Embedded Software Design Techniques

Embedded Software Overview
C programming 1: C language overview

Tsuyoshi Isshiki
Dept. Communications and Integrated Systems
Tokyo Institute of Technology
TAIST ICTES

Conduct by Asst. Prof. Dr. Kanjanapan Sukvichai

Lecture Outline

- Embedded software overview
  - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview + python
  - Function, declaration, statement, expression
  - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model + python
  - Bubble sort vs quick sort
  - Stack memory and program execution
- C programming 3: programming techniques in image processing + python
  - Dynamic memory allocation, image array implementation
  - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications + python
  - Program development steps (ex. Huffman coding)
  - Binary tree construction, tree traversal
  - Bitstream handling
- Real time operating systems and application development + STM32 Maple
  - RTOS services, kernels
  - Context switching, task scheduling
  - Multi-task programming model

Basically add python

About the source programs: source_programs/

- Sorting:
  - qsort5.c : contains quick_sort and bubble_sort
- BMP program:
  - bmp.h, bmp.c : basic BMP file APIs and data structures
  - bmptest2.c : contains main function and others
  - Scenic009smaller.bmp : test image (used in the slides)
- Huffman coding:
  - huff.h: data structures and function declarations
  - huff_io.c: writing and reading compressed files
  - huff_main.c: contains main function and others
  - cppinternals.info: text file (for test input)

How to compile on GCC

- Change to your working directory
- Set path to gcc
  - Type: set path=c:\MinGW\bin;%path%
- To compile a single source file:
  - > gcc qsort5.c
  - → a.exe file is created
- To compile multiple source files:
  - > gcc bmp.c bmptest2.c
  - You don't need to type the header file (bmp.h)
  - → a.exe file is created
- To specify the executable file name
  - > gcc bmp.c bmptest2.c –o b.exe
  - → b.exe file is created

Embedded Software Design Techniques

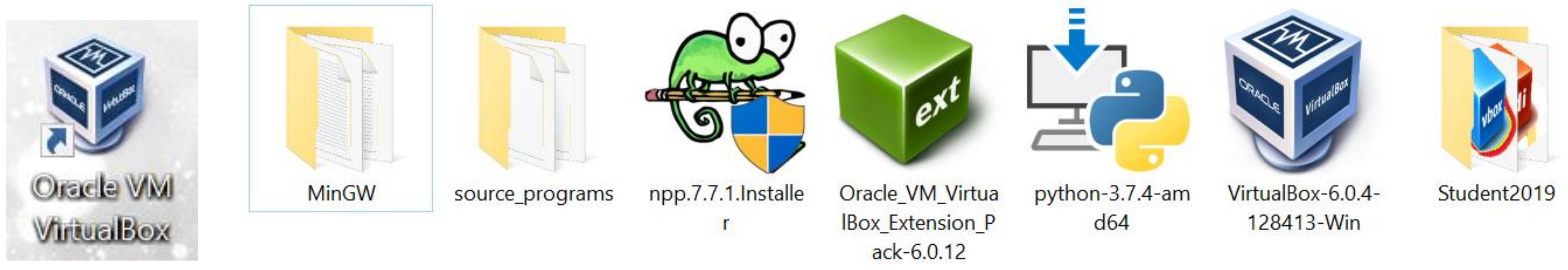
Submitting Exercise problems

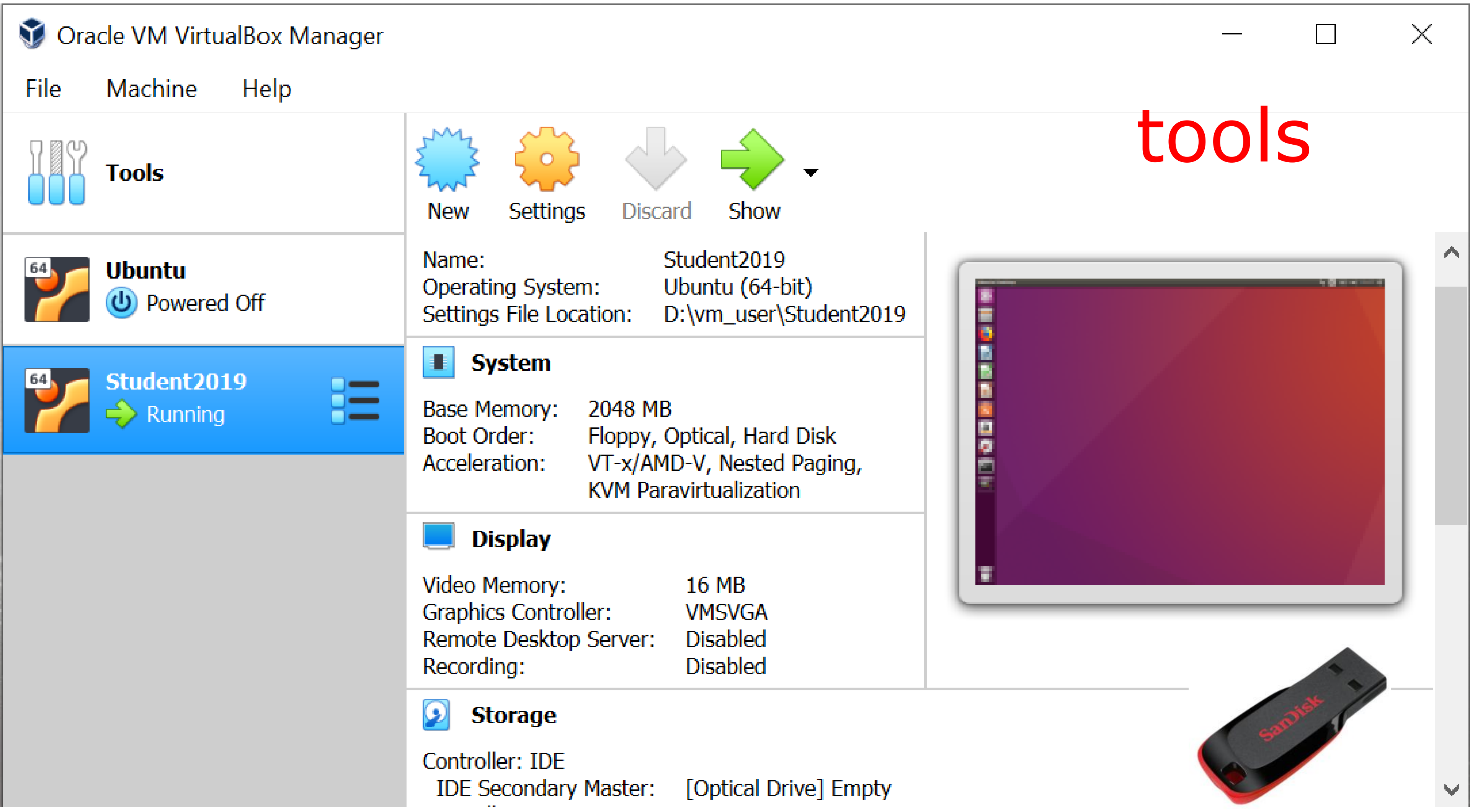
- **Send to: fengkpsc@ku.ac.th, fengdcw@ku.ac.th**
- **Cc: to yourself → so that, in case your email does not arrive here, you can resend it**
- **Subject: ESDT exercise #xx**
- In the text body, describe your name, student ID, and a brief description of the contents of the attached file.
- In the attached file, put your source codes and result outputs
- To dump the printf output to a file, use fprintf(fp, "…", …);

```
FILE * fp = fopen("filename", "w");
fprintf(fp, "%d = %d\n", data0, data1); // replace printf
fclose(fp);
```

Lecture Outline

- Embedded software overview
  - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview
  - Function, declaration, statement, expression
  - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model
  - Bubble sort vs quick sort
  - Stack memory and program execution
- C programming 3: programming techniques in image processing
  - Dynamic memory allocation, image array implementation
  - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications
  - Program development steps (ex. Huffman coding)
  - Binary tree construction, tree traversal
  - Bitstream handling
- Real time operating systems and application development
  - RTOS services, kernels
  - Context switching, task scheduling
  - Multi-task programming model





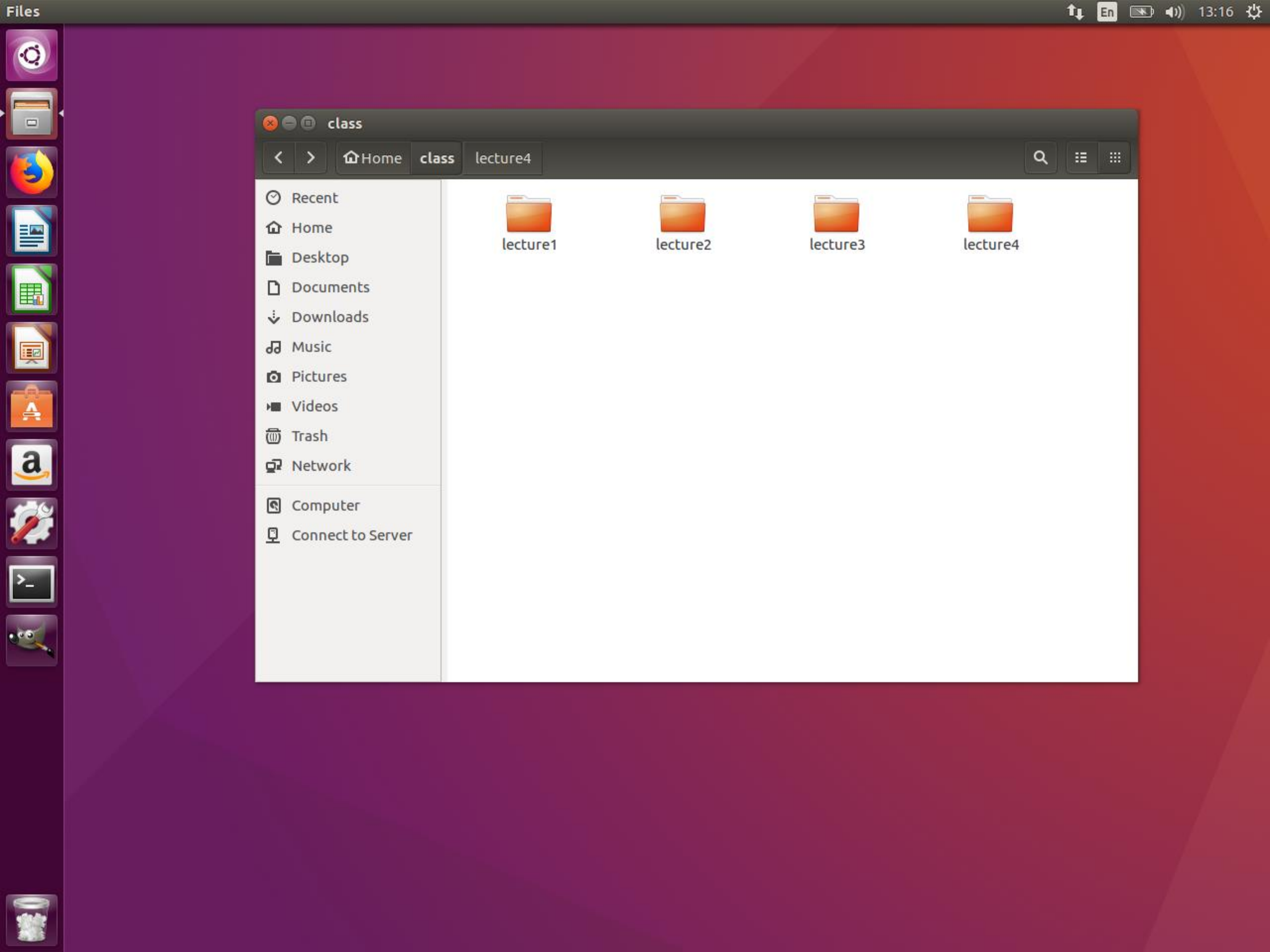
tools

Name: Student2019
Operating System: Ubuntu (64-bit)
Settings File Location: D:\vm_user\Student2019

Base Memory: 2048 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization

Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled

Controller: IDE
IDE Secondary Master: [Optical Drive] Empty


Files
En
13:16
class
Home
class
lecture4
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Network
Computer
Connect to Server
lecture1
lecture2
lecture3
lecture4

What is “Embedded System”??

- Embedded system
  - Definition:
    - “A special-purpose computer system designed to perform a set of dedicated functions” (source: Wikipedia)
  - Attributes:
    - Real time computing constraints
    - “Embedded” as a complete device with hardware and mechanical parts
  - Components:
    - CPUs, memories, dedicated logic blocks (hardware)
    - Software
    - External parts (IO devices, sensors/actuators, switches, network ports)
- Embedded system implementation → System-On-Chip (semiconductor)
  - System integration:
    - Fewer # of components, higher productivity, higher reliability
  - Miniturization:
    - Smaller size/weight, lower power consumption
  - Added values:
    - higher performance, more functions

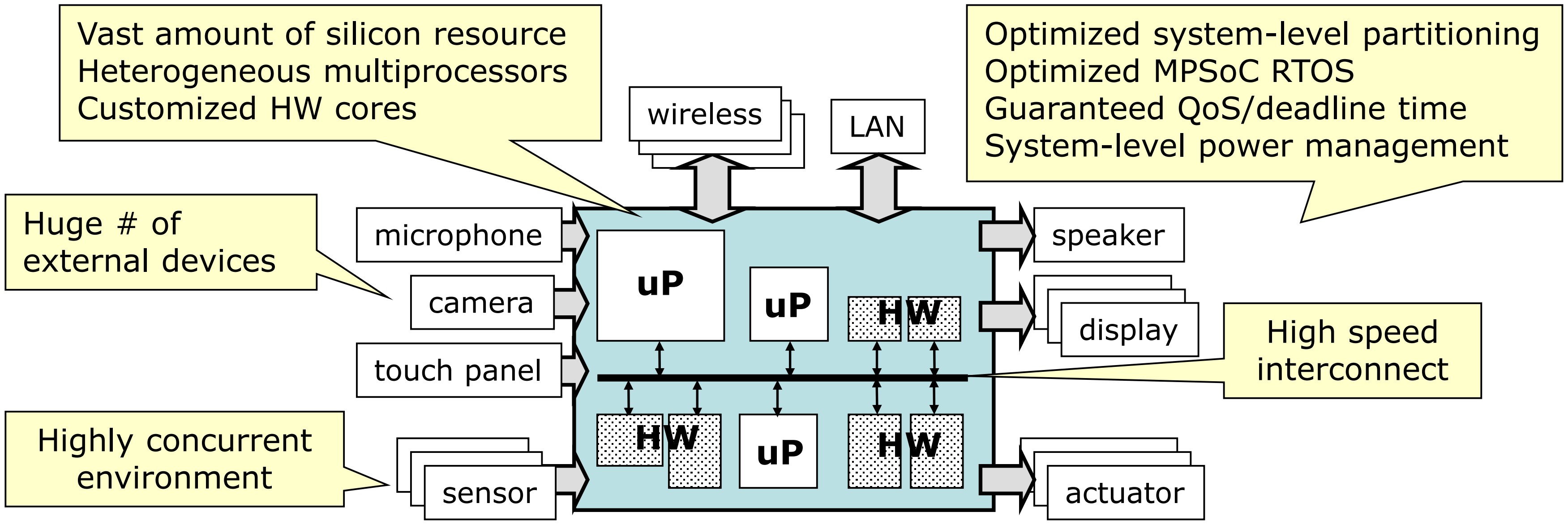
Where is “Embedded System” used?

- Home appliance
  - Microwave oven, rice cooker, refregerator, washing machine, air conditioner
- Consumer electronics
  - Television, video recorder, digital camera, audio player
  - Game, digital instrument, Karaoke
- Computer peripherals, office equipments
  - Printer, scanner, hard-disk drive, CD-ROM/DVD drive
  - Copier, fax
- Communications
  - Telephones, cellular phones, telecom switches, network router
- Transportation
  - Automobiles, traffic lights, trains, airplanes, ships
- Factory automation, plants, buildings
  - Machinery equipments, industrial robots, lighting systems, air conditioning systems, elevators, power management systems
- Medical
  - Blood pressure measurement, heart monitor, CT scanner, MRI, pace maker
- Aerospace, military
  - Rockets, satellites, mission probes, missiles

Embedded Systems Requirement

- Low cost
  - Application-specific system optimizations
- Low power
  - Mobile phone, PDA, notebook PC
- High reliability/security
  - Life-critical systems: automotive, transportation, power plants, building control
- Real-time
  - Control systems
- Time-to-market
  - Adapting to market needs
- Operating conditions
  - Temperature, vibration
- Hardware constraints
  - Size, weight
- Maintainability
  - Fault recovery/repair, field upgrades

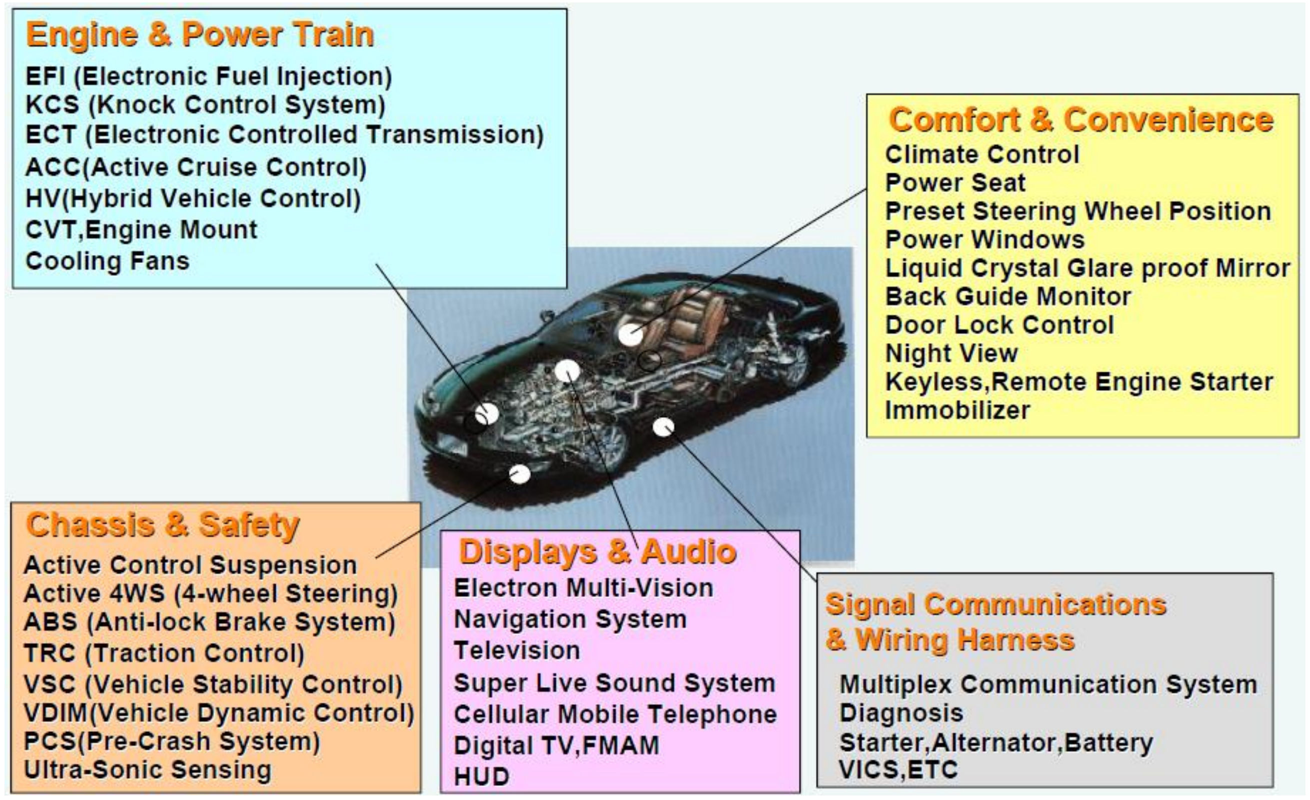
Embedded Systems and SoCs


Vast amount of silicon resource
Heterogeneous multiprocessors
Customized HW cores
Optimized system-level partitioning
Optimized MPSoC RTOS
Guaranteed QoS/deadline time
System-level power management
Huge # of external devices
Highly concurrent environment
High speed interconnect
wireless
LAN
microphone
camera
touch panel
sensor
speaker
display
actuator
uP
HW

- Huge design complexity (SW/HW)
- Complex system environment (#external devices, concurrency)
- Highly-parallel heterogeneous system architecture
- Fully optimized at system-level
  - SW/HW designs (algorithms, cores, architecture)
  - System management (RTOS, guaranteed QoS, low power)

Automotive System

- 70 – 80 ECUs (Electronic Control Units)
- More than 2 million lines of software codes



Engine & Power Train
EFI (Electronic Fuel Injection)
KCS (Knock Control System)
ECT (Electronic Controlled Transmission)
ACC(Active Cruise Control)
HV(Hybrid Vehicle Control)
CVT,Engine Mount
Cooling Fans

Comfort & Convenience
Climate Control
Power Seat
Preset Steering Wheel Position
Power Windows
Liquid Crystal Glare proof Mirror
Back Guide Monitor
Door Lock Control
Night View
Keyless,Remote Engine Starter
Immobilizer

Chassis & Safety
Active Control Suspension
Active 4WS (4-wheel Steering)
ABS (Anti-lock Brake System)
TRC (Traction Control)
VSC (Vehicle Stability Control)
VDIM(Vehicle Dynamic Control)
PCS(Pre-Crash System)
Ultra-Sonic Sensing

Displays & Audio
Electron Multi-Vision
Navigation System
Television
Super Live Sound System
Cellular Mobile Telephone
Digital TV,FMAM
HUD

Signal Communications & Wiring Harness
Multiplex Communication System
Diagnosis
Starter,Alternator,Battery
VICS,ETC

Source: www.dvlsi.jst.go.jp/topics/071201ws/pdf/02presen.pdf

Embedded Software Development

- What languages are used in embedded software develepment?

  (data source: ITRON report 2001)
  - C: 78.5%
  - Assembly language: 11.5%
  - C++: 8.4%
  - Java: 1.2%
  - Others: less than 0.5%

→ So the main objective of this lecture is to learn how to program in C!!!
  - Major portion of this lecture will be devoted for general C programming techniques
  - Programming issues related to hardware will also be focused in some details

Lecture Outline

- Embedded software overview
  - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview
  - Function, declaration, statement, expression
  - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model
  - Bubble sort vs quick sort
  - Stack memory and program execution
- C programming 3: programming techniques in image processing
  - Dynamic memory allocation, image array implementation
  - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications
  - Program development steps (ex. Huffman coding)
  - Binary tree construction, tree traversal
  - Bitstream handling
- Real time operating systems and application development
  - RTOS services, kernels
  - Context switching, task scheduling
  - Multi-task programming model

Where to Find Good C Language References

- On the WEB:
  - http://msdn.microsoft.com/en-us/library/fw5abdx6(VS.80).aspx
    - Microsoft MSDN: contains full description of C and C++ languages
  - http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
    - Very good reference for C standard functions
  - http://cslibrary.stanford.edu/101/
    - “Essential C”: very concise tutorial of C
    - “Pointers and Memory”
    - “Linked List Problems”
    - “Binary Tree”

"hello world!"

```
#include <stdio.h>
void main()
{
     printf("hello world!\n");
}
```

```
Output:
hello world!
```

- `#include <stdio.h>`
  - Include "stdio.h" library header file (to use "printf" function)
- `void main(){ ... }`
  - Definition of "main" *function*
  - All C programs start executing from "main" function
  - Function "printf" is called inside the "main" function
- `printf("hello world!\n");`
  - Prints "hello world!" to console terminal
  - `"\n"` : new-line character

C Program Elements

- *Variable* : stores computation results in the program
  - Data types:
    - Basic types: `int`, `char`, `short`, `long`, `unsigned`, `signed`, `float`, `double`
    - Complex types: arrays, pointers, `struct`, `union`
- *Expression* : arithmetic & logic operation on variables
  - Constants, string literals (a sequence of constant `char` values to represent text strings)
  - Function call
- *Statement* : controls execution order of expressions
- *Function* : contains variables, expressions and statements to build the program
  - Parameter variables: initialized at the function call expression (before function begins)
  - Return type: returned value type after function ends
  - Function body: specifies local variables and statements inside the function

- *Preprocessor* : modifies the C source before program compilation
  - Starts with '`#`' : `#include, #define, #if, #ifdef, #elif, #else, #endif`
- *Comments* : does not have any effect on the program but very useful in making the program readable
  - Character string enclosed by `/* ... */` are comments and replaced with a <space> character before compilation: Ex:`/* this is a comment */`
  - `// ...` is also accepted as comments until end-of-line (originally defined on C++ but became popular also in C compilers)

“sum_up”

```
#include <stdio.h>
int sum_up(int n)/* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main() /* more strict declaration would be: void main(void) */
{
      int c = sum_up(100);
      printf("c = %d\n", c);
}
```

```
Output:
c = 5050
```

```
Inside function sum_up:
c = 0;
i = 1 : c = 0 + 1;  (= 1)
i = 2 : c = 1 + 2;  (= 3)
i = 3 : c = 3 + 3;  (= 6)
  ...
  ...
i = 100 : c = 4050 + 100; (= 5050)
```

“sum_up”

```
#include <stdio.h>
int sum_up(int n)/* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main() /* more strict declaration would be: void main(void) */
{
      int c = sum_up(100);
      printf("c = %d\n", c);
}
```

Defined functions

- `int sum_up(int n)` :
  - Function name: `sum_up`
  - Function parameter: `(int n)` (one parameter of `int` type)
  - Function return type: `int`
- `void main()`:
  - Function name: `main`
  - Function parameter: `()` (no parameter: formally denoted as `(void)`)
  - Function return type: `void` (no return value)

“sum_up”

```
#include <stdio.h>
int sum_up(int n)/* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main() /* more strict declaration would be: void main(void) */
{
      int c = sum_up(100);
      printf("c = %d\n", c);
}
```

Declared variables

- `int i, c = 0;` :
  - Declared variable type: `int` (32-bit signed integer)
  - Declared variable names: `i, c`
  - Variable initialization: `c = 0` (`c` is initialized to `0`)
- `int c = sum_up(100);`
  - Declared variable type: `int` (32-bit signed integer)
  - Declared variable names: `c`
  - Variable initialization: `c = sum_up(100)` (`c` is initialized to the return value of call `sum_up(100)`)

Function Names and Variable Names (Identifiers)

- Function names and variable names are introduced into the program as "*identifiers*"
- <*identifier*> consists of:
  - <*non-digit*> character : `_ a b c ... y z A B C ... Y Z`
  - <*digit*> character : `0 1 2 3 4 5 6 7 8 9`
- <*identifier*> must not start with <*digit*> character
- <*identifier*> must NOT be identical to C keywords (reserved in C language):
  - `auto, double, int, struct, break, else, long, switch, case, enum, register, typedef, char, extern, return, union, const, float, short unsigned, continue, for, signed, void, default, goto, sizeof, volatile, do, if, static, while`
- <*identifier*> are "case-sensitive" (distinguishes between upper-case letters and lower-case letters)
  - "`ii`" and "`II`" are different identifiers (so "`Auto`" is not a C keyword)
- Invalid identifiers:
  - `0_var, 1i, 000` (these start with <*digit*>)
  - `auto, float` (these are C keywords)
- Valid identifiers:
  - `_0var, Auto, I, i, c, integer, floating`

“sum_up”

```
#include <stdio.h>
int sum_up(int n)/* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main() /* more strict declaration would be: void main(void) */
{
      int c = sum_up(100);
      printf("c = %d\n", c);
}
```

Expressions

- `i = 1` : assign `i` to `1`
- `i <= n` : evaluate “`i` is less or equal to `n`”
- `i = i + 1` : evaluate “`i` plus `1`” and assign this value to `i`
- `c = c + i` : evaluate “`c` plus `i`” and assign this value to `c`
- `sum_up(100)` : call function `sum_up`
- `printf("c = %d\n", c)` : call function `printf`

“sum_up”

```
#include <stdio.h>
int sum_up(int n) /* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main() /* more strict declaration would be: void main(void) */
{
      int c = sum_up(100);
      printf("c = %d\n", c);
}
```

Function calls (special type of expressions)

- `sum_up(100)`
  - Called function: `sum_up`
  - Call arguments:
    - 1st argument : 100 (copied to the 1st function parameter `int n`)
- `printf("c = %d\n", c)`
  - Called function: `printf` (this is declared in the header file “stdio.h”)
  - Call arguments:
    - 1st argument : `"c = %d\n"` (copied to the 1st function parameter of `printf`)
    - 2nd argument : `c` (copied to the 2nd function parameter of `printf`)
  - `printf` is declared as: `int printf(const char * format, ...);`
    → *We will learn what this means later*

“sum_up”

```
#include <stdio.h>
int sum_up(int n)/* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main() /* more strict declaration would be: void main(void) */
{
      int c = sum_up(100);
      printf("c = %d\n", c);
}
```

<u>Statements</u>

- `for(i = 1; i <= n; i = i + 1)`
  - Statement type: iteration statement (“for-loop”: iterates statement `c = c + i;`)
    1. `i = 1;` : initial expression before starting the loop
    2. `i <= n;` : condition expression (checks if `i` is less or equal to `n`)
       - Loop iteration continues as long as this condition is TRUE
    3. `i = i + 1;` : increment expression
       - executed after each iteration (after `c = c + i;`)
- `c = c + i;`
  - Statement type: expression statement (evaluates `c = c + i`)
- `return c;`
  - Statement type: return statement (returns the value of `c` back to the caller)

C Program Elements

- *Variable* : stores computation results in the program
  - Data types:
    - Basic types: `int, char, short, long, unsigned, signed, float, double`
    - Complex types: arrays, pointers, `struct`, `union`
- *Expression* : arithmetic & logic operation on variables
  - Constants, string literals (a sequence of constant `char` values to represent text strings)
  - Function call
- *Statement* : controls execution order of expressions
- *Function* : contains variables, expressions and statements to build the program
  - Parameter variables: initialized at the function call expression (before function begins)
  - Return type: returned value type after function ends
  - Function body: specifies local variables and statements inside the function

- *Preprocessor* : modifies the C source before program compilation
  - Starts with '`#`' : `#include, #define, #if, #ifdef, #elif, #else, #endif`
- *Comments* : does not have any effect on the program but very useful in making the program readable
  - Character string enclosed by `/* ... */` are comments and replaced with a <space> character before compilation: Ex:`/* this is a comment */`
  - `// ...` is also accepted as comments until end-of-line (originally defined on C++ but became popular also in C compilers)

Data Types

- Data manipulated in the C program are categorized into “data types”
- Basic types
  - Integral types (integer types): 1-byte, 2-byte, 4-byte data
    - Signed or unsigned
  - Floating types (floating point types): 4-byte (single precision) or 8-byte (double precision)
- Complex types
  - Arrays: a group of identical type data allocated in contiguous memory space
  - Structures: a composition of data types
  - Pointers: an integer type variable whose value represent an address of another variable

Integer Types

- Word type: **`char`** (1-byte), **`short`** (2-byte), **`int`** (4-byte), **`long`** (4-byte)
  - 1 byte = 8 bits
  - Technically, “**`int`**” is the word size of the particular machine (for 16-bit machine, “**`int`**” is 2-byte)
- Sign type: **`unsigned`**, **`signed`**
  - If omitted, intepreted as **`signed`**
  - Signed integer uses 2’s complement: $\boxed{val = -2^{N-1} B_{N-1} + \sum_{i=0}^{N-2} 2^i B_i}$
- Equivalent types
  - **`char`** = **`signed char`**
  - **`short`** = **`short int`** = **`signed short`**
  - **`int`** = **`signed int`** = **`signed`**
  - **`long`** = **`signed long`**
  - **`unsigned short`** = **`unsigned short int`**
  - **`unsigned int`** = **`unsigned`**
  - → Order of “word type” “sign type” can be arbitrary

Integer Constants

- Decimal constants
  - Digits: `0 1 2 3 4 5 6 7 8 9`
  - Highest digit: must not be `0`
  - Ex: `12, 64`
- Octal constants
  - Digits: `0 1 2 3 4 5 6 7`
  - Highest digit: must be `0`
  - Ex: `12 = 014, 64 = 0100`
- Hexadecimal constants
  - Digits: `0 1 2 3 4 5 6 7 8 9 A a B b C c D d E e F f`
  - Starts with `0X` or `0x`
  - Ex: `12 = 0xC, 64 = 0x40`
- Integer suffix (optional, attached at end of constants)
  - `U, u`: unsigned
  - `L, l`: long
  - Ex: `12U` (unsigned int), `12UL` (unsigned long)

Character Constants

- Character constant format: `'`*<character-data>*`'`
- *<character-data>* : one of below
  - *<c-char>*
  - *<simple-escape-sequence>*
  - *<octal-escape-sequence>*
  - *<hexadecimal-escape-sequence>*
- *<c-char>* : one of below (source character set)
  - alphabet characters (upper-case, lower-case): **`a b c . . . y z A B C . . . Y Z`**
  - Operator characters: **`_ ! # $ % & ( ) = ^ ~ | [ ] { } @ < > + - * / ? . , ; :`**
  - Double quotation character: `"`
- *<simple-escape-sequence>* : one of below
  - **`\n`** (new-line), **`\t`** (tab), **`\r`** (carriage-return), **`\b`** (back-space), **`\v`** (vertical tab), **`\a`** (alert-bell), **`\f`** (form-feed), `\'` (character `'`), `\"` (character `"`), `\\` (character `\`), `\?` (character `?`)
- *<octal-escape-sequence>* :
  - `\` *<1 to 3 octal-digits>* : character constant value in octal
- *<hexadecimal-escape-sequence>* :
  - **`\x`** *<1 or more hexadecimal-digits>*: character constant value in hexadecimal

Strings and String Literals

- Strings are sequence of char data where `'\0'` (character constant whose value is `0`) is attached at the end of char sequence
  - Ex. `"hello world!"` (total 12 bytes) requires 13 bytes of memory space → `'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!', '\0'`
- String literals are sequence of character constants
- String literal format: `"`*<s-char-sequence>*`"`
- *<s-char-sequence>* : a sequence of below character data

  <s-char>

  <simple-escape-sequence>

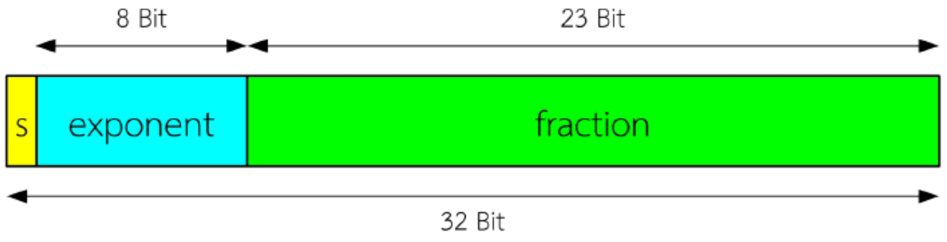
  <octal-escape-sequence>

  <hexadecimal-escape-sequence>
- *<s-char>* (source character set)
  - Same as *<c-char>* set except for *double-quotation-character* (`"`) and *single-quotation-character* (`'`):
    - Double-quotation excluded from *<s-char>* set but included in *<c-char>* set
    - Single-quotation included in *<s-char>* set but excluded from *<c-char>* set

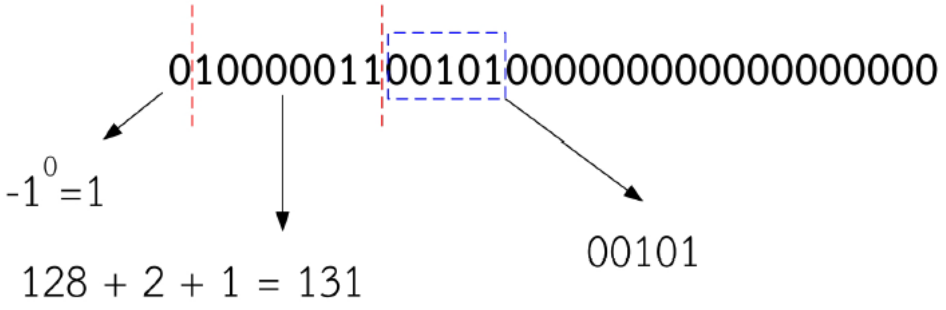
Floating Point Types

- **`float`** (32-bit, 4-byte)
  - sign (S : 1-bit) + exponent (E : 8-bit) + fraction (F : 23-bit) → *all "unsigned" integer values*
  - Exponent value is biased by –127 (2^8 –1)
  - Normalized numbers (common case)
    - Value magnitude range: [2^{-126} , 2^{127}] ("0.0" is NOT included here)
    - Most significant bit of fraction part is always '1' (so can be omitted)
  - normalized_num = $(-1)^S * 2^{(E-127)} * (1 + F * 2^{-23})$
    - ex.
      1.00 → (0 01111111 00000000000000000000000)
      1.50 → (0 01111111 10000000000000000000000)
      2.25 → (0 10000000 00100000000000000000000)
- **`double`** (64-bit, 8-byte)
  - sign (S : 1-bit) + exponent (E : 11-bit) + fraction (F : 52-bit)
  - normalized_num = $(-1)^S * 2^{(E-1023)} * (1 + F * 2^{-52})$
- **`long double`** : equivalent to **`double`** in most cases

IEEE Standard Floating Point Arithmetic (32-bit)



$$\boxed{N = -1^{s} \times 1.\text{fraction} \times 2^{(\text{exponent} - 127)},\ 1 \le \text{exponent} \le 254}$$



$$\boxed{N = 1 \times 1.00101 \times 2^{(131 - 127)}}$$

2^{0} 2^{-1} 2^{-3} 2^{-5}

$$1.00101 = 1\text{x}2^{0} + 1\text{x}2^{-3} + 1\text{x}2^{-5} = 1 + 1/8 + 1/32 = 1.15625$$

$$\boxed{N = 1.15625 \times 2^{4} = 18.5}$$

การแปลงกลับจากเลขทศนิยมเป็นค่า IEEE Standard Floating Point Arithmetic (32-Bit) ตัวอย่างเช่น 74.5

$$74.5 = 74 + 0.5 = 1001010.1_2$$

$$2^6 \quad 2^3 \quad 2^1 \quad 2^{-1}$$

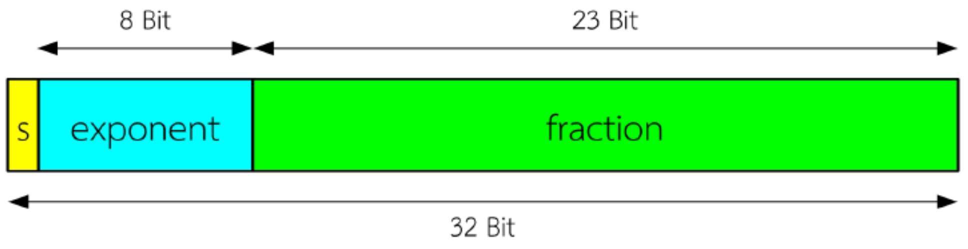
$$1.0010101_2 \ \times 2^6$$

$$2^0 \quad 2^{-3} \quad 2^{-5} \quad 2^{-7}$$

$$-1^{0} \times 1.0010101_2 \ \times 2^{(133-127)}$$

$$\boxed{N = -1^{s} \times 1.\text{fraction} \times 2^{(\text{exponent} - 127)},\ 1 \le \text{exponent} \le 254}$$

01000010100101010000000000000000



Floating Point Types

- Cases outside normalized representation range
  - Denormalized number : very small numbers (< 2^{-126})
    - E = 0 : represents 2^{-126} (not 2^{-127} !)
    - Most significant bit of fraction part is always ‘0’
    - denormalized_num = $(-1)^S * 2^{-126} * (F * 2^{-23})$
  - Zero (two types!)
    - E = 0, F = 0
    - S = 0 $\rightarrow$ +0
    - S = 1 $\rightarrow$ –0
  - “Inf” (infinity)
    - E = 255, F = 0
    - S = 0 $\rightarrow +\infty$
    - S = 1 $\rightarrow -\infty$
  - “NaN” (not-a-number)
    - E = 255, F $\neq$ 0
    - x / 0.0 is “NaN” (divided by 0)
    - if x is “NaN”, then (x == x) = false

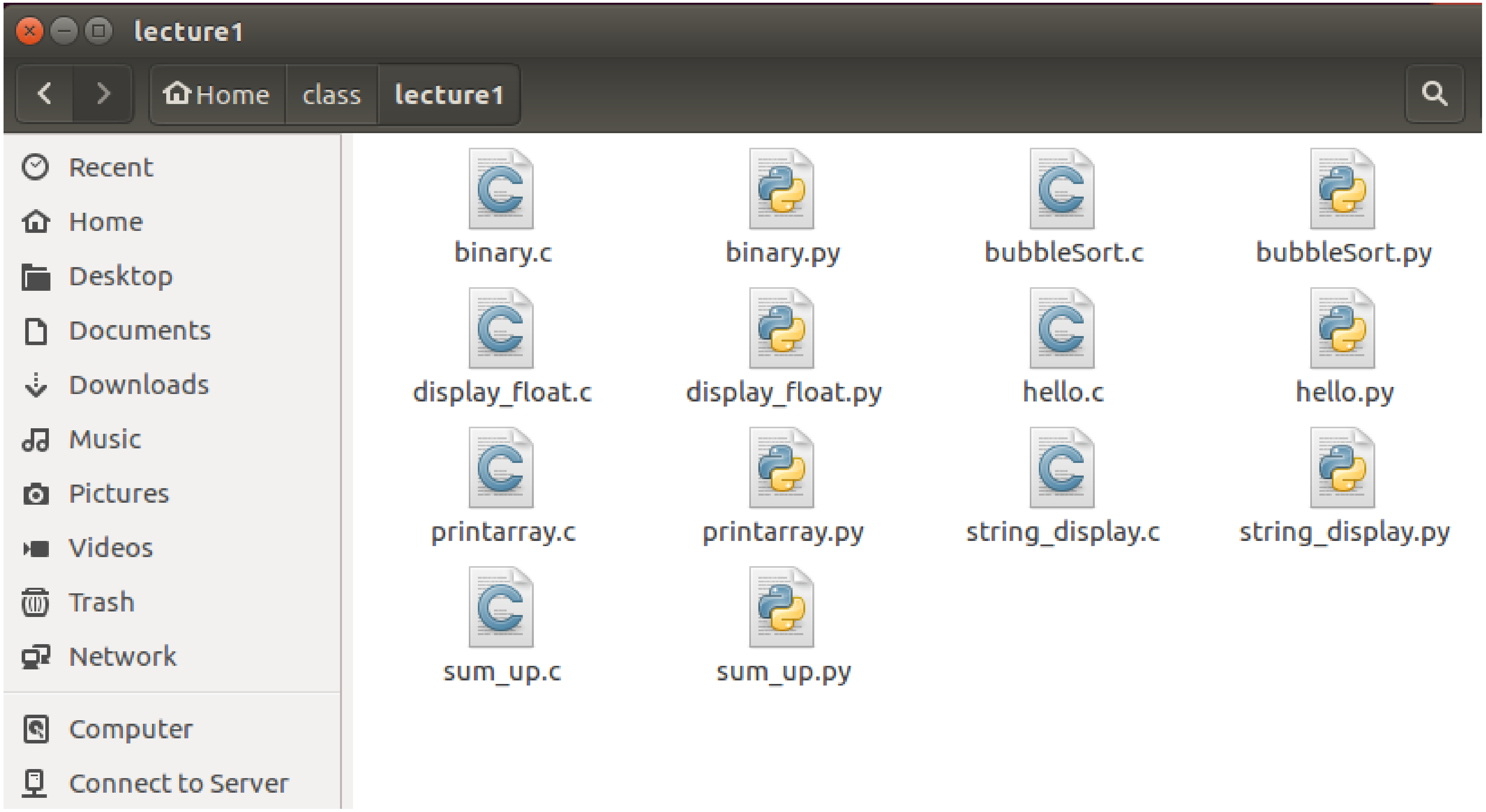
Floating Point Constants

- *<floating-point-constant>* : Floating point constant format
  - *<fraction-part> <exponent-part> <floating-suffix>*
- *<fraction-part>* : one of below format
  - *<digit-sequence> . <digit-sequence>*
  - *<digit-sequence> .*
  - *. <digit-sequence>*
  - *<digit-sequence>*
- *<digit-sequence>* : sequence of **`0, 1, ..., 9`**
- *<exponent-part>* : one of below format
  (can be omitted if *<fraction-part>* includes decimal point ".")
  - **`E`** *<sign> <digit-sequence>*
  - **`E`** *<digit-sequence>*
  - **`e`** *<sign> <digit-sequence>*
  - **`e`** *<digit-sequence>*
- *<sign>* : **`+ -`**
- *<floating-suffix>* : one of below format
  (if omitted, assumed "**`double`**" constant)
  - **`F, f`**: float type
  - **`L, l`**: long double type
- Ex.: **`0.75 = .75 = .0075e2 = 0.075e1 = 0.075E+1 = 75e-2`**

Variable Declaration of Basic Types

- Each *variable* must first be "declared" before being used in the program
- Declaration syntax of basic type variables (more "formal" definition of declaration syntax is given in the next slide):

  <type-specifiers> <identifier> `;`
- *<type-specifiers>* : variable types
  - `int, char, short, long` : integer data types
  - `signed, unsigned` : sign types
  - `float`, `double` : floating-point types
- *<identifier>* : variable name
  - Identifier consists of: upper-case and lower-case alphabets, '_' (underscore character), digit character
  - First character of identifier must not be a digit character
- Declaration terminates with semi-colon '`;`'
- Multiple variables sharing the same data-type can be declared using commas "`,`" separating these names
  - Ex.: `unsigned int a, b, c;`
- Declarations can have "initializer" expression
  - Ex.: `int a = 0; int b = a + c;`



binary.c
binary.py
bubbleSort.c
bubbleSort.py
display_float.c
display_float.py
hello.c
hello.py
printarray.c
printarray.py
string_display.c
string_display.py
sum_up.c
sum_up.py

Displaying Binary Format

```
#include <stdio.h>
void print_bits(int v)    /* print binary format of int v */
{
        int i;
        for(i = 31; i >= 0; i --){
            if(v & (1 << i)) printf("1");
            else printf("0");
        }
        printf("\n");
}
```

```
 100 = 00000000000000000000000001100100
-100 = 11111111111111111111111110011100
```

- `for(i = 31; i >= 0; i --){ ... }`
  - Code block inside `{ ... }` will iterate for 32 times (i = 31, 30, …, 1, 0)
  - `i --` is same as `i = i - 1`
- `if(v & (1 << i)) printf("1"); else printf("0");`
  - `(1 << i)` : shift 1 by `i` bits to left (= 2^i)
  - `(v & (1 << i))` : bit-wise AND operation of `v` and `(1 << i)`
  - → if `i`th bit of `v` is `1` then print “`1`”
  - → if `i`th bit of `v` is `0` then print “`0`”

Displaying Binary Format

```
#include <stdio.h>
void print_bits(int v)    /* print binary format of int v */
{
      int i;
      for(i = 31; i >= 0; i --){
         if(v & (1 << i)) printf("1");
         else printf("0");
      }
      printf("\n");
}
void main()
{
      int c = 100;
      printf("%5d = ", c);
      print_bits(c);
      c = -100;
      printf("%5d = ", c);
      print_bits(c);
}
```

```
  100 = 00000000000000000000000001100100
 -100 = 11111111111111111111111110011100
```

Displaying Floating Format

```
#include <stdio.h>
void print_float(float f)     /* print binary format of float f */
{
        int i;
        int v = *(int *)(&f);
        for(i = 31; i >= 0; i --){
            if(v & (1 << i)) printf("1");
            else printf("0");
            if(i == 31 || i == 23) printf(" ");
        }
        printf("\n");
}
void main()
{
        float f = 1.0;
        printf("%f = ", f);
        print_float(f);
        f = 1.5;
        printf("%f = ", f);
        print_float(f);
        f = 2.25;
        printf("%f = ", f);
        print_float(f);
}
```

This is a trick you can do on C language: `float f` is treated as an `int` value (we will learn what this means later when we learn about "pointers")

```
1.000000 = 0 01111111 00000000000000000000000
1.500000 = 0 01111111 10000000000000000000000
2.250000 = 0 10000000 00100000000000000000000
```

Displaying Floating Format

```c
#include <stdio.h>
void print_float(float f)     /* print binary format of float f */
{
        int i;
        int v = *(int *)(&f);
        for(i = 31; i >= 0; i --){
            if(v & (1 << i)) printf("1");
            else printf("0");
            if(i == 31 || i == 23) printf
        }
        printf("\n");
}
void main()
{
        int a;
        float f = 1.5;
        printf("%f = ", f);
        print_float(f);
        for(a = 0; a < 10; a ++){
            f = f * f;
            printf("%e = ", f);
            print_float(f);
        }
}
```

```
   1.500000 = 0 01111111 10000000000000000000000
2.250000e+000 = 0 10000000 00100000000000000000000
5.062500e+000 = 0 10000001 01000100000000000000000
2.562891e+001 = 0 10000011 10011010000100000000000
6.568408e+002 = 0 10001000 01001000011010111010000
4.314399e+005 = 0 10010001 10100101010100111111100
1.861404e+011 = 0 10100100 01011010101101101010010
3.464823e+022 = 0 11001001 11010101100100100011011
1.#INF00e+000 = 0 11111111 00000000000000000000000
1.#INF00e+000 = 0 11111111 00000000000000000000000
1.#INF00e+000 = 0 11111111 00000000000000000000000
```

Even floating point numbers can overflow fairly quickly (if you do this)

Strings and Characters

```
#include <stdio.h>
#include <string.h>
void main()
{
        char text[12];
        int i;
        strcpy(text, "hello world");
        printf("text = %s\n", text);
        for(i = 0; i < 12; i ++){
            printf("text[%2d] = '%c' (val = %d)\n",
                        i, text[i], text[i]);
        }
        strcpy(text, "world!");
        printf("text = %s\n", text);
        for(i = 0; i < 12; i ++){
            printf("text[%2d] = '%c' (val = %d)\n",
                        i, text[i], text[i]);
        }
}
```

`strcpy(text, "hello world");` copies the string “hello world” to text

```
text = hello world
text[ 0] = 'h' (val = 104)
text[ 1] = 'e' (val = 101)
text[ 2] = 'l' (val = 108)
text[ 3] = 'l' (val = 108)
text[ 4] = 'o' (val = 111)
text[ 5] = ' ' (val = 32)
text[ 6] = 'w' (val = 119)
text[ 7] = 'o' (val = 111)
text[ 8] = 'r' (val = 114)
text[ 9] = 'l' (val = 108)
text[10] = 'd' (val = 100)
text[11] = ' ' (val = 0)
text = world!
text[ 0] = 'w' (val = 119)
text[ 1] = 'o' (val = 111)
text[ 2] = 'r' (val = 114)
text[ 3] = 'l' (val = 108)
text[ 4] = 'd' (val = 100)
text[ 5] = '!' (val = 33)
text[ 6] = ' ' (val = 0)
text[ 7] = 'o' (val = 111)
text[ 8] = 'r' (val = 114)
text[ 9] = 'l' (val = 108)
text[10] = 'd' (val = 100)
text[11] = ' ' (val = 0)
```

C Program Elements

- *Variable* : stores computation results in the program .
  - Data types:
    - Basic types: `int, char, short, long, unsigned, signed, float, double`
    - Complex types: arrays, pointers, `struct, union`
- *Expression* : arithmetic & logic operation on variables
  - Constants, string literals (a sequence of constant `char` values to represent text strings)
  - Function call
- *Statement* : controls execution order of expressions
- *Function* : contains variables, expressions and statements to build the program
  - Parameter variables: initialized at the function call expression (before function begins)
  - Return type: returned value type after function ends
  - Function body: specifies local variables and statements inside the function

- *Preprocessor* : modifies the C source before program compilation
  - Starts with '`#`' : `#include, #define, #if, #ifdef, #elif, #else, #endif`
- *Comments* : does not have any effect on the program but very useful in making the program readable
  - Character string enclosed by `/* ... */` are comments and replaced with a <space> character before compilation: Ex:`/* this is a comment */`
  - `// ...` is also accepted as comments until end-of-line (originally defined on C++ but became popular also in C compilers)

Declaration Specifier

- Variable declaration of complex data types makes use of "declarators"
  - Variable name is one part of the "declarators"
- *<declaration>* : "formal" declaration syntax
  - *<declaration-specifiers> <init-declarator-list> ;*
- *<declaration-specifiers>* : sequence of below "specifiers" and "qualifiers"
  - *<type-specifiers>*
  - *<storage-class-specifiers>*
  - *<type-qualifier>*
- *<type-specifiers>* : one of below (variable types)
  - **`void, int, char, short, long, signed, unsigned, float, double`**: basic types
  - *<struct-or-union-specifier>, <enum-specifier>, <typedef-name>* : complex types
- *<storage-class-specifiers>* : specifies variable storage class
  - **`auto, register`** : local lifetime storage
  - **`static`** : global lifetime internal-level storage
  - **`extern`** : global lifetime external-level storage
  - **`typedef`** : user-defined type
- *<type-qualifier>* : specifies variable attributes
  - **`const`** : read-only, non-modifiable
  - **`volatile`** : value can be changed by other processes → always access to memory
- *<init-declarator-list>* : a list of *init-declarators*
  - *<init-declarator> , <init-declarator-list>* (left-recursive grammar rule)
- *<init-declarator>* :
  - *<declarator>* : uninitialized declaration
  - *<declarator> = <initializer>* : initialized declaration

Declarator

- Declarator introduces *pointer types*, *variable names*, *array types* and *function types* → Functions can be considered as variables with special attributes (parameter list, memory allocation to function body code)
- *<declarator>* :
  - *<pointer> <direct-declarator>*
  - *<direct-declarator>*
- *<pointer>* : pointer type
  - `*` *<type-qualifier> <pointer>*
  - `*` *<pointer>*
- *<direct-declarator>* :
  - *<identifier>* : variable name
  - `(`*<declarator>*`)` : used in complex declarator
  - *<direct-declarator>* `[`*<constant-expression>*`]` : array type
  - *<direct-declarator>* `(`*<parameter-type-list>*`)` : function type
- Ex:
  - `int * p;` (a pointer to an "int" type variable)
  - `int a[10];` (an array of 10 elements of "int" type)
  - `int * a[10];` (an array of 10 elements of "int *" type)
  - `int f(int a);` (a function with an "int" type parameter that returns "int")
  - `int ** f(void);` (a function with no parameters that returns "int **")

“printArray”

```
#include <stdio.h>
void printArray(int array[], int n) /* print array a of size n */
{
      int j;
      printf(“array[%d] = {”, n);
      for(j = 0; j < n; j ++){
         if(j > 0) printf(“, ”); /* print comma from 2nd iteration */
         printf(“%d”, array[j]);
      }
      printf(“}\n”);
}
void main()
{
      int a[10] = {1,2,3,4,5,6,7,8,9,10};
      printArray(a, 10);
}
```

```
Output:
array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

- `int a[10] = {1,2,3,4,5,6,7,8,9,10};`
  - An array of 10 elements of “int” type data
  - Initialized to a[0] = 1, a[1] = 2, a[3] = 3, …, a[9] = 10

Arrays

- Array is a group of variables allocated in contiguous memory space.
  - `int a[10] = {1,2,3,4,5,6,7,8,9,10};` (an array of 10 elements of type "int". {…} is the array initializer)
- Array subscript operator "`[]`" is used to access an element of an array (subscript index starts from 0)
  - `a[j]` (value of jth element of array `a`)
- Array variable itself represents the address of the head element ("0th" element), and can be used to calculate address of other elements
  - `a + j` (address of jth element of array `a`)
- Indirection operator "`*`" can be used to access the value stored in the indicated address
  - `*(a + j)` (value of jth element of array `a`: equivalent to `a[j]`)
- Multidimensional arrays
  - `int b[10][100];` (an array of 10 arrays of 100 elements of type "int")
  - `b[i][j]` (value of (i, j)th element of 2D-array `b`)
  - `b + i * 100 + j` (address of (i, j)th element of 2D-array `b`)
- Array access : C compiler does not check whether the subscript index is within the range of the array size

  → *Accessing the array elements beyond the array range is one of the major cause of bugs in C programs*

Pointers

- Pointers store the memory address of a variable (variable address can be computed by "address-of" operator `&`)
  - `int aa, a[10];`
  - `int * p0 = &aa;` (pointer `p0` stores the address of `aa`: `p0` points to `aa`)
  - `int * p1 = a;` (pointer `p1` stores the address of the head element of array `a`: equivalent to `p1 = &a[0]`)
- The variable that the pointer points to (pointer-target, pointee) can be accessed from the pointer by "dereferencing"
  - `*p0 = 100;` (dereference `p0` accesses `aa`, `aa` is assigned `100`)
  - `*p1 = 10;` (dereference `p1` accesses `a[0]`, `a[0]` is assigned `10`)
  - `p1 = p1 + 1;` (`p1` stores the address of the next element `a[1]`)
  - `*p1 = 20;` (dereference `p1` accesses `a[1]`, `a[1]` is assigned `20`)
- Pointers and arrays are treated similarly in expressions
  - Array subscript operator "`[]`" and indirection operator "`*`" applies to both arrays and pointers
- Differences between arrays and pointers are:
  - Arrays reserves the memory space for the entire array elements, but pointer reserves the memory space only for storing a single address value (pointer value)
  - Value of array variable (array address) is determined by the compiler (programmer cannot assign an address to an array), but pointer values are always computed explicitly in the program.

Functions

- Each function must first be “declared” or “defined” before being called inside the program
- *<declaration>* : function declaration syntax
  - *<declaration-specifiers> <declarator> ;*
- *<function-definition>* : function definition syntax
  - *<declaration-specifiers> <declarator> <compound-statement>*
- *<compound-statement>* :

```
    {
            <declaration-list>
            <statement-list>
    }
```

  - *<declaration-list>* : list of declarations
  - *<statement-list>* : list of “statements” (specifies sequence of computations)
  - → Declarations cannot appear after statements

    (later versions of C (C99, C++) do not have this order restriction of declarations and statements)
  - → In the function parameter list, parameter names can be omitted. Especially for function declaration, parameter types are the essential information but parameter names are irrelevant.
    - `int f(int);` (a function with “int” return type and a parameter of “int” type)

Functions, Arrays, Pointers

- Function parameters are passed from the "caller" expression to the "callee" function by values : "call-by-values"
  - Parameter variables are first copied to the local storage of the callee function before jumping to the callee function
  - For array-type parameters, the array address is copied (array-type parameters are actually equivalent to pointer-type parameters)
  - Modifications to the parameter values inside the callee function do not have any side-effects at the caller

  → *We will look more closely to how function calls are implemented later on in this lecture*
- There cannot be an "array of functions", and there cannot be a function that returns "array type" → because both are indistinguishable on the C grammar
  - `int f[10](int);` (a function with "int [10]" return type? Or a 10-element array of functions?) → this description is INVALID!!
- Function pointers:

```
/* definition of function "func" */
int func(int a){ ... }

/* inside some function */
int (* fp)(int);
/* fp is a pointer to a function with an int-type parameter that returns int */
int a = 1, b;
fp = func; /* assigns address of func to fp */
b = (*fp)(a); /* b = fp(a); is also OK */
```

  → In fact, function name itself designates the address to that function code in the program

“sort”

```
#include <stdio.h>
void swap(int * a0, int * a1) /* swap *a0 and *a1 */
{
        int t = *a0
        *a0 = *a1;
        *a1 = t;
}
void sort(int a[], int n) /* sort array a of size n */
{
        int i, j;
        for(i = 0; i < n; i = i + 1){
            for(j = 1; j < n - i; j = j + 1){
               if(a[j - 1] > a[j]) swap(&a[j - 1], &a[j]);
            }
        }
}
void main()
{
        int a[10] = {10,5,2,1,7,6,3,4,9,8};
        printArray(a, 10);
        sort(a, 10);
        printArray(a, 10);
}
```

```
Output:
array[10] = {10, 5, 2, 1, 7, 6, 3, 4, 9, 8}
array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Complex Declarators

- Due to the circular grammar definition of declarators, declarators can specify very complex combinations of arrays, pointers, and functions
- *<declarator>* :
  - *<pointer> <direct-declarator>*
  - *<direct-declarator>*
- *<direct-declarator>* :
  - **(***<declarator>***)** : used in complex declarator
- Complex declaration:
  - `int (* ap)[10];` ("ap" is a pointer to an array of 10 "int"-type elements)
  - `int (*(* apfp)(void))[10];` ("apfp" is a pointer to a function with no parameters that returns a pointer to an array of 10 "int"-type elements)

Functions with Variable Number of Parameters

- `int printf(const char * format, ...);`
- `const char * format` : printf format specifier string, consisting of
  - Text strings : displayed directly to the console
  - Escape sequence : character starting with `\` (such as `'\n'`, `'\t'`)
  - Format specifier : starts with `'%'` for formatting scalar data and strings with following fields
    - type : `%d` (decimal integer), `%x` (hexadecimal integer), `%c` (character), `%s` (string), `%f` (floating-point)
    - width : if specified, if the data string is less than the width, space characters are filled in
    - precision : number of fraction part digits (applies only for `"f"` type)
    - Ex:
      - `"%5d"` (width = 5, type = decimal integer)
      - `"%5.1f"` (width = 5, precision = 1, type = floating-point)
      - `"%10s"` (width = 10, type = string)
- `"..."` : variable number of argument values referred in "printf format specifier string"
  - "`int`"-type values : `%d, %x, %c`
  - char-array address: `%s`
  - "`double`"-type values : `%f`

C Program Elements

- *Variable* : stores computation results in the program
  - Data types:
    - Basic types: `int, char, short, long, unsigned, signed, float, double`
    - Complex types: arrays, pointers, `struct, union`
- *Expression* : arithmetic & logic operation on variables
  - Constants, string literals (a sequence of constant `char` values to represent text strings)
  - Function call
- *Statement* : controls execution order of expressions
- *Function* : contains variables, expressions and statements to build the program
  - Parameter variables: initialized at the function call expression (before function begins)
  - Return type: returned value type after function ends
  - Function body: specifies local variables and statements inside the function
- *Preprocessor* : modifies the C source before program compilation
  - Starts with '`#`' : `#include, #define, #if, #ifdef, #elif, #else, #endif`
- *Comments* : does not have any effect on the program but very useful in making the program readable
  - Character string enclosed by `/* ... */` are comments and replaced with a <space> character before compilation: Ex:`/* this is a comment */`
  - `// ...` is also accepted as comments until end-of-line (originally defined on C++ but became popular also in C compilers)

Data Structure

- Data structure is a group of variables allocated in contiguous memory space
  - Array is one kind of data structure where all variables have the same type and its element is accessed by array subscript operator `[ ]`
- "`struct`" are also data structures where any kind of variable types can be combined, and its data "member" is accessed by member selection operator "`.`" (dot operator)
- *<type-specifiers>* : struct/union definition
  - *<struct-or-union-specifier>*
- *<struct-or-union-specifier>* :
  - **`struct`** *<identifier>* `{` *<struct-declaration-list>* `}`
  - **`union`** *<identifier>* `{` *<struct-declaration-list>* `}`
- *<identifier>* : struct (or union) tag name
- *< struct-declaration-list >* : similar to *<declaration-list>*
- Ex:

```
struct vector {
        int x, y;
};
struct vector v0;
v0.x = 10;
v0.y = 20;
```

- "`union`" is similar to "`struct`" but the memory allocation is different
  - Each "`struct`" member has distinct memory allocation
  - All "`union`" members have overlapping memory allocation → their addresses are all the same
  - "`union`" is used when only one member is used at a time → to save memory space

Enumeration

- "enum" is a group of integer constants that are associated with names
  - *<type-specifiers>* : *<enum-specifier>*
- Ex:

```
enum employee_status {
        ES_Trainee,
        ES_Technician,
        ES_Manager,
        ES_SeniorManager,
        ES_DepartmentManager
};
```

- Each element is assigned an distinct integer value starting from 0
  - `ES_Trainee = 0, ES_Technician = 1, ES_Manager = 2,...`
- Integer values can be assigned explicitly:

```
enum employee_status {
        ES_Trainee = 1,
        ES_Technician,
        ES_Manager = 4,
        ES_SeniorManager,
        ES_DepartmentManager = 8
};
```

`ES_Technician` does not have explicit value, so it will be assigned 2 by default

`ES_SeniorManager` does not have explicit value, so it will be assigned 5 by default

User-Type Definition

- “typedef” specifies a user-defined type (one of the <storage-class-specifier>)
- typedef name becomes the synonym for the introduced user-defined type
- Ex:

```
typedef struct vector{
        int x, y;
} VECTOR;
VECTOR v0; /* same as: struct vector v0; */
v0.x = 10;
v0.y = 20;
```

C Program Elements

- *Variable* : stores computation results in the program
  - Data types:
    - Basic types: `int, char, short, long, unsigned, signed, float, double`
    - Complex types: arrays, pointers, `struct, union`
- *Expression* : arithmetic & logic operation on variables
  - Constants, string literals (a sequence of constant `char` values to represent text strings)
  - Function call
- *Statement* : controls execution order of expressions
- *Function* : contains variables, expressions and statements to build the program
  - Parameter variables: initialized at the function call expression (before function begins)
  - Return type: returned value type after function ends
  - Function body: specifies local variables and statements inside the function

- *Preprocessor* : modifies the C source before program compilation
  - Starts with '`#`' : `#include, #define, #if, #ifdef, #elif, #else, #endif`
- *Comments* : does not have any effect on the program but very useful in making the program readable
  - Character string enclosed by `/* ... */` are comments and replaced with a <space> character before compilation: Ex:`/* this is a comment */`
  - `// ...` is also accepted as comments until end-of-line (originally defined on C++ but became popular also in C compilers)

Expressions

- Expressions specify how to manipulate data (operator) on variables and constants (operands), and how to call functions
- Arithmetic operators: `+` (add), `-` (subtract), `*` (multiply), `/` (divide), `%` (remainder, modulo)
  - Ex.: `a + b, a - b, a * b, a / b, a % b` (remainder of `a / b`, a mod b)
  - `%` operator applies to integer types only
- Shift operators: `<<` (shift left), `>>` (shift right)
  - Ex.: `a << b` (shift a to left by b-bits : a * 2^{b})
  - Ex.: `a >> b` (shift a to right by b-bits: a * 2^{-b})
  - Shift operators apply to integer types only
- Bitwise Boolean operators: `&` (and), `|` (or), `^` (exor), `~` (not)
  - Ex.: `unsigned char a = 5, b = 3; (a ← 0101, b ← 0011)`
    `(a & b) → 1, (a | b) → 7, (a ^ b) → 2, (~a) → 10`
  - Bitwise Boolean operators apply to integer types only

Boolean Types And Logical Expression

- Relational operators: `==` (equal), `!=` (not equal), `>` (greater than), `<` (less than), `>=` (greater or equal), `<=` (less or equal)
- Logical operators: `&&` (logical-and), `||` (logical-or), `!` (logical-not)
- No explicit Boolean types in C (use "int" type)
  - Results of relational and logical operators are 0 (false) or 1 (true)
  - Integer values are interpreted as Boolean in logical operators (0 is false, non-zero is true)
- Ex.: `a = 10; b = 0;`
  - `a > b` → `1` (true), `a < b` → `0` (false)
  - `a && b` → `0`, `a && (!b)` → `1`
- "Short-circuit evaluation" of logical expressions
  - Logical expressions are evaluated from left to right
  - If the Boolean value of the 1st operand is sufficient to determine the value of the expression, the evaluation of the 2nd operand is skipped.
  - Ex.: `(a == 0 || b / a)` → If a is 0, the expression is "true" and (b / a) is not evaluated. If a is non-zero, (b / a) is evaluated safely. (If not for the short-circuit evaluation, (b / a) will be evaluated when a == 0 which is "divided-by-zero")

Assignment Expressions

- Assignment operators:
  - Simple assignment: `=`
  - Self-modifying assignment: `+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=`
  - Ex.: `a += b` → `a = a + b`
  - Operators `%=, <<=, >>=, &=, |=, ^=` apply to integer types only
  - Value of modified variable becomes the return value of assignment expression
    - Ex: `a = (b += c);` → c is added to b, and then b is added to a
    - Ex: `a = b = c;` → c is assigned to b, and then b is assigned to a (assignment operator is evaluated from right to left)
- Increment operators:
  - Pre-increment operators: `++ a, -- a`
    - `++ a` → `(a += 1)`
    - `-- a` → `(a -= 1)`
  - Post-increment operators: `a ++, a --`
    - Increment operation performed after the value is read
  - Ex: a = 10;

    b = (++ a) + 2;
    1. a = 10 + 1 = 11
    2. b = 11 + 2 = 13

    b = (a ++) + 2;
    1. b = 10 + 2 = 12
    2. a = 10 + 1 = 11

Arithmetic Conversions

- Arithmetic conversions are automatically applied to expressions when
  - two operands have different types
  - operand size is smaller than size of "`int`" (`char`, `short`)
- Arithmetic conversion rules
  - Floating-point conversion
    - If either operand is `double`, convert both to `double`
    - Otherwise if either operand is `float`, convert both to `float`
  - Integer conversion
    - If either operand is `unsigned long`, convert both to `unsigned long`
    - Otherwise if either operand is `long` and the other is `unsigned int`, convert both to `unsigned long`
    - Otherwise if either operand is `long` , convert both to `long`
    - Otherwise if either operand is `unsigned int`, convert both to `unsigned int`
    - Otherwise convert both to `int`
- Assignment conversion rules
  - Value assignment to a variable will cause the value to convert to the type of the assigned variable
- Ex: `int a; short b; char c; double d; float f;`
    - `a = a + b;` → `b` is converted to `int` before adding to `a`
    - `a = b + f;` → `b` is converted to `float` before adding to `f`, then the result `float` value is converted back to `int` and stored to `a`
    - `d = a / 2.0;` → `a` is converted to `double` before being divided by `2.0`
    - `d = b / a;` → `b` is converted to `int` before being divided by `a` (NOTE THAT this is an integer division!!) and the result is converted to `double` and stored to `d`

Other Expressions

- Function call expression:
  - Format: *<func-address>*`(`*<argument-expression-list>*`)`
  - *<func-address>* : function-name or function-pointer
  - *< argument-expression-list >* : list of argument expressions that will be passed (by value) to the function parameters
- Type-cast expression:
  - Format: `(`*<type-name>*`)` *<expression>*
  - Operation: convert type of *<expression>* to *<type-name>*
  - Ex: `int a = 257; int b = (char) a;`
    - `(char)` operator truncates the value to 8 bits → value of `b` is 1
- Conditional expression:
  - Format: *<expression-1>* `?` *<expression-2>* `:` *<expression-3>*
  - Operation: if *<expression-1>* is true, *<expression-2>* is evaluated and *<expression-3>* is skipped. If *<expression-1>* is false, *<expression-2>* is skipped and *<expression-3>* is evaluated
- Sequential expression:
  - Format: *<expression-1>*`,` *<expression-2>*
  - Operation: *<expression-1>* and *<expression-2>* are evaluated in sequential order. Return value is the value of *<expression-2>*
  - More than two expressions can be sequentialized in the same manner (with commas) where the return value is that of the last expression

NOTE: Conversions on Function Call Arguments

- C language is VERY loose in type checking on function calls → this can cause many problem in the code that is extremely hard to detect
  - To avoid these troubles, make sure that all functions are "properly" declared before being called
- If the function declaration with parameter type list exists before the function call, each call argument will undergo the usual "assignment conversion"
- If the function declaration is "incomplete" (missing parameter type list), each call argument will undergo a special conversion
  - "`signed`" integers (`char, short, int`) → converted to `int`
  - "`unsigned`" integers (`char, short, int`) → converted to `unsigned int`
  - `float` → converted to `double`
  - Above argument conversions also apply to "variable length arguments" (because obviously they don't have any type information)
  - → The real danger here is that the types of the converted argument types and function parameter variable types may not match, and unexpected results may occur (which is very difficult to locate the cause of the problem, because these unwanted argument conversions are not visible on the program)
  - → For example, if the call argument and the corresponding function parameter are both `float`, failing to provide the parameter type list results in the conversion of the argument to `double`, and the compiler won't even recognize this mistake!!
- If the function declaration does not exist before the call, the compiler does not issue an error, but it will:
  - internally declare a function with the call name that returns an `int` type
  - with "missing parameter type list"

C Program Elements

- *Variable* : stores computation results in the program
  - Data types:
    - Basic types: `int, char, short, long, unsigned, signed, float, double`
    - Complex types: arrays, pointers, `struct, union`
- *Expression* : arithmetic & logic operation on variables
  - Constants, string literals (a sequence of constant `char` values to represent text strings)
  - Function call
- *Statement* : controls execution order of expressions
- *Function* : contains variables, expressions and statements to build the program
  - Parameter variables: initialized at the function call expression (before function begins)
  - Return type: returned value type after function ends
  - Function body: specifies local variables and statements inside the function

- *Preprocessor* : modifies the C source before program compilation
  - Starts with '`#`' : `#include, #define, #if, #ifdef, #elif, #else, #endif`
- *Comments* : does not have any effect on the program but very useful in making the program readable
  - Character string enclosed by `/* ... */` are comments and replaced with a <space> character before compilation: Ex:`/* this is a comment */`
  - `// ...` is also accepted as comments until end-of-line (originally defined on C++ but became popular also in C compilers)

Statements

- Statements specify the execution order of expressions
  - Statements also decribe the hierarchical program structure with "compound statements"
- *<statement>* : one of below
  - *<expression-statement>* : expression terminated by '`;`'
  - *<compound-statement>* : a "code block" which can have multiple declarations and multiple statements (function body is a compound statement)
  - *<selection-statement>* : specifies conditional execution of code blocks
    - if-statement, switch-statement
  - *<iteration-statement>* : specifies iterative execution of code blocks
    - for-statement, while-statement, do-while-statement
  - *<jump-statement>* : redirects the program execution to a certain location
    - **`break, continue, return, goto`**
  - *<labeled-statement>* : statement with a "label" that can redirect the program execution
    - target statement of switch (case, default)
    - target statement of goto

Compound Statement

- Format:

```
{
        <declaration-list>
        <statement-list>
}
```

- *<declaration-list>* : declarations of local variables accessible inside the compound statement
- *<statement-list>* : list of statements (can also include compound statements to form a hierarchical statement structure)
- Sometimes called “code block”

If-Else Statement

- Format:
  - `if (<expression>) <statement>`
    - if *<expression>* is true, statement1 is executed
  - `if (<expression>) <statement-1> else <statement-2>`
    - if *<expression>* is true, *<statement-1>* is executed. Otherwise *<statement-2>* is executed.
- *<statement-1>*, *<statement-2>* : typically a simple expression statement, compound statement or another if-else statement (can of course be any other type of statements)
- Dangling "else" problem:

```
if (a)
if (b) S2;
else S3;
```

  - `if (a) { if (b) S2;} else S3;`
  - `if (a) { if (b) S2; else S3;}` → C compiler interprets this way ("else" is associated to the closest "if" statement)

Iteration Statement

- Format:
  - **`for`**`(`*<expression-1>*`;` *<expression-2>*`;` *<expression-3>*`)` *<statement>*
    - *<expression-1>*, *<expression-2>*, *<expression-3>*, can be partially or totally omitted
    - Behavior:
      - *<expression-1>* is executed before entering the loop
      - *<statement>* is iterated as long as *<expression-2>* is true
      - *<expression-3>* is evaluated after each iteration (after *<statement>* execution)
  - **`while`**`(`*<expression>*`)` *<statement>*
    - *<expression>* cannot be omitted
    - Behavior:
      - *<statement>* is iterated as long as *<expression>* is true
  - **`do`** *<statement>* **`while`**`(`*<expression>*`);`
    - *<expression>* cannot be omitted
    - Behavior:
      - *<statement>* is iterated at least once, and as long as *<expression>* is true

Switch Statement

- Format:

```
switch(<expression>)
{
    case <const-int-value-1>:
         <statement-1>
        break;
    case <const-int-value-2>:
         <statement-2>
        break;
    default:
         <statement-3>
        break;
}
```

- case-statement:
  - Statements “labeled” with integer constants (*<const-int-value-1>*, *<const-int-value-2>* : constant integer values)
  - Program execution jumps to the case-statement with the corresponding integer value of (*<expression>*). If no such case-statement exist, it jumps to the default-statement (if it exists)
  - **break** statement is needed to stop the execution of the case-statement (if **break** does not exist, the execution will continue to the next case-statement)

Jump Statement

- `break;`
  - Inside iteration statement: exits loop
  - Inside switch statement: exits switch statement body
- `continue;`
  - Inside iteration statement: skips all succeeding statements in the loop and jumps back to loop entrance
- `return;`
  - Jumps to end of function
- `return` *<expression>*`;`
  - *<expression>* becomes the return value of the function
- *Labeled-statement*:
  <identifier> : <statement>
- `goto` *<identifier>*`;`
  - Jumps to the labeled-statement with *<identifier>*

“main” function arguments

- “main” function prototypes
  - `void main()`
  - `int main()`
  - `void main(int argc, char * argv[])`
  - `int main(int argc, char * argv[])`
    - `int argc` : # of command arguments
    - `char * argv[]` : array of command strings
    - Use command arguments to pass parameters to main() such as array size, sort algorithm type, random seed number.
  - Return value of “main” function: used to signal the program execution status to the OS
- Ex:
  - `C:\mywork> a.exe 300 0 1`
  - argc = 4
  - argv[0] = “a.exe” (executable file name created by C compiler)
  - argv[1] = “300”
  - argv[2] = “0”
  - argv[3] = “1”

“sum_up”

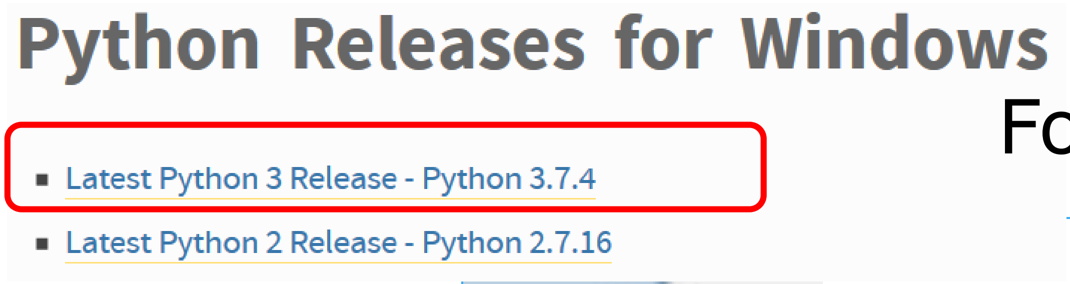
```
#include <stdio.h>
#include <stdlib.h>
int sum_up(int n)   /* sum up to n */
{
      int i, c = 0; /* declares "int" data i, c (c is initialized to 0) */
      for(i = 1; i <= n; i = i + 1) /* i = 1, 2, ..., n */
         c = c + i;
      return c;
}
void main(int argc, char * argv[])
{
      int c;
      if(argc != 2){
         printf("Invalid commands!\nusage: %s <number to sum up>\n", argv[0]);
         return;
      }
      c = sum_up(atoi(argv[1]));
      printf("c = %d\n", c);
}
```

`atoi(argv[1])`: converts string to `int` value

```
> a.exe 10
c = 45
> a.exe 100
c = 4950
> a.exe 10000
c = 49995000
```

Python Programing

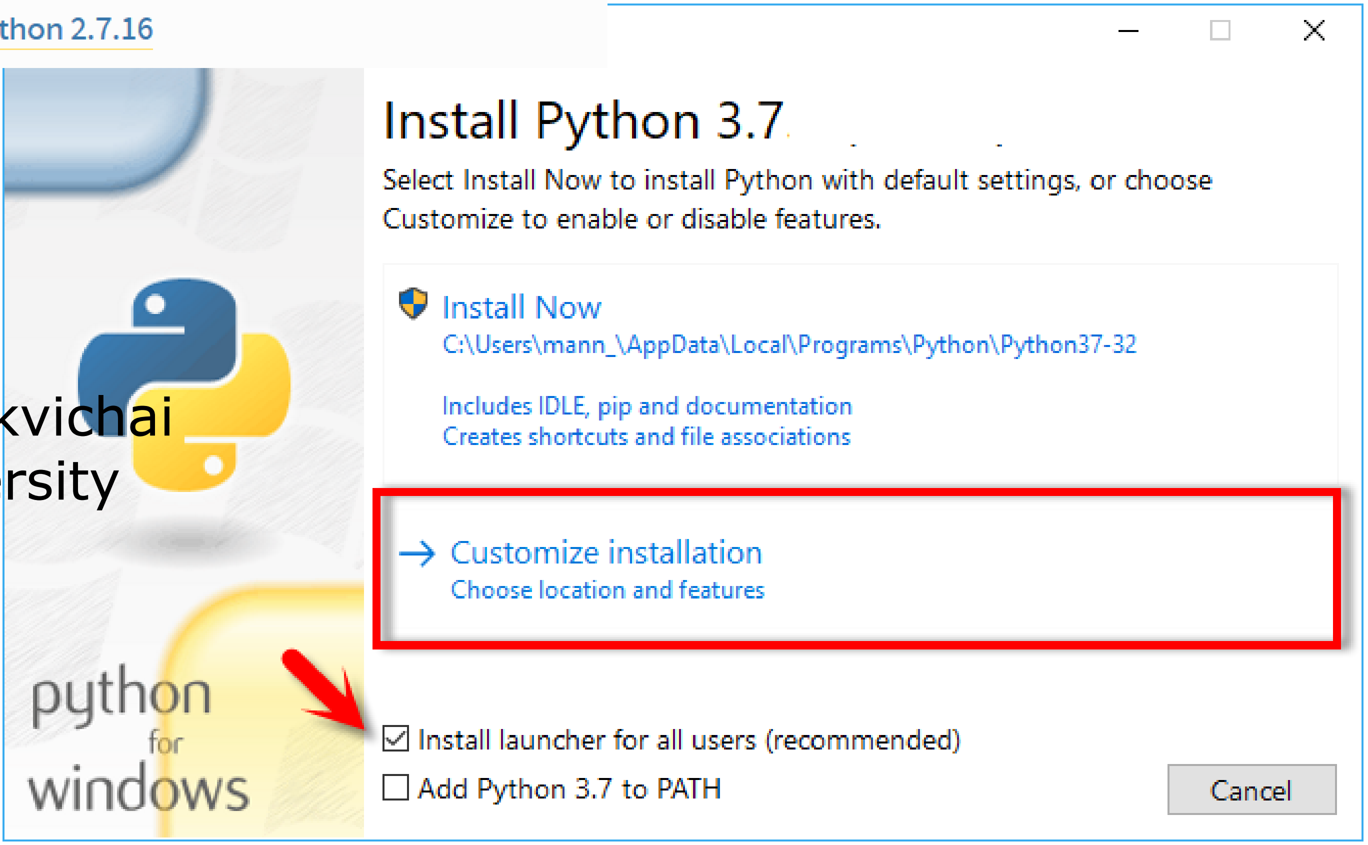
https://www.python.org/downloads/windows/



Python Releases for Windows

- Latest Python 3 Release - Python 3.7.4
- Latest Python 2 Release - Python 2.7.16

For Ubuntu do nothing



Install Python 3.7

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\mann_\AppData\Local\Programs\Python\Python37-32

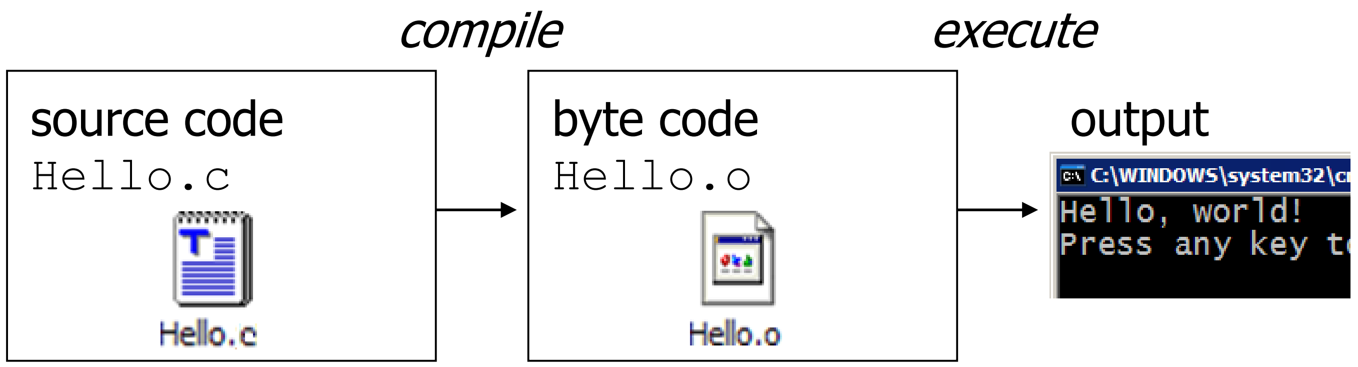
Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.7 to PATH

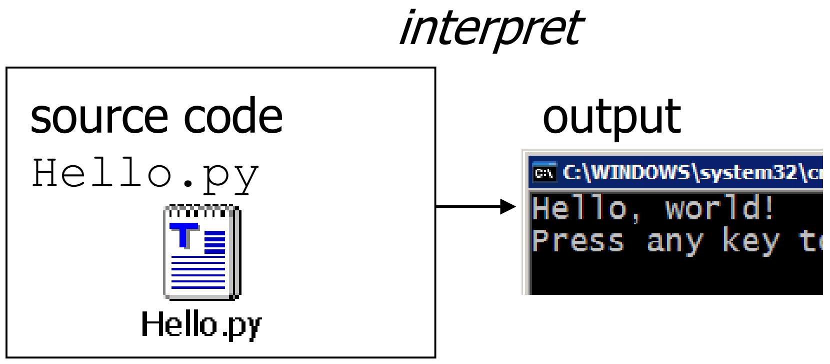
By
Kanjanapan Sukvichai
Kasetsart University

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



```
Hello, world!
Press any key t
```

- Python is instead directly *interpreted* into machine instructions.



```
Hello, world!
Press any key t
```

- **expression**: A data value or set of operations to compute a value.

  Examples: `1 + 4 * 3`

  `42`

- Arithmetic operators we will use:

`+ - * /`	addition, subtraction/negation, multiplication, division
`%`	modulus, a.k.a. remainder
`**`	exponentiation

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

    `1 + 3 * 4` is `13`

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is `16`

- When we divide integers with `/` , the quotient is also an integer.

```
      ___3                 _____52
    4 ) 14              27 ) 1425
        12                   135
         2                     75
                               54
                               21
```

  - More examples:
    - `35 / 5` is `7`
    - `84 / 10` is `8`
    - `156 / 100` is `1`

- The `%` operator computes the remainder from a division of integers.

```
      ___3                   ___43
    4 ) 14                 5 ) 218
        12                     20
         2                      18
                                15
                                 3
```

- Python has useful commands for performing calculations.

Command name	Description
`abs(`***value***`)`	absolute value
`ceil(`***value***`)`	rounds up
`cos(`***value***`)`	cosine, in radians
`floor(`***value***`)`	rounds down
`log(`***value***`)`	logarithm, base *e*
`log10(`***value***`)`	logarithm, base 10
`max(`***value1***`, `***value2***`)`	larger of two values
`min(`***value1***`, `***value2***`)`	smaller of two values
`round(`***value***`)`	nearest whole number
`sin(`***value***`)`	sine, in radians
`sqrt(`***value***`)`	square root

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

- To use many of these commands, you must write the following at the top of your Python program:

```
from math import *  or import math
```

- `print` : Produces text output on the console.
- Syntax:

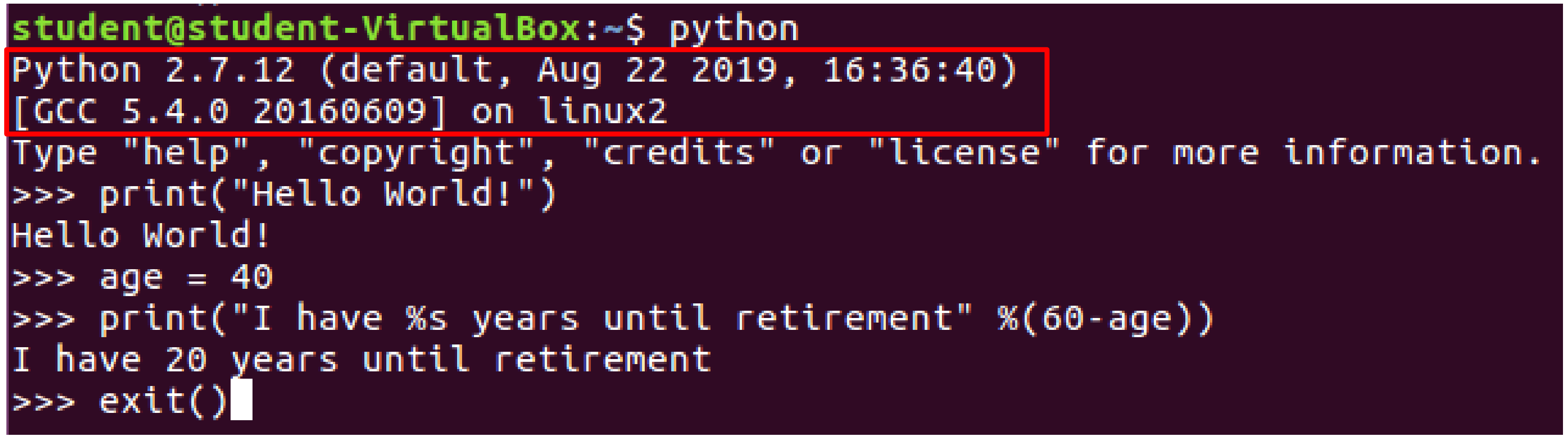
  print("***Message***")

  ```
  print(" Message %type ‶ %(variable))
  ```
- Examples:

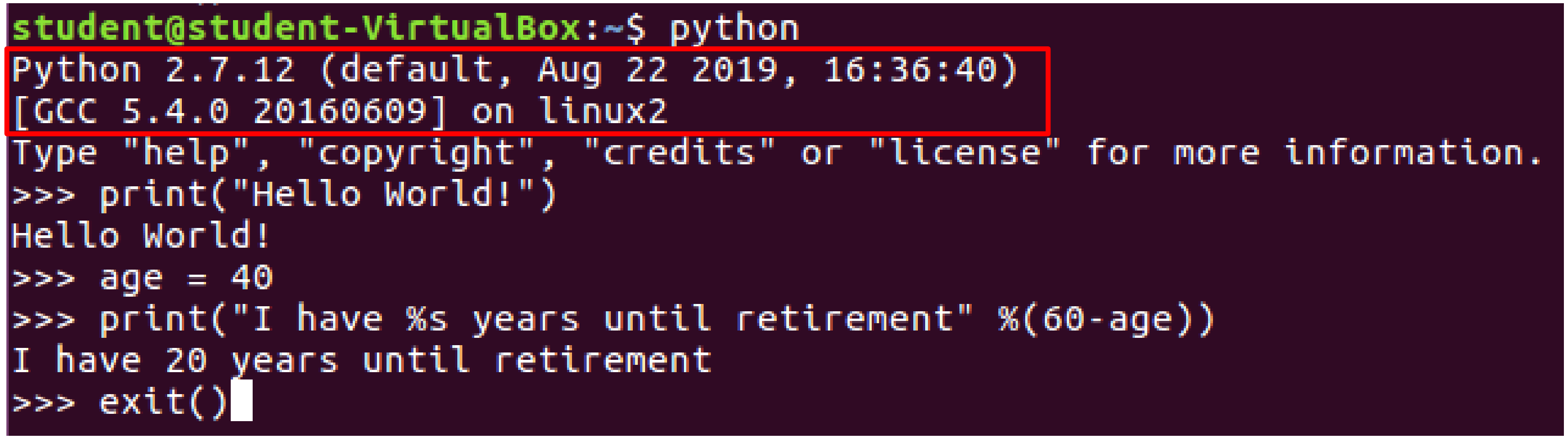
  ```
  print("Hello, world!")
  age = 40
  print("I have %s years until retirement" %(60-age))
  ```

  Output:

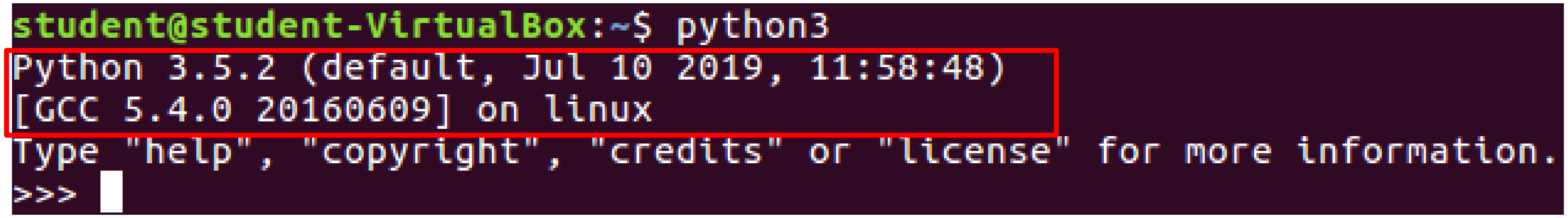
  ```
  Hello, world!
  I have 20 years until retirement
  ```



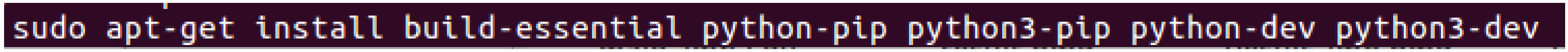
```
student@student-VirtualBox:~$ python
Python 2.7.12 (default, Aug 22 2019, 16:36:40)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> age = 40
>>> print("I have %s years until retirement" %(60-age))
I have 20 years until retirement
>>> exit()
```



```
student@student-VirtualBox:~$ python
Python 2.7.12 (default, Aug 22 2019, 16:36:40)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> age = 40
>>> print("I have %s years until retirement" %(60-age))
I have 20 years until retirement
>>> exit()
```



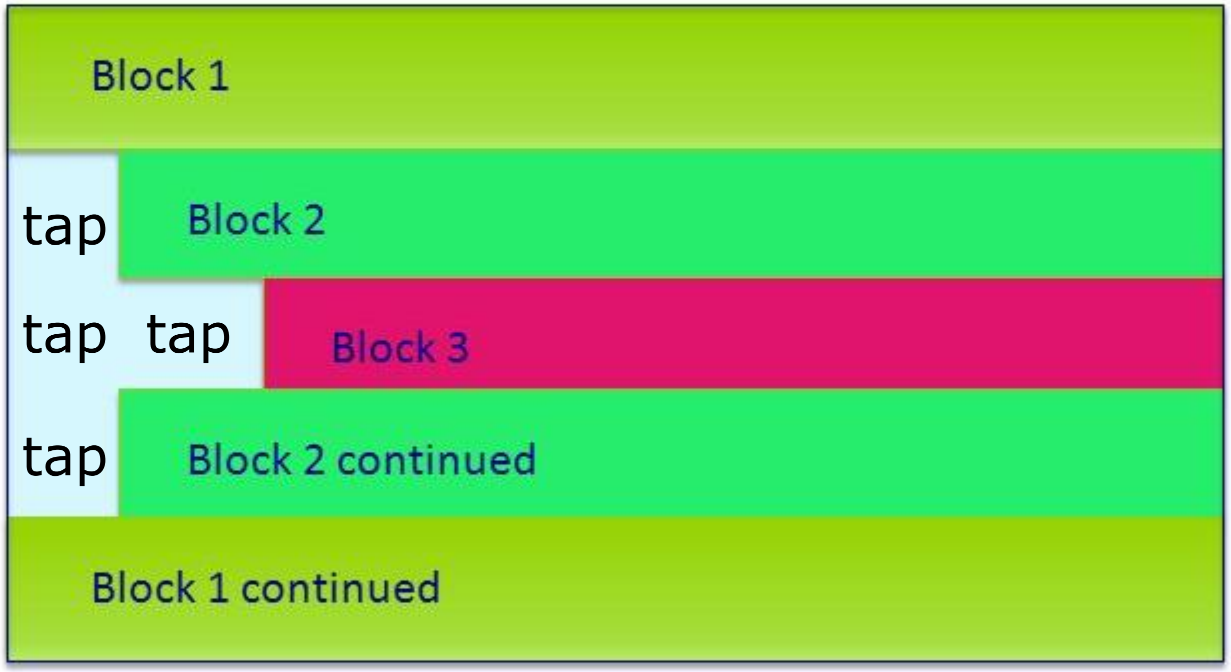
```
student@student-VirtualBox:~$ python3
Python 3.5.2 (default, Jul 10 2019, 11:58:48)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

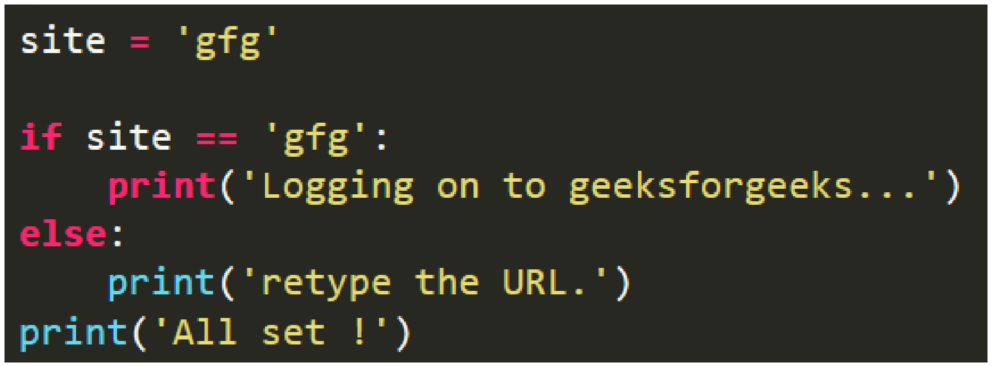


```
sudo apt-get install build-essential python-pip python3-pip python-dev python3-dev
```

Python - indentation

- Python code is structured through indentation (the skeleton of a program below contains no functions):





```
site = 'gfg'

if site == 'gfg':
    print('Logging on to geeksforgeeks...')
else:
    print('retype the URL.')
print('All set !')
```

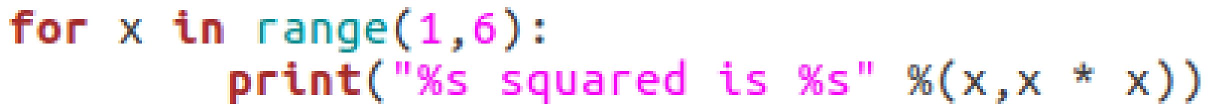
- `for` **loop**: Repeats a set of statements over a group of values.
  - Syntax:

    `for` ***variableName*** `in` ***groupOfValues*** **`:`**

          statements

    - We indent the statements to be repeated with tabs or spaces.
    - ***variableName*** gives a name to each value, so you can refer to it in the ***statements***.
    - ***groupOfValues*** can be a range of integers, specified with the `range` function.
- The `range` function specifies a range of integers:
    - `range(`***start*** **`,`** ***stop***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive)
  - It can also accept a third value specifying the change between values.
    - `range(`***start*** **`,`** ***stop*** **`,`** ***step***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive) by ***step***

  - Example:



```
for x in range(1,6):
    print("%s squared is %s" %(x,x * x))
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

```python
j = 1
while(j<= 5):
     print(j)
     j = j + 1
```

Output:

```
1
2
3
4
5
```

```python
site = 'gfg'

if site == 'gfg':
    print('Logging on to geeksforgeeks...')
else:
    print('retype the URL.')
print('All set !')
```

```python
a = [
    [1, 2, 3],
    [3, 4, 5],
    [5, 6, 7]
    ]

print(a)
```

Output:

```
[[1, 2, 3], [3, 4, 5], [5, 6, 7]]
```

```python
x = \
    1 + 2 \
    + 5 + 6 \
    + 10

print(x)
```

Output:

```
24
```

```python
x = 10

while(x != 0):
 if(x > 5):    # Line 1
  print('x > 5')  # Line 2
 else:         # Line 3
  print('x < 5') # Line 4
 x -= 2        # Line 5
```

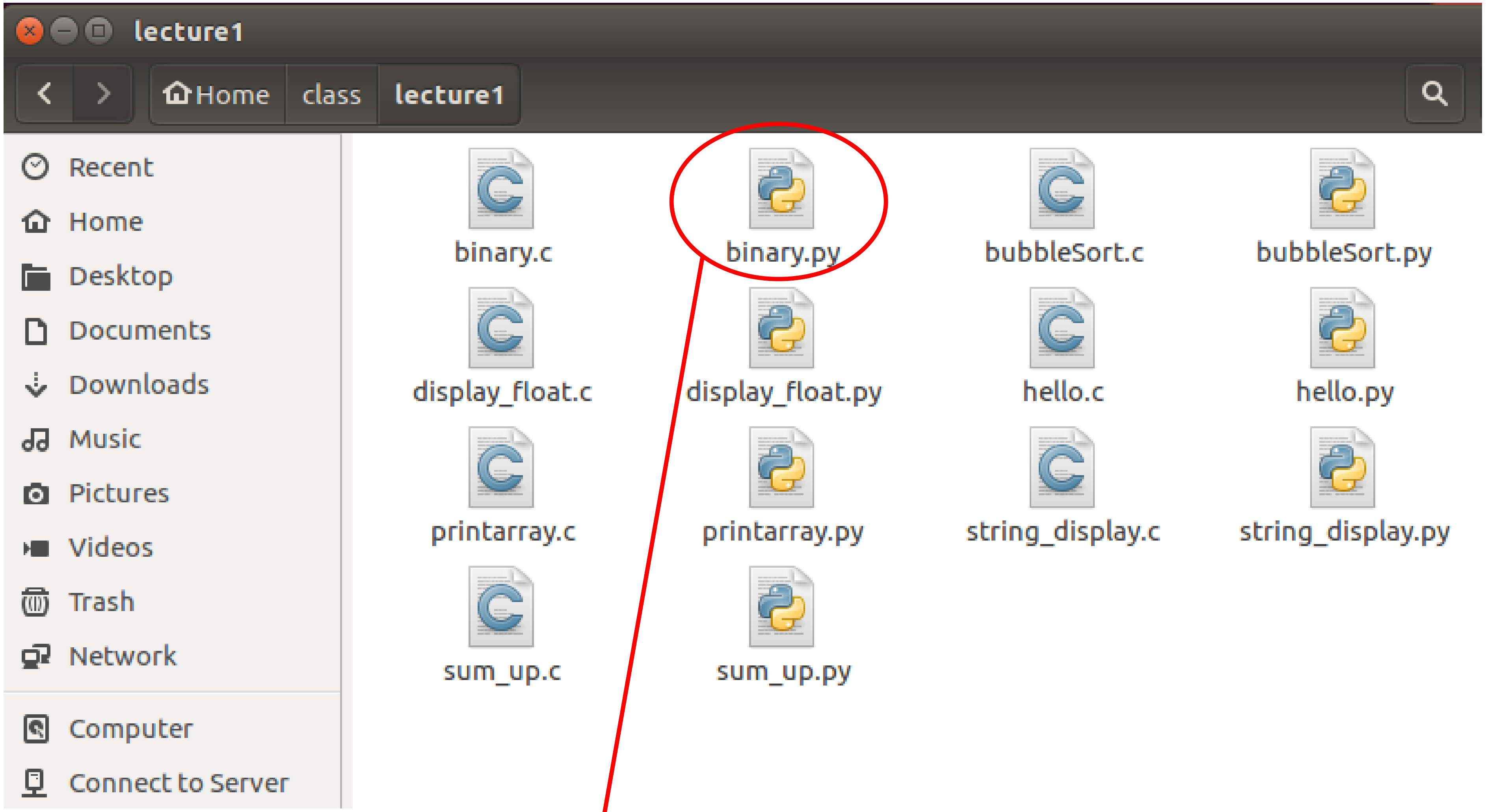
Output:

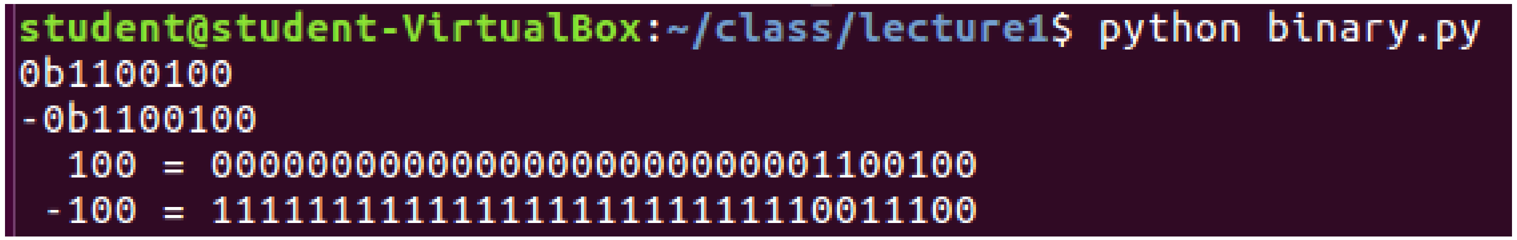
```
x > 5
x > 5
x > 5
x < 5
x < 5
```

```python
x = [1, 2, 3, 4]

# x[1:3] means that start from the index
# 1 and go upto the index 2
print(x[1:3])
```

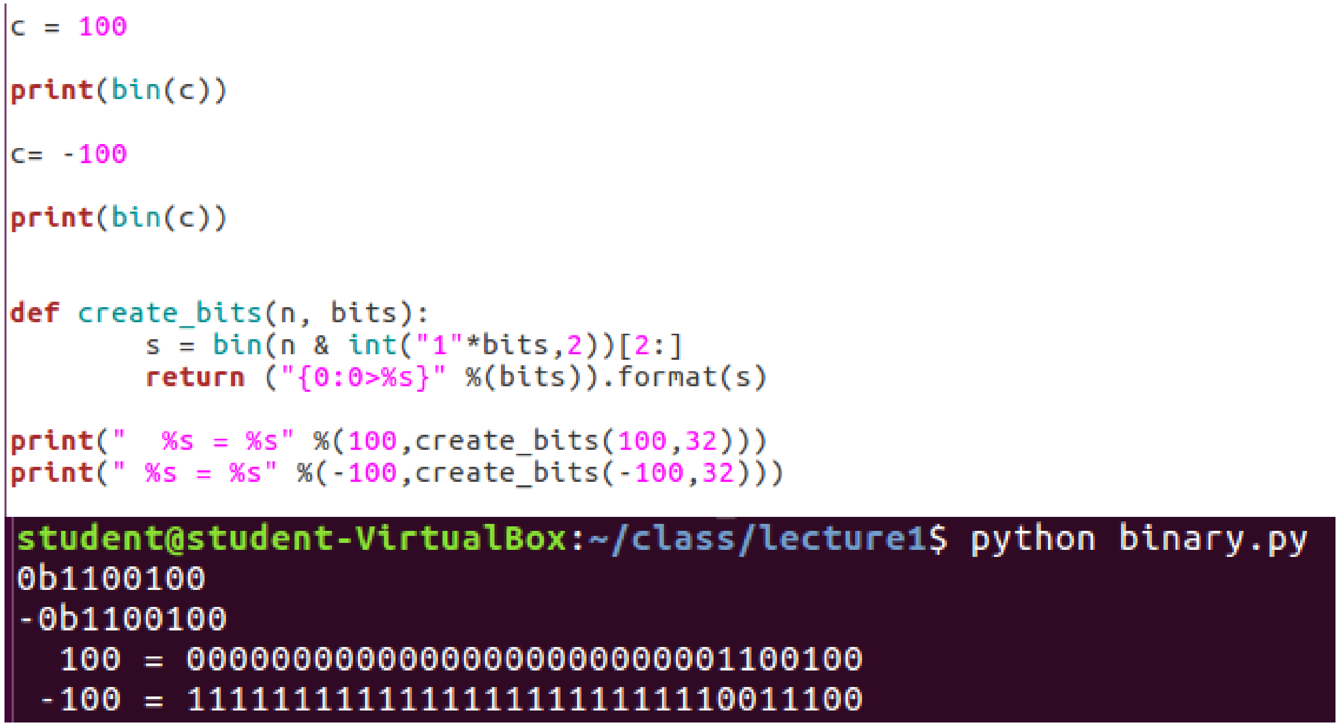
Output:

```
[2, 3]
```





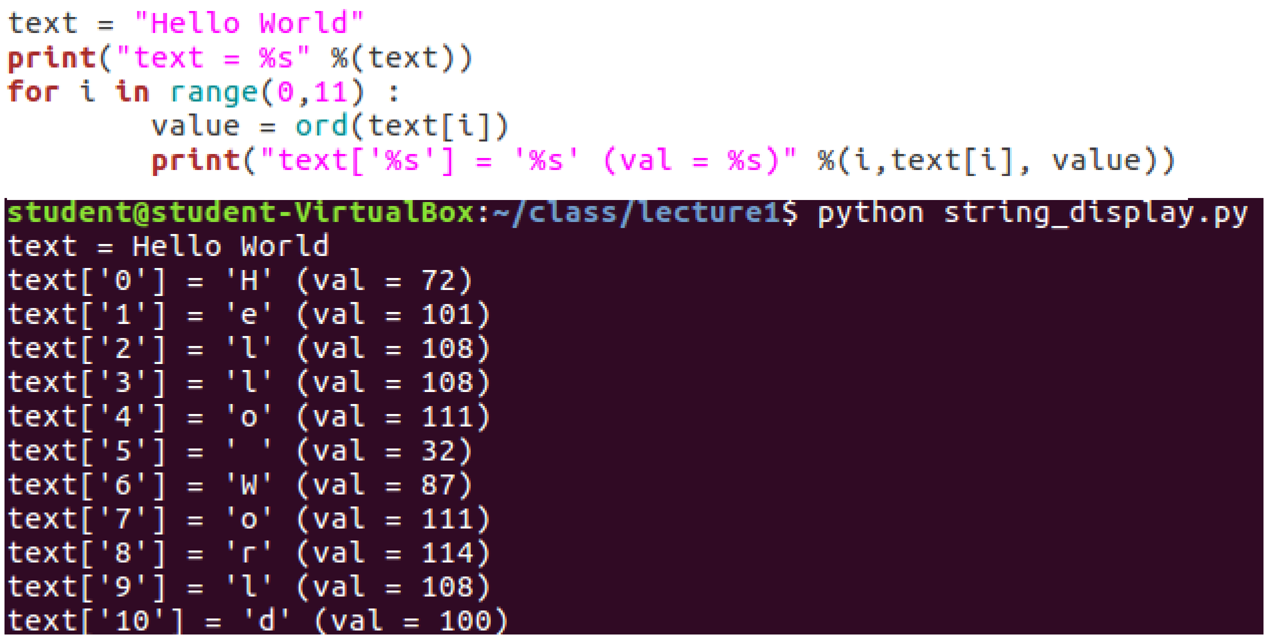
```
student@student-VirtualBox:~/class/lecture1$ python binary.py
0b1100100
-0b1100100
  100 = 00000000000000000000000001100100
 -100 = 11111111111111111111111110011100
```



```
c = 100

print(bin(c))

c= -100

print(bin(c))


def create_bits(n, bits):
        s = bin(n & int("1"*bits,2))[2:]
        return ("{0:0>%s}" %(bits)).format(s)

print("  %s = %s" %(100,create_bits(100,32)))
print(" %s = %s" %(-100,create_bits(-100,32)))
```

```
student@student-VirtualBox:~/class/lecture1$ python binary.py
0b1100100
-0b1100100
  100 = 00000000000000000000000001100100
 -100 = 11111111111111111111111110011100
```

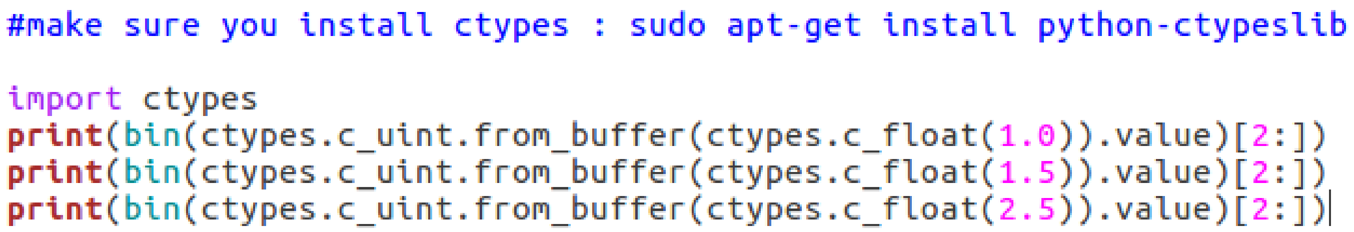
binary.py



```
text = "Hello World"
print("text = %s" %(text))
for i in range(0,11) :
        value = ord(text[i])
        print("text['%s'] = '%s' (val = %s)" %(i,text[i], value))
```

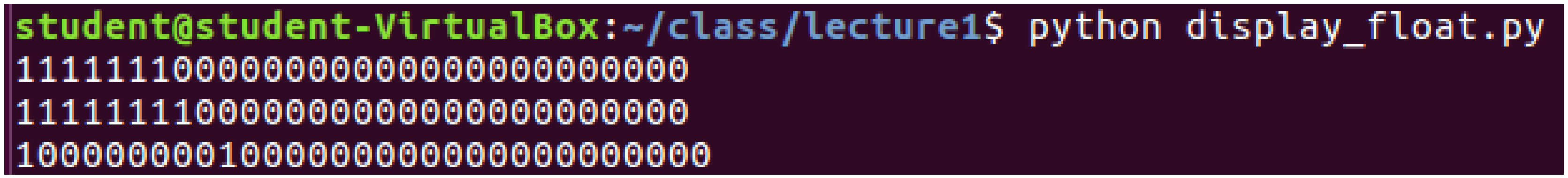
```
student@student-VirtualBox:~/class/lecture1$ python string_display.py
text = Hello World
text['0'] = 'H' (val = 72)
text['1'] = 'e' (val = 101)
text['2'] = 'l' (val = 108)
text['3'] = 'l' (val = 108)
text['4'] = 'o' (val = 111)
text['5'] = ' ' (val = 32)
text['6'] = 'W' (val = 87)
text['7'] = 'o' (val = 111)
text['8'] = 'r' (val = 114)
text['9'] = 'l' (val = 108)
text['10'] = 'd' (val = 100)
```

string_display.py

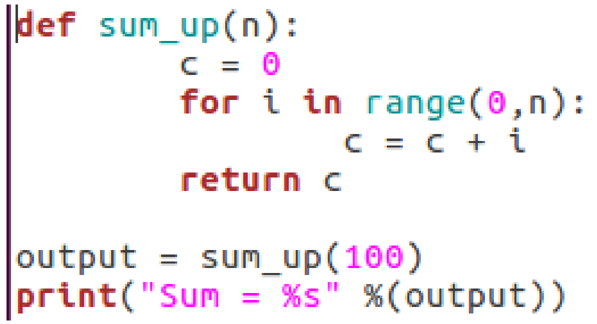


```
#make sure you install ctypes : sudo apt-get install python-ctypeslib

import ctypes
print(bin(ctypes.c_uint.from_buffer(ctypes.c_float(1.0)).value)[2:])
print(bin(ctypes.c_uint.from_buffer(ctypes.c_float(1.5)).value)[2:])
print(bin(ctypes.c_uint.from_buffer(ctypes.c_float(2.5)).value)[2:])
```

display_float.py

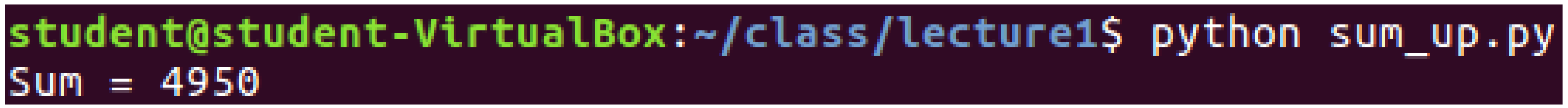


```
student@student-VirtualBox:~/class/lecture1$ python display_float.py
111111100000000000000000000000
111111110000000000000000000000
1000000001000000000000000000000
```

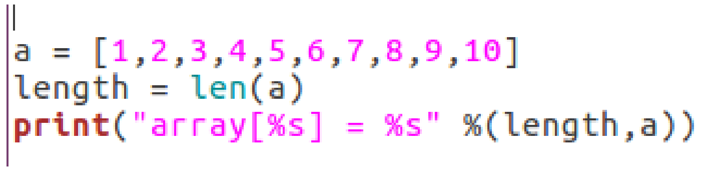


```
def sum_up(n):
        c = 0
        for i in range(0,n):
                c = c + i
        return c

output = sum_up(100)
print("Sum = %s" %(output))
```

sum_up.py

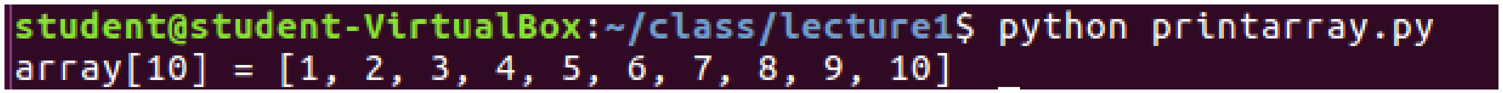


```
student@student-VirtualBox:~/class/lecture1$ python sum_up.py
Sum = 4950
```

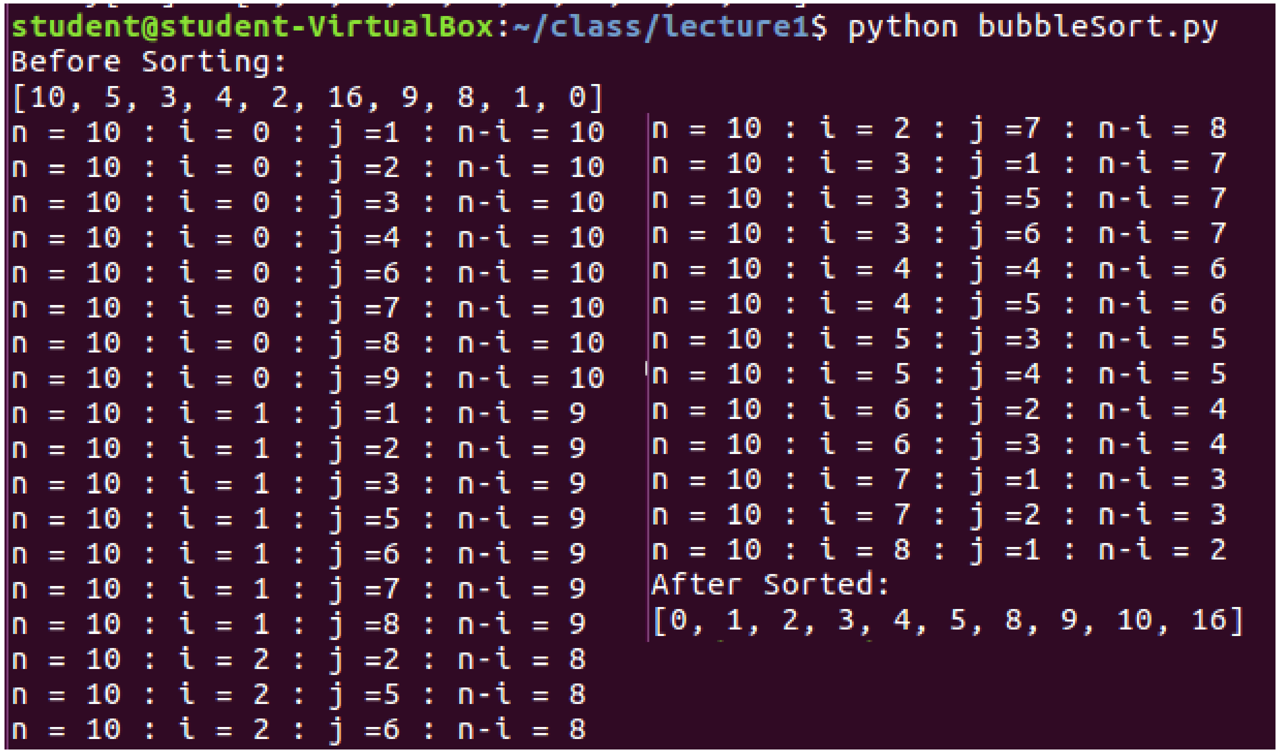


```
a = [1,2,3,4,5,6,7,8,9,10]
length = len(a)
print("array[%s] = %s" %(length,a))
```

printarray.py

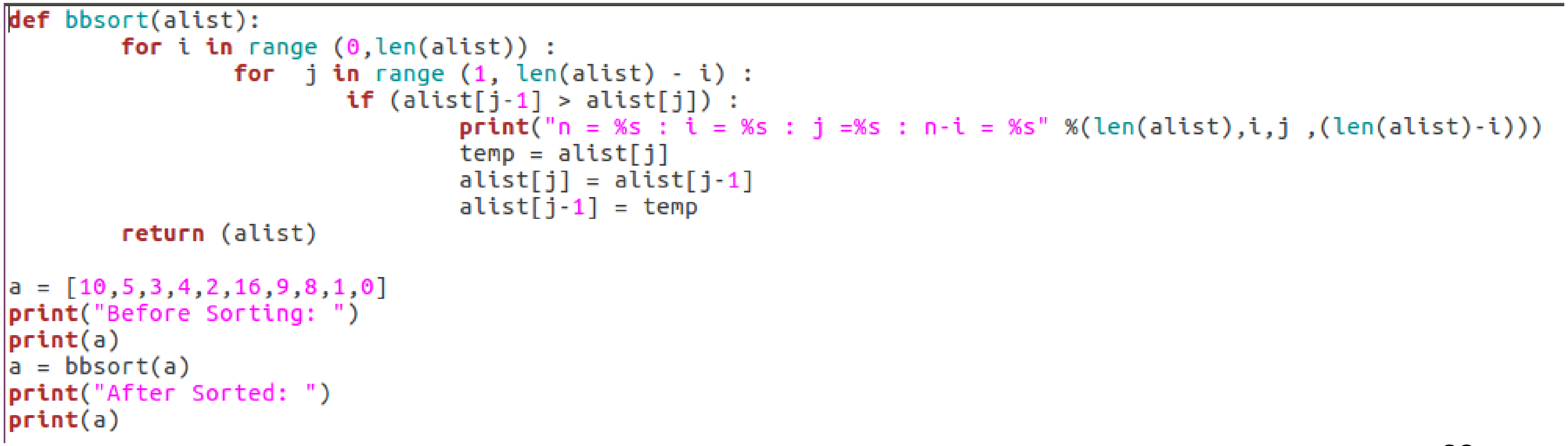


```
student@student-VirtualBox:~/class/lecture1$ python printarray.py
array[10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```



```
student@student-VirtualBox:~/class/lecture1$ python bubbleSort.py
Before Sorting:
[10, 5, 3, 4, 2, 16, 9, 8, 1, 0]
n = 10 : i = 0 : j =1 : n-i = 10
n = 10 : i = 0 : j =2 : n-i = 10
n = 10 : i = 0 : j =3 : n-i = 10
n = 10 : i = 0 : j =4 : n-i = 10
n = 10 : i = 0 : j =6 : n-i = 10
n = 10 : i = 0 : j =7 : n-i = 10
n = 10 : i = 0 : j =8 : n-i = 10
n = 10 : i = 0 : j =9 : n-i = 10
n = 10 : i = 1 : j =1 : n-i = 9
n = 10 : i = 1 : j =2 : n-i = 9
n = 10 : i = 1 : j =3 : n-i = 9
n = 10 : i = 1 : j =5 : n-i = 9
n = 10 : i = 1 : j =6 : n-i = 9
n = 10 : i = 1 : j =7 : n-i = 9
n = 10 : i = 1 : j =8 : n-i = 9
n = 10 : i = 2 : j =2 : n-i = 8
n = 10 : i = 2 : j =5 : n-i = 8
n = 10 : i = 2 : j =6 : n-i = 8
n = 10 : i = 2 : j =7 : n-i = 8
n = 10 : i = 3 : j =1 : n-i = 7
n = 10 : i = 3 : j =5 : n-i = 7
n = 10 : i = 3 : j =6 : n-i = 7
n = 10 : i = 4 : j =4 : n-i = 6
n = 10 : i = 4 : j =5 : n-i = 6
n = 10 : i = 5 : j =3 : n-i = 5
n = 10 : i = 5 : j =4 : n-i = 5
n = 10 : i = 6 : j =2 : n-i = 4
n = 10 : i = 6 : j =3 : n-i = 4
n = 10 : i = 7 : j =1 : n-i = 3
n = 10 : i = 7 : j =2 : n-i = 3
n = 10 : i = 8 : j =1 : n-i = 2
After Sorted:
[0, 1, 2, 3, 4, 5, 8, 9, 10, 16]
```

bubbleSort.py



```python
def bbsort(alist):
        for i in range (0,len(alist)) :
                for  j in range (1, len(alist) - i) :
                        if (alist[j-1] > alist[j]) :
                                print("n = %s : i = %s : j =%s : n-i = %s" %(len(alist),i,j ,(len(alist)-i)))
                                temp = alist[j]
                                alist[j] = alist[j-1]
                                alist[j-1] = temp
        return (alist)

a = [10,5,3,4,2,16,9,8,1,0]
print("Before Sorting: ")
print(a)
a = bbsort(a)
print("After Sorted: ")
print(a)
```

Exercise 1

1. First, write those programs shown in this slides and get used to GCC and CMD console (and DOS)
   - Use “Notepad”, “Wordpad” or any other text editor that you have to write programs
2. Write a program that prints all prime numbers up to N (value N should be given from the command argument)
   - Recall that `(a % b)` means “remainder of `a / b`”
3. Write a program that prints the “prime factored form” of N, such as:
   - 10 = 2 * 5
   - 81 = 3 * 3 * 3 * 3
   - 100001 = 11 * 9091
4. Write a program that multiplies two signed integers WITHOUT using “`*`” operator
5. Write a program that divides two signed integers WITHOUT using “`/`” operator
6. Write your own “`atoi`” function

7. Write a program that prints a sequence of numbers (given from the command argument), sort these numbers, and print the sorted number sequence
   - You can assume some “maximum” length of number sequence that is fixed inside your program.
   - But you should give a warning message that “there are too many numbers” in the command argument.