Memory Address and Pointers

Content:

- C data types (revisit)
- Data on memory
- Memory addresses
- Address operator and pointers
- More on scanf library function
- Brief note on using const for function arguments

Data on the Memory

Computer Memory

- Data related to variables and function arguments are stored on the RAM.
- The most commonly used unit of computer memory is called Byte.
- In most of the current system each byte is 8 bits.
- The value of a bit is either 0 or 1.
- Each byte of memory has its own unique address, which is number that indicates the location of the memory.

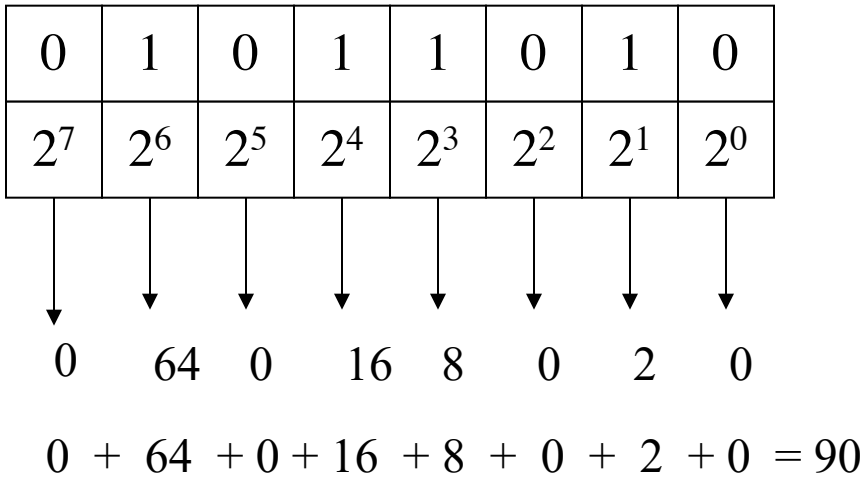
Binary Presentation Data on the Memory

- This is slide and the two following slide are the review of what you have learned in previous programming course (ENGG 233)
  - A computer hardware can only releases two digits:
    - 0 (zero), OFF
    - 1 (one), ON
  - For characters (letters, digits, etc), there is a standard code, called ASCII (American Standard Code for Information Interchange.

Character	Decimal Value	Binary Value
A	65	01000001
Z	90	01011010
2	50	00110010

Binary Presentation of Base-10 Numbers

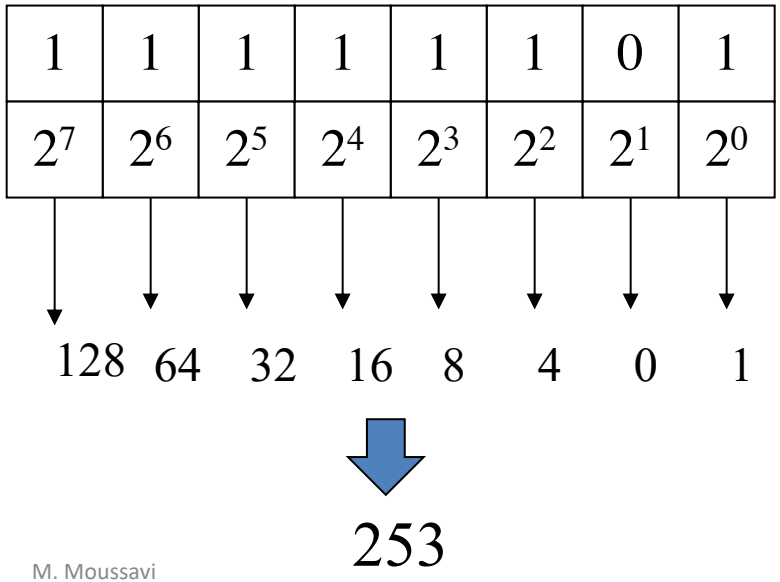
- The following example shows how letter ‘Z’, with ASCII value of 90, is represented in binary system.



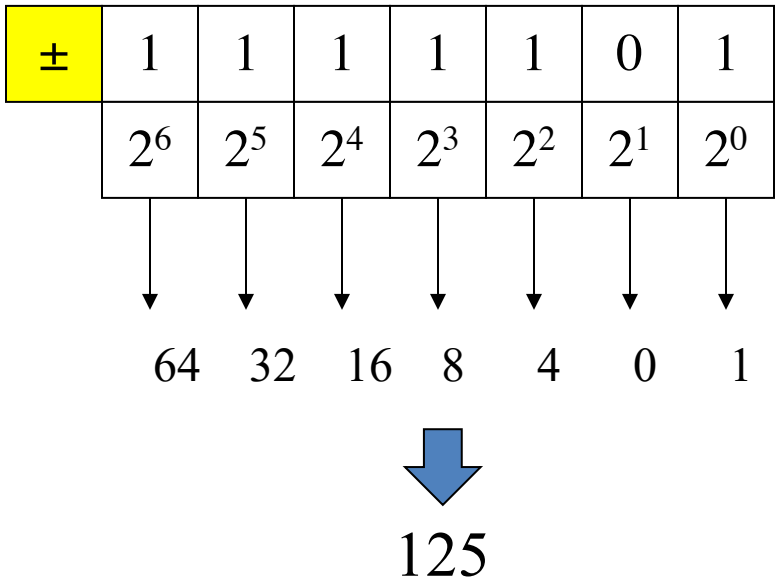
Memory on the Computer

- Memory for integer or integer-like data types can be either signed or unsigned:
  - **Signed** allocated memory uses the left-most bit to indicate the sign of the number (positive or negative).
  - **Unsigned** allocated memory uses all the bits for digits (no bit for sign). Here is visual presentations and example.
  - The range of data that can be stored on one byte signed character is form – 128 to 127. Where the most left bit (the 8^{th} bit) is used for the sign (+ or -).

Unsigned allocated memory



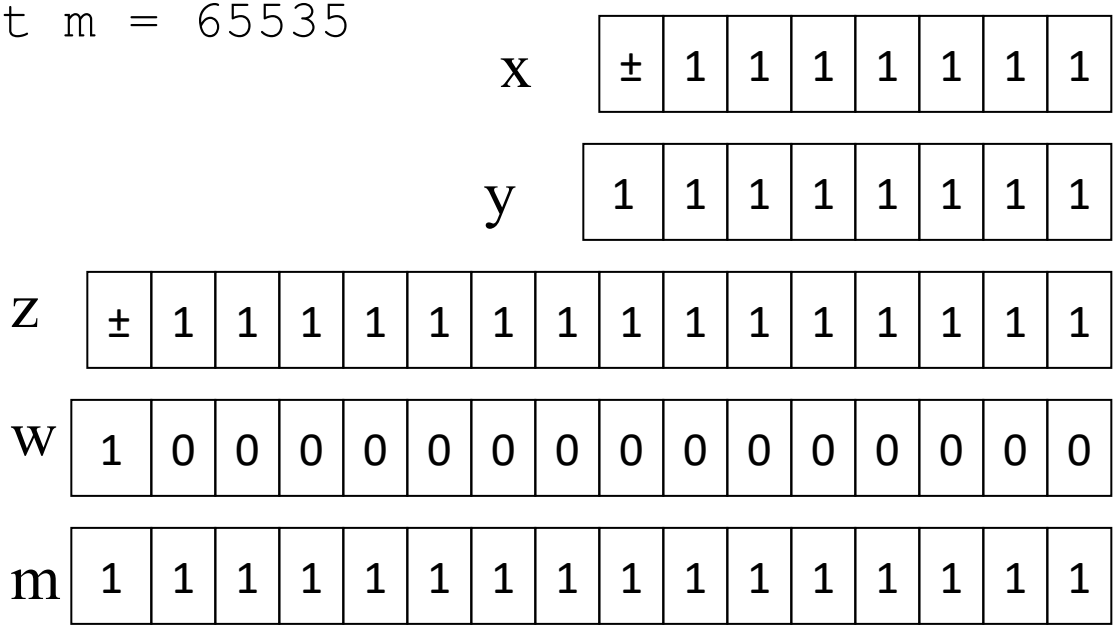
Signed allocated memory



Data Types Capacity

- Each data type has a maximum and minimum capacity for holding data:
  - Some examples of max/min values in different data types:

```
char x = 127;
unsigned char y = 255;
short int z = 32767;
short int w = -32768
unsigned short m = 65535
```



Hexadecimal Notation (Base – 16)

- Among programmer is normally preferred to refer to the memory addresses in hexadecimal notations. Which is a base 16 numbering system.
- Base 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Base 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
- Example: 63 is represented as 3F.
- What is Hex base value of 666?
  - Simply make an integer division by 16 until it turns to zero.

666 / 16 41 Remainder is 10 which is A

41 / 16 2 Remainder is 9

2 / 16 0 Remainder is 2

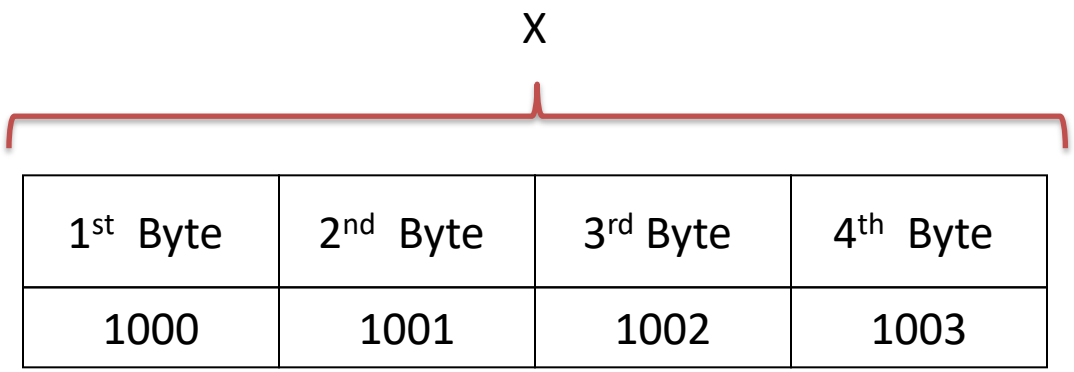
Answer = 29A

Memory and Memory Addresses

Memory Address

- The address of the first byte also represents the address of the variable:
  - E.g.

```
int x;          /* Assuming the size of integer is 4 bytes */
```



In this example the address of x is 1000

Address Operator and Pointers

The "Address of" Operator in C

- C provides an operator which returns the address of the variable immediately after it. The operator is called "address-of" operator and is represented by the '&' sign.
- e.g. .

```
#include <stdio.h>
int main( void )
{
  char myChar = 'a';
  printf("The address of myChar is %lu \n", (unsigned long int)  &myChar );
  return 0;
}
```

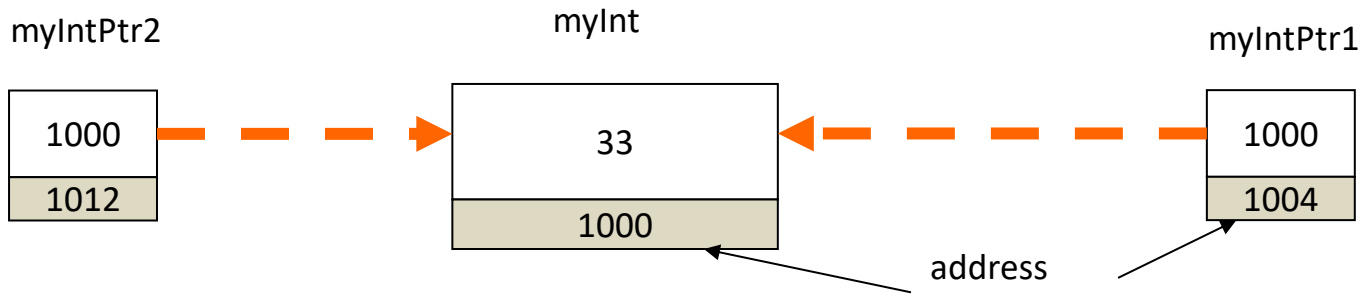
- %lu is a format specifier for unsigned-long-integer and in this example we have used the type-cast operator (unsigned long int) to convert the address of myChar to unsigned long int.
- Please notice that long-integers may have different sizes on different systems. Therefore, a better format specifier for addresses and pointers is %p.
- You can also use %x to display an address in hexadecimal notation.

Introduction to Pointer Types

- A pointer is a data type that can hold the address of another variable of the same type. For example an integer pointer can hold the address of an integer.

```
int myInt = 33;
int* myIntPtr1;        /* an integer pointer which is not initialize, its called a dangling points */
int* myIntPtr2;        /* another integer not initialized */
myIntPtr1 = &myInt     /* integer pointer holds the address of an integer */
myIntPtr2 = &myInt     /* integer pointer holds the address of same integer */
```

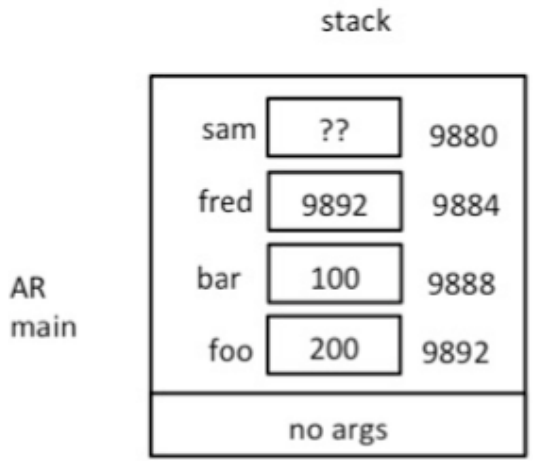
Address Operator



Let's assume the memory space foo, bar, fred, and sam have the given address.

- What are the values in memory space `foo, bar, fred,` and `sam` in the following cod segment when CPU control reaches point two?

```
int foo;
int bar;
int *fred;
int *sam;
bar = 100;
foo = 200;
fred = &foo;
/* point one */
sam = &bar;
*sam += 30;
*fred -= 40;
/* point two */
```



Introduction to Pointer Types

- Now, you can also have access to the value of myInt in the previous slide, indirectly through myIntPtr1 or myIntPtr2:

```
printf ("%d" ,    * myIntPtr1);        /* prints 33 */
```

Called an Indirection or Dereferencing Operator

```
printf("%lu", myIntPtr1);     /* prints 1000 */

printf("%lu", &myIntPtr1);    /* prints 1004 */

*myIntPtr2 = 44;              /* changes the value of of myInt from 33 to 44 */
```

- Draw AR diagram for the following program:

```
int main () {
   int j, k, *p;
   // point one
   p = &j;
   // point two
   *p = 3;
   // point three
   p = &k;
   // point four
   *p = 7;
    // point five
  return 0;
}
```

Pointer Initialization

- The following statements are all known as initialization operations in C and C++:

```
int  a  = 45;
int*  b = &a;
int*  c = NULL;
int*  d = &a;
```

- Where:
  - pointer b is initialized with the address of a (pointing to a).
  - c is a NULL-pointer that points to nowhere.
  - d also points to a
- The following statements are assignment or replacement operations. For example it says the value of b is replaced with the address of a. Or, the value of the location that d is pointing to, is changed to 44.

```
a  = 9;
b = &a;
c = &a;
d = &a;
*d = 44;
```

Pointers as Function Arguments

Passing Data between Functions

- Data between C functions can be either passed by value or by address.

  Passed by Value:

  – Copies of the data will be created in the called function.

  Passed by Address:

  – The called-function receives the address of the data and can have access to the data in the calling-function through a pointer.

- What is the output of the following C program?

```
main() {
     int x = 23,   y = 33,   z = 44;
     modify(x);
     printf("x =   %d y = %d   z = %d" , x, y, z);
     return 0;
     }
```

```
void modify(int x)
{
     x++;

}
```

- Does the program produce the expected output? Does it change the value of x in the main function to 24?

- The modify function in the previous slide doesn't change the value of x in the main.
- There are two solutions to this problem;

Solution 1: incremented x in function modify is returned to main

```
main() {
     int x = 23,  y = 33,  z = 44;
     x = modify(x);
     printf("x =  %d y = %d  z = %d" , x, y, z);
     return 0;
}
```

```
int modify(int x)
{
     x++;
     return x;

}
```

Output: x = 24 y = 33 z = 44

Solution 2: x in main is modified through pointer x in function modify

```
main() {
     int x = 23,  y = 33,  z = 44;
     modify(&x);
     printf("x =  %d y = %d  z = %d" , x, y, z);
     return 0;
     }
```

```
void modify(int* x)
{
     (*x)++;

     // point one

}
```

Output: x = 24 y = 33 z = 44

- Let's draw a memory diagram for point 1 in solution 2.

Another Example

- Although the problem with function modify() in the previous slide could be solved by returning the value of x, but this solution may not work, if the function should modify more than one variables' values.
- What is wrong with the following function swap that is supposed to swap the values of its argument a and b in main?

```
void swap(int x, int y);

int main() {
        int a = 5, b = 8;
        swap(a, b);
        printf("a = %d  b = %d", a, b);   /* prints a=5 and b=8. Which is not what we really expect */
}

void swap (int x, int y) {
        int tmp = x;
        x = y;
        y = tmp;
        // POINT ONT
}
```

- What is the solution? Lets find it out…
- Lets draw an AR diagram for point one

Another Example

```
int main()
{
  int total_in = 75;
  int feet, inches;
  foot_and_inch(total_in, &feet, &inches);
  printf("%d inches is %d feet, %d inches", total_in, feet, inches);
  return 0;
}
```

```
void foot_and_inch(int inch_only, int *p_ft, int *p_in)
{
  // point one
  *p_ft = inch_only / 12;
  *p_in = inch_only % 12;
  // point two
}
```

Exercise: Draw an AR diagram for points one and two in the function foot_and_inch

More on `scanf` Library Function

Input Stream

- C program sees input as a stream of characters, not as a sequence of lines of text. scanf will consume just enough characters to do its job, or to find out that it can't do its job.
- Remaining characters on the input stream are not consumed--instead they remain in the stream, waiting for the next input operation.
- If you type and run this program, you see that the second scanf in the program that is supposed to read a character does not work! Why?

```
#include <stdio.h>
void main( void )
{
  int i;
  char ch;

  printf("Please enter an integer ");
  scanf("%d", &i );

  printf("Please enter a character ");
  scanf("%c", &ch );
}
```

Solution:

- To solve the problem in the previous slide, the input stream (input buffer) must be cleaned up.
- The following code segment cleans up the standard input buffer (what ever is leftover from last read

```
char ch;
do
{
    ch = getchar(); // read a character
} while (ch != '\n' && ch != EOF);
```

Return value from scanf

- `scanf` returns the number of items that reads successfully from standard input stream, or return –1 (`EOF`), if reaches the end of file:

```
int x, status;
double y;
status =  scanf(“ %d%lf ”, &x, &y);
```

If user enters: 234 543.78
x will be 234, *y* will be 543.78, and *status* will be 2.

Read the given listing and answer the following questions:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)  {
   int age, n;
   double wage;
   char gender;

   printf ("Please enter your age, wage and gender: ");
   n = scanf("%d%lf %c", &age, &wage, &gender);

   if(n != 3)
    {
      printf("Invalid input...");
      exit(1);
    }

   printf("Your age is %d, your wage is %6.3f, and your gender is: %c", age, wage, gender);
   return 0;
}
```

- What is the program output if user enters: 22 5.6 M
- What happens if we change %c to %d?
- What happens if user enters: 22 abcd M

Closer look at C Data Types

Data Types and Their Sizes

- To be able to accommodate for different size (range of values) on the computer, every programming language provides a wide range of data types.
- Here is the list most common built-in data types supported in C.
  - Note: Size of some types are system dependent. For example size of `int` can range from 2 to 4 bytes on different machines.
  - The following is the corresponding sizes for each type in our ICT 320 lab
    - char 1 byte
    - Short 2 bytes
    - int 4 byte
    - long 4 bytes
    - float 4 bytes
    - double 8 bytes
    - long double machine/compiler dependent
    - unsigned char 1 byte
    - unsigned int 2, or 4 bytes
    - unsigned long 4 bytes
    - unsigned short 4 bytes
- C99 also supports long long int type: 8 bytes
- Pointer types that we will discuss them later in details are also machine-dependent data types. The size of pointers on typical machines can be 4 or 8 bytes.