number, a decimal point is used:
  \[28.75 = 2 \times 10^1 + 8 \times 10^0 + 7 \times 10^{-1} + 5 \times 10^{-2}\]

Binary and Related Systems

• In binary (base 2), octal (base 8) and hexadecimal (base 16) systems, numbers employ the same kind of positional notation, the binary number: \(101011\)
• can be viewed as
  \[1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]
  which is
  \[32 + 0 + 8 + 0 + 2 + 1 = 43_{10}\]
Binary Number and Binary Point

• Similarly, $11.101_2$ can be viewed as

$$11.101_2 = 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 2 + 1 + 0.50 + 0 + 0.125$$

$$= 3.625_{10}$$

Binary Addition

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 1 = 0$ plus carry 1 to the next column

Example:

$1111010_{10}$

$+ 1010111_{10}$

$11010001_{10}$

Binary Subtraction

• Here borrows are used in stead of carries

Example:

- $1111010_{10}$
- $- 1010111_{10}$
- $0100011_{10}$

Hexadecimal Number System (Base 16)

• Binary numbers are difficult for humans to readily identify

• Hex number system can act as an interim between binary and ourselves as it is easier to recognize hex numbers yet can be readily converted to/from binary

• One hexadecimal digit is equivalent to 4 bits, so 1 byte can be used to store two hex digits.
Hexadecimal Number System (Base 16)

- Four binary digits (bits) at a time are grouped together to form one hex symbol (digit), so 1 byte can be used to store two hex digits.
- Four binary digits have 16 possible states (or values) thus need 16 different symbols.
- So, the hexadecimal system uses the digits 0 through 9 and the letters A through F (upper or lower case) corresponding to the bit sequences 0000, 0001, 0010, ..., 1001, 1010, 1011, 1100, 1101, 1110, 1111.

Hexadecimal Number System

- The next number after F is “zero and 1 to carry”, that is 10, designated as $10_{16}$.
- When writing hexadecimal numbers in documentation (not in programs) postfix the number with a ‘H’ or ‘h’, eg 157H, AB12H, 157h. If the first digit is A-F, it may be necessary to append 0 in front to indicate a numeric value. (An other convention is to use a ‘$’ sign to represent Hex numbers. This is used in Motorola convention.)
- In C/C++ programs, hex numbers are prefixed with ‘0x’, eg 0x157

Octal Number System (Base 8)

- In the octal system 3 binary digits are grouped together to form an octal digit.
- Therefore, digits 0 through 7 give 8 possible states corresponding to the bit pattern 000, 001, 010, 011, 100, 101, 110, 111.
- The next number after 7 is “zero and one to carry”, that is 10, usually designated as $10_8$.
- In C/C++ programs, octal numbers are prefixed with 0 (zero), eg, 0145

Number System Conversions

<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Binary</td>
</tr>
<tr>
<td>Div by 16</td>
<td>Div by 8</td>
</tr>
<tr>
<td>Binary</td>
<td>Multiply and add</td>
</tr>
<tr>
<td>Octal</td>
<td>Multiply and add</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>Multiply and add</td>
</tr>
</tbody>
</table>
Base 10 to Base N

1) Divide the number by the base N, and note the Quotient and the Remainder
2) Divide the Quotient by the Base again recording the resulting Quotient and Remainder
3) Repeat step 2 until Quotient is 0
4) The succession of Remainders (from last to first) is the number in the new base.

Example

• convert $35_{10}$ to Binary
  35/2 gives Quotient 17, Rem 1
  17/2 gives Quotient 8, Rem 1
  8/2 gives Quotient 4, Rem 0
  4/2 gives Quotient 2, Rem 0
  2/2 gives Quotient 1, Rem 0
  1/2 gives Quotient 0, Rem 1
  Therefore, $35_{10} = 100011_2$

• Similarly, to convert to hexadecimal system, divide the decimal number repeatedly by 16 as above.

Conversion between Binary, Octal and hexadecimal

• Base 8 (octal) to Base 2 (binary):
  Each octal digit is split into 3 binary digits, working right to left. Eg,
  $43_8 = 100 \ 011_2$
• Base 2 to Base 8:
  3 binary digits are combined to form 1 octal digit, working right to left. Eg,
  $111101_2 = 1 \ 111 \ 101_2 = 175_8$

• Base 2 to Base 16:
  4 binary digits are combined to form 1 hex digit, working right to left. Eg,
  $111101_2 = 111 \ 101_2 = 7D_{16}$
• Base 16 to Base 2:
  Each hex digit is split into 4 binary digits, working right to left. Eg,
  $A5_{16} = 1010 \ 0101_2$
Conversion between Binary, Octal and hexadecimal

- **Base 16 to Base 8 or Base 8 to Base 16:**
  Convert to binary first using technique mentioned earlier. Then convert to target base using one of the earlier techniques. Eg,
  \[ 7D_{16} = 0111 \ 1101_2 \]
  \[ = 01 \ 111 \ 101_2 \]
  \[ = 175_8 \]

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Oct</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td>19</td>
<td>10011</td>
<td>23</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>10100</td>
<td>24</td>
<td>14</td>
</tr>
</tbody>
</table>

**Units**

- 1 byte = 8 bits
- 1 KB (KiloByte) = 1024 bytes \((2^{10})\)
- 1 MB (MegaByte)
  - = approx. 1,000,000 bytes
  - or 1024 KB or \(2^{20}\) bytes
- 1 GB (GigaByte)
  - = approx. 1,000,000,000 bytes
  - or 1024MB or \(2^{30}\) bytes.

**Bit Labelling**

- **MSB** - most significant bit - left most bit, Also called the *high order* bit
- **LSB** - least significant bit - right most bit Also called the *low order* bit
- Bits are **numbered from right to left** (starting with LSB being bit 0).
Example

• In the case of a byte
  MSB       LSB
  \[\begin{array}{ccccccc}
  0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
  7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
  \end{array}\]

• Similarly, in a 16-bit word, bit 0 is the LSB and bit 15 is the MSB.
• A bit may be set (1) or cleared (0)

Representing Information as Bit Strings (roc stop)

• Representing Integer Data
• Non-negative (unsigned) integers:
  • Can be represented by using all bits in the bits string to store the magnitude of the integer value
  • Eg, if all bits of a byte are used for the magnitude, the highest number we can store in a byte is 11111111, which is 255

Range of unsigned integers

• In an n-bit word we can represent non-negative integers in the range 0 to \(2^n-1\)
• For n=16, the range is 0 .. 65535
• For n=32, the range is 0 .. 4,294,967,295
• The data type unsigned int of C uses this representation.

Signed integers

• Two possible systems:
  • Sign-magnitude System
    • The left most bit (MSB) is used to represent the sign of the number (0 means positive, 1 means negative)
    • The other bits represent the magnitude of the number, eg, in a 3-bit word string (just an example !)
    \[
    \begin{array}{c|c}
      \text{Binary} & \text{Decimal} \\
      \hline
      001 & +1 \\
      010 & +2 \\
      011 & +3 \\
      100 & 0 \\
      101 & -1 \\
      110 & -2 \\
      111 & -3 \\
    \end{array}
    \]
Limitation

- Note that the number zero has two representations in this system 000 and 100, so 000 and 100 are considered identical in this representation.
- Problem with this representation is that it does not lend itself to binary arithmetic which is performed using logic circuits.

Two's Complement System

- In this system, the MSB provides an indication of the sign: 0 means positive, 1 means negative
- Eg, a 3-bit word can be used to represent numbers from -4 to +3
- If the value of MSB is zero, the rest of the bits in the word represent a positive normal binary number, i.e., a number in the range 0 to 3

Two’s Complement System

- If the value of MSB is one, this represents -4 which is to be added to the total of other 2 bits (assuming 3-bit word) that represent a normal binary number.
  - 001 represents 1 (a +ve number, i.e., MSB is 0, the total of other 2 bits is 1)
  - 010 represents 2
  - 011 represents 3

Two's Complement System

- 101 represents -3 (a -ve number, i.e., MSB is -4, to which we add the total of other 2 bits, i.e. 1, giving -3)
- 111 represents -1 (i.e. -4 is added to the total of other 2 bits)
Similarly,
- 100 represents -4
- 110 represents -2
Range Limit

- In the above example, an attempt to go beyond +3 would give 100₂, which is the same as the pattern assigned to -4.
- Similarly, an attempt to yield a value more negative than -4 would give 011₂, which is the same as the pattern assigned to +3.
- It is therefore necessary to recognize range limits with digital representation of numbers.

Limits of 2’s complement system

- In general, the range of numbers that can be represented in a Two’s complement system is:
  \[-(2^{n-1}) .. + (2^{n-1} – 1)\]

Steps in finding two’s complement of a negative number

1. Find n bit base-2 representation of positive number
2. Logically Complement number: replace 0’s with 1’s and 1’s with 0’s (this step also called 1’s complement)
3. Add 1 to the result
   - You should understand the limitations of representations in relation to bit string length (and overflow)

Overflow

- 2 numbers to be added have the same sign (both +ve or -ve) and the result has the opposite sign.
- Overflow can happen even if there is no carry.
Example of Overflow

• eg (+5) + (+4) in 4 bits
  0101
  0100
  1001
  1011
  Overflow:
  is a negative number (that is, -7)
• eg. (-7) + (-6)
  1001
  1010
  10011
  Overflow:
  is a +ve number. (that is, +3)
  Note: there is a carry of 1

Examples of 8-bit numbers and their Two’s Complement

15 = 00001111
   11110000
   + 1
   ------------
   11110001
   10

-15 = 10000001
    01111110
    + 1
    ------------
    01111111
    2 = 127

Another Example

-127 = 10000001
      01111110
      + 1
      ------------
      01111111
      2 = 127

Addition in 2’s Complement System

Addition: N1 + N2 (each is n-bits)
Add N1 and N2, ignoring carry from MSB

Example: 21 + 11
 assume n = 8 (no. of bits)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>00010101</td>
</tr>
<tr>
<td>+ 11</td>
<td>+ 00001011</td>
</tr>
<tr>
<td>32</td>
<td>00100000</td>
</tr>
</tbody>
</table>


Subtraction in 2's Complement System

Difference: \( N_1 - N_2 \) (each is \( n \)-bits)
Add \( N_1 \) to the Two’s Complement of \( N_2 \), ignoring the carry from MSB

**Example 1:** \( 21_{10} - 11_{10} \)
Convert 11 into its two’s complement form, then add it to 21

\[
\begin{align*}
21 & \quad 00100101 \\
11 & \quad + 11110101 \\
\hline \\
10 & \quad 10000101
\end{align*}
\]
• There is a carry from the MSB, which is ignored. The result is: \( 00001010_2 \)

**Example 2:** \( 11_{10} - 21_{10} \)
Convert 21 into its two’s complement form, then add it to 11

\[
\begin{align*}
11 & \quad 00001011 \\
21 & \quad + 11101011 \\
\hline \\
-10 & \quad 10101110
\end{align*}
\]

**Example 3:** \( -11_{10} - 21_{10} \)
Convert both 11 and 21 into its two’s complement form, then add:

\[
\begin{align*}
-11 & \quad 11101010 \\
-21 & \quad + 11101101 \\
\hline \\
-32 & \quad 11100000
\end{align*}
\]
• There is a carry from the MSB, which is ignored. The result is: \( 11100000_2 \)

**Representing Real Data**

- Real numbers (also called ‘floating point numbers’) are represented in scientific notation where the magnitude is separated into 2 parts:
  - the mantissa ‘\( m \)’ and the exponent ‘\( n \)’
  - \( m \times 10^n \)
- eg. \( 3.25 \times 10^{-5} \) for the number 0.0000325
  - where
    - mantissa = 3.25
    - exponent = -5
- or, \( 0.325 \times 10^{-4} \) - this number is said to be normalised

Most CPUs including the iAPX (i.e., intel 8080 – 80786) class of computers use 2’s complement system for storing integers.
Real Numbers in Binary

• To represent real numbers in a computer (binary form), the notation takes the form:
  \[ m \times 2^n \]
  where, both mantissa and exponent are binary and are stored as individual bit strings
  eg, \( 832_{10} = 1101000000_2 \) can be written as \( 0.1101 \times 2^{10} \)
  where
  mantissa = \( 1101_2 \)
  exponent = \( 1010 = 1010_2 \)

• Similarly, \( 101.0011_2 \) can be written as \( 0.1010011_2 \times 2^3 \); and \( 0.00101_2 \) can be written as \( 0.101_2 \times 2^{-2} \)

• Note the obvious complications when storing real numbers – two different pieces of information need to stored:
  mantissa (including its sign) and exponent (including its sign)

Real Numbers in Binary

• Each mantissa and exponent field is represented as a signed integer
• Each field usually has very different bit string lengths
• BUT one entire bit string still used to represent one real number

• Eg, on the 80x86 computers where 32-bit string is commonly used to store a real number, the mantissa (or Significand) is represented using first 23 bits. The exponent is represented using 8 bits. MSB is used as the sign bit.
• Note that the real numbers are stored as an approximation only.
• The precision of the number is determined by the number of bits used to represent the mantissa.
Real Numbers in Binary

- The more bits that are used to store the mantissa the more precisely the number can be specified.
- The “size” (e.g., largest number) of the number depends on the number of bits used to represent the exponent.
- The more bits used to store the exponent gives a greater range of numbers.

- In summary, bit strings are used to represent floating point numbers (some bits allocated for storing the exponent and others to the mantissa) and the precision of the numbers depends on the field size (no. of bits) of each.
- Most modern HLLs provide 2 types of real numbers – single precision (float) and double precision (double).

Representing Character Data

- In a typical language such as C/C++, 4 bytes are used for floats and 8 bytes are used for doubles.
- Also note that different computer systems use different representations for floating point numbers. They vary in the no. of bytes used, the order of bytes, and also whether 2’s complement or signed integer representations are used.

- Characters consist of:
  - Letters: A, B, C, ..., Z, a, b, c, ..., z
  - Digits: 0, 1, 2, ..., 9
  - Special chars: +, *, /, ^, $, %, space, ...
  - Non-printing chars: Line Feed (LF), Carriage Return (CR), ...
ASCII Codes

• Each character is represented as a distinct bit pattern according to some established character-encoding convention
• ASCII (American Code for Information Interchange) is a widely used convention – it uses one byte for every character.
• Alternatively, the leftmost bit is used as parity bit for error detection.

ASCII Code

• Examples:

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>01000001₂ = 65₁₀ = 41₁₆</td>
</tr>
<tr>
<td>'W'</td>
<td>01010111₂ = 87₁₀ = 57₁₆</td>
</tr>
<tr>
<td>space</td>
<td>00100000₂ = 32₁₀ = 20₁₆</td>
</tr>
<tr>
<td>'.'</td>
<td>00111100₂ = 46₁₀ = 2E₁₆</td>
</tr>
<tr>
<td>'2'</td>
<td>00110010₂ = 50₁₀ = 32₁₆</td>
</tr>
</tbody>
</table>

Other schemes exist (eg EBCDIC, Unicode), but not looked at in this unit. Unicode uses 2 bytes for every character, thus can represent up to $2^{16}$ (65536) characters – sufficient to represent just about every character and symbol in every language (plus some empty spaces for future growth)!

• Note that numeric characters are quite different from numbers represented in a system such as 2’s complement, eg,
• The number 7 is represented in 8 bits by 00000111
• The character 7 (i.e. ‘7’) is represented in 8-bit ASCII code by 00110111
• Strings are stored simply as collections of individual characters, eg the string *HELLO THERE* may be stored in computer memory as follows:

<table>
<thead>
<tr>
<th>High Byte</th>
<th>Low Byte</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01000101</td>
<td>01001000</td>
<td>E</td>
<td>H</td>
</tr>
<tr>
<td>01001100</td>
<td>01001100</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>00100000</td>
<td>01001111</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>01001000</td>
<td>01010100</td>
<td>H</td>
<td>T</td>
</tr>
<tr>
<td>01010010</td>
<td>01000101</td>
<td>R</td>
<td>E</td>
</tr>
<tr>
<td>01000101</td>
<td></td>
<td></td>
<td>E</td>
</tr>
</tbody>
</table>

**Representing Machine Instructions**

• Each type of computer has a *machine language* – a series of low level instructions.
• Each instruction is a group of bits (bit string) which tells the computer to perform one of its basic functions – arithmetic, compare etc.
• A program is a list of these instructions.

**Machine Instructions**

• Each instruction consists of 2 parts:
  *Operation code* – tells the computer what to do
  One or more *operands* – identify the data or memory locations which will be operated upon

**What Type of Information is Stored?**

• Given a bit string, how do we know what does it represent?
• Can’t tell until context or purpose is known!
• Same problem occurs within a computer
• Everything within a computer is a bit string
A program must start at the “right” place!

- Given that the hardware can guarantee starting execution at a particular machine instruction – it is the responsibility of programmer to put appropriate further machine instructions and data at the correct places and time!
- Compilers generate machine code which attempt to make sure representations are interpreted as they are supposed to be interpreted - when the program is executing.

Other information storage

- bit sequences which represent instructions are interpreted as such
- bit sequences which represent different types of data are interpreted as such.
- Similar to computers - information which is saved on discs, external memory cards, transmitted over data communication cables and networks, ... all represented as bit strings

Organization of Main Memory

- Words are stored at memory addresses
- While a program is executing, together with the data, they are stored in main memory (RAM)
- The memory must hold the bit strings which represent the machine instructions and data
- By convention and convenience, the bit string lengths are fixed or at least multiples of some base unit

Basic unit of memory

- For nearly all modern computers, the base unit is a byte: 8 bits
- Computer memory is simply a very large collection of these bytes
- Each byte has an address, addressed being numbered from 0 upwards
- Typically 2 consecutive bytes are called a word (16 bits)
**Word Size**

- Word size of a computer is usually based upon the number of representations required for the machine instructions - depends on the CPU.
- The increase in word size represents the development of subsequent CPU generations.

**Addresses**

- Address of a word is based on the first byte:
  - Word 0 - bytes 0 & 1,
  - Word 2 - bytes 2 & 3,
  - Word 4 - bytes 4 & 5, …
- Similarly,
  - Word 1 - bytes 1 & 2,
  - Word 3 - bytes 3 & 4, …

**Accessing addresses**

- There are different machine instructions for accessing words vs bytes
- Must be able to store Addresses - so that instructions can reference contents of specific addresses
- Address Size must be identified - number of bits required determines the representation
- Varies amongst computers: 20, 24, 32 bits for different IBM PC and modes of operation; 24, 32 bits for MacIntosh, Commodore 64, Different game computers, etc.

**Memory Size**

- Address Size puts an upper limit on the total amount of memory that can be accessed
- For address size n, the addresses will range from 0 to 2^n-1, so we can have almost 2^n bytes of memory in the system. eg,
  - 16 bit address gives range 0 .. 65535 (64K maximum)
  - 20 bit address: 0..2^20-1 (1048575 bytes = 1MB maximum)
  - 24 bit address: 0..2^24-1 (16,777,216 bytes = 16MB maximum)
  - 32 bit address: 0..2^32-1 (4,294,967,295 bytes = 4096 MB maximum)
Storage of Variables in HLL Programs

- Whilst we use names for variables, the compiler assigns memory addresses to the variables and uses these addresses to refer to them.
- Contents of an address corresponding to a variable represent the value of that variable.
- Number of consecutive bytes allocated for a particular variable depends on the type of that variable, the compiler and the computer.

An example in C

Generally, but not always, consecutively declared variables are assigned the required number of consecutive bytes.

Consider the following C code:

```c
int x, y;
long z;
char u, v;
char m, n;
```

As an exercise, draw a diagram showing memory allocation for the above variables.

Find data type and size from Help

- Use the C/C++ HELP system to look up all the standard types and their sizes. Get help on `printf` and then on conversion type characters – i.e. `things like %d, %c, %s, .., etc.`