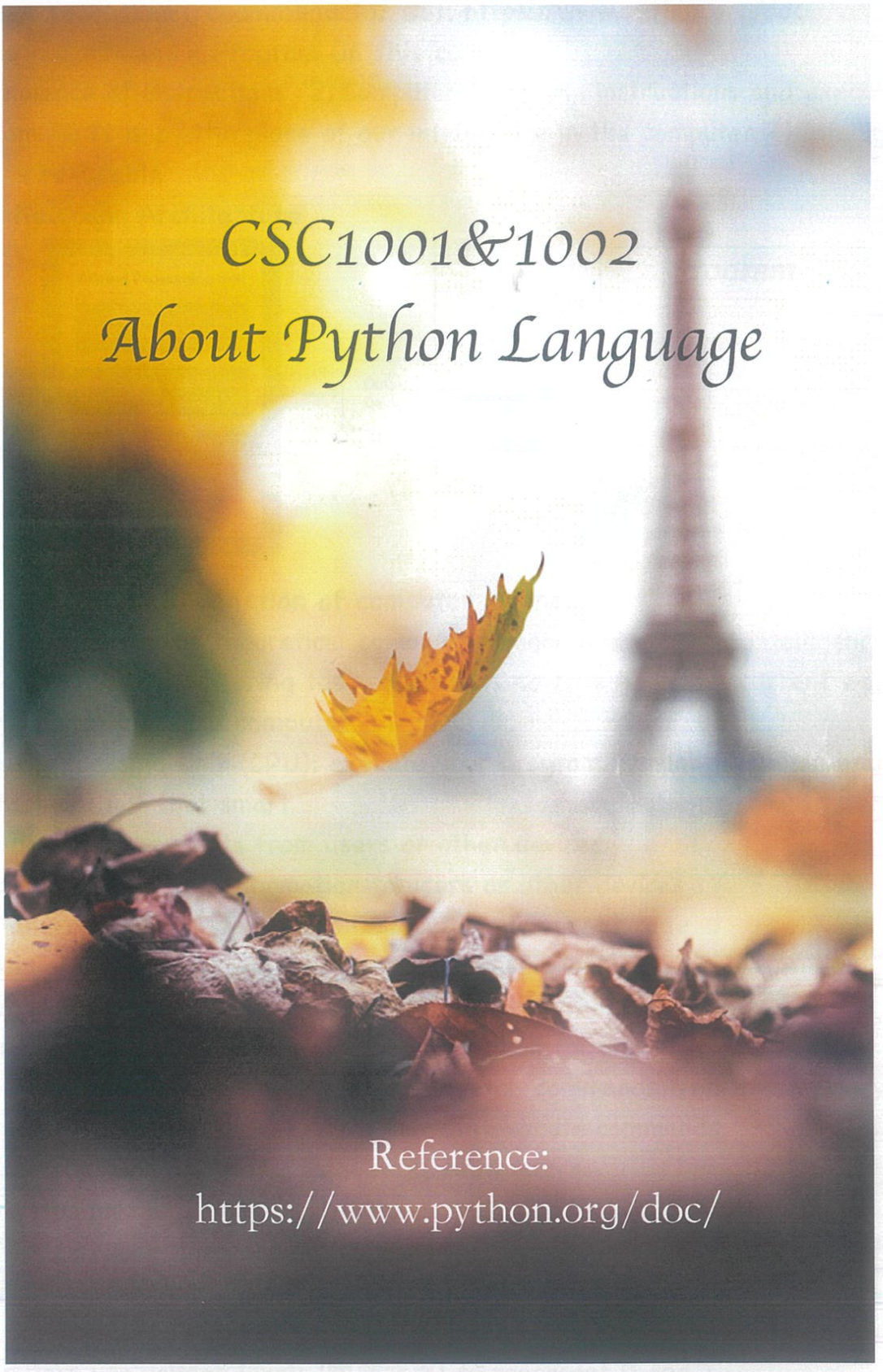
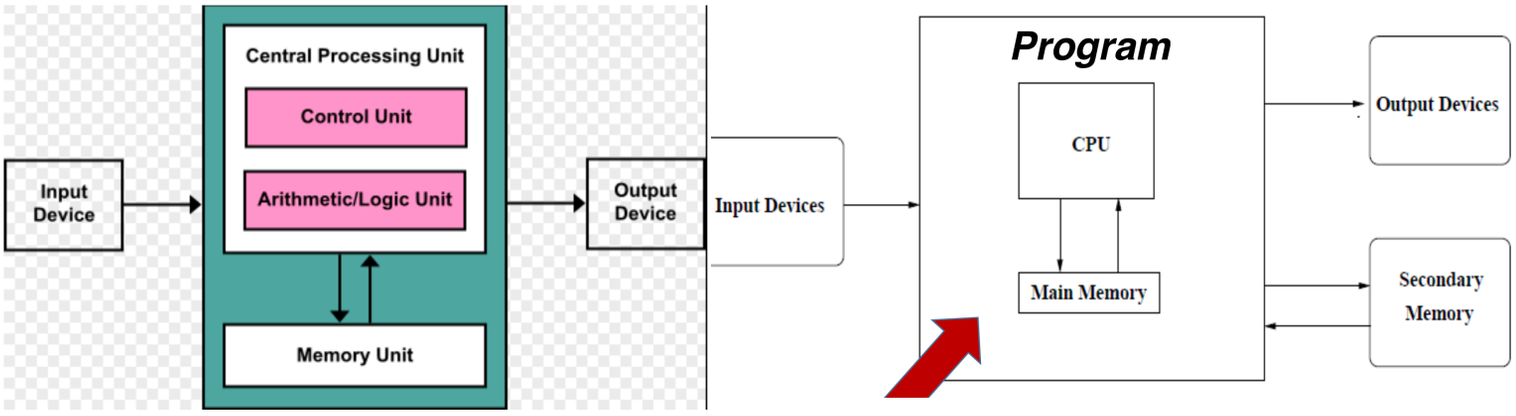

CSC1001&1002
About Python Language
Reference:
https://www.python.org/doc/

CSC 1001 Week 1

- **Online resources (recommend for CS):** https://www.python.org/doc
- **Code = Software = Program (in this course)**

1) A sequence of instructions 2) Computers take the instructions and execute (**执行**) them. 3) It is a little piece of our intelligence in the computer. 4) Intelligence which is **re-usable.**

- **Von Neumann Architecture**



- **The theoretical foundation of computer science**

Alan Turing (Father of theoretical computer science and artificial intelligence)
All the AI cannot pass Turing test, so there is no true AI in the current world.

- **Key components in a computer**

Central processing unit (CPU): execute your program. Similar to human brain, very fast but not that smart
Input device: take inputs from users or other devices
Output device: output information to users or other devices
Main memory (储存): store data, fast but temporary storage
Secondary memory: slower but large size, permanent storage

- **Central Processing Unit**

A processor (**处理器**) contains two units, a control unit (CU) and an arithmetic/logic unit (ALU). **CU** is used to fetch commands from the memory. **ALU** contains the electric circuits which can execute commands.

Processor manufacturers: Intel, AMD, ARM, etc. •

- **Memory/Storage**

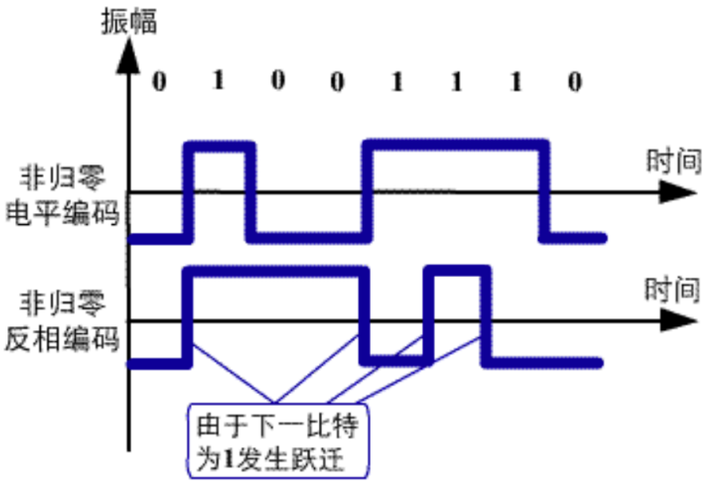
High speed cache, Internal RAM, Internal ROM, External RAM, Flash, Hard disk.

- **Computer's "language"**

The computers used nowadays can understand only binary number(i.e. 0 and 1)
Computers use voltage levels (**电压水平**) to represent 0 and 1
NRZL (**非归零电平编码**)and NRZI (**非归零反相编码**)coding

NRZL—— Use 0 for low, while 1 for high

NRZI—— Use change from 0/1 to 1/0 for change between high and low.



- **Low level language**

1) **Machine language:** The instructions expressed in binary number. It is the fastest language, but completely platform dependent. (i.e. It changes between platforms)
2) **Assembly language:** a type of low level language which has a very strong (generally one-to-one) correspondence between itself and machine code instructions. Each assembly language is specific to a particular computer architecture, converted into executable machine code by a utility program referred to as an **assembler.**

- **"Middle" level language**

1) **C language**(1969-1973): developed by **Dennis Ritchie** between 1969 and 1973 at **AT&T Bell Labs.** It is one of the early high-level programming language, somewhere **between** assembly and other high level languages, widely used in **low level applications**, such as operating systems, embedded programming, super computers, etc.
2) **C++ language**(1979): developed by **Bjarne Stroustrup** at **Bell Labs** since 1979. Inherent major features of C; An object oriented programming language, supporting code reuse; High efficiency and powerful in low level memory manipulation; Still platform dependent.
3) **Java language(1995):** developed by **James Gosling** at **Sun Microsystems** (which has since been acquired by **Oracle Corporation(甲骨文公司)**) and released in 1995; A new generation of general-purpose object oriented (**面向对象的**) programming language; **Platform independent,** ""write once, run anywhere" (WORA); one of the most popular programming languages currently in use.
4) **Python language(1991):** Developed by **Guido van Rossum** in 1989, and formally released in 1991. An **open source (开源),** object oriented programming language with **powerful libraries (库)**; Powerful interfaces to integrate other programming languages.

○ **Language efficiency v.s. development efficiency**

High level languages cannot be executed directly

High level languages **must be converted** into low level languages first

Lower level languages have **higher language efficiency** (they are faster to run on a computer)

Higher level languages have **higher development efficiency.**(it is easier to write programs in these languages)

O Operating Systems

The operating system (OS) is a **low level program,** which provides all **basic services** for managing and controlling a computer's activities

Applications are programs which are built based upon an OS.

Main functions of an OS: Controlling and monitoring system activities; Allocating and assigning system resources; Scheduling operations.

Popular OS: Windows, Mac OS, Linux, iOS, Android...(except for Windows, other OSs are all based on Linux— an open source)

O Data Representation

We use **positional notation (进位记数法)** to represent or encode numbers in a computer.

Data are stored essentially as **binary numbers** in a computer.

In practice, we usually represent data using either **binary (二进制), decimal (十进制), octal (八进制) or hexadecimal (十六进制)** number systems.

We may need to **convert data** between different number systems.

*Note: in the hexadecimal number systems, the **base** is 16, we use **16 symbols** {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

O Conversion

1) Other systems to decimal systems: using numbers time weights

Every number can be decomposed into the **sum** of a series of numbers, each is represented by a **positional value** times a **weight.**

i.e. $N=a_n\times(\text{base})^{n} + \dots + a_0\times(\text{base})^{0} + a_{-1}\times(\text{base})^{(-1)} + \dots$

2) Decimal systems to other systems: big divisions (demonstrating principles)

2	57	1 Lower position
2	28	0
2	14	0
2	7	1
2	3	1
2	1	1 Higher position

$(57)_{10}=(111001)_2$

Assume $(a)_{10} = a_n\cdot 2^n + a_{n-1}\cdot 2^{n-1} + \dots + a_0\cdot 2^0$ ($a_i\in\{0,1\}$)

every time when you divided 2, you decrease the power and get the remain from low to high.

$0.6875 \times 2 = 1.375$ Integer part: 1 Higher position

$0.375 \times 2 = 0.75$ Integer part: 0

$0.75 \times 2 = 1.5$ Integer part: 1

$0.5 \times 2 = 1$ Integer part: 1 Lower position

(the same for $a_{-1}\cdot 2^{-1}+\dots$ just times 2)

3) Conversion between binary, octal and hexadecimal systems: using the one to one correspondence. (demonstrating principles)

$(a)_{8^0} = a_n\cdot 8^n + a_{n-1}\cdot 8^{n-1} + \dots + a_0\cdot 8^0$ just write every $8^n = 2^{3n}$ for carrying $3n$ positions and rewrite $a_n = b_2\cdot 2^2 + b_1\cdot 2^1 + b_0 2^0$ to get a one-to-one correspondence.

$(0)_8 = (000)_2$
$(1)_8 = (001)_2$
$(2)_8 = (010)_2$
$(3)_8 = (011)_2$
$(4)_8 = (100)_2$
$(5)_8 = (101)_2$
$(6)_8 = (110)_2$
$(7)_8 = (111)_2$

$(A)_{16} = (1010)_2$
$(B)_{16} = (1011)_2$
$(C)_{16} = (1100)_2$
$(D)_{16} = (1101)_2$
$(E)_{16} = (1110)_2$
$(F)_{16} = (1111)_2$

$(0)_{16} = (0000)_2$.
$(1)_{16} = (0001)_2$.
$(2)_{16} = (0010)_2$.
$(3)_{16} = (0011)_2$.
$(4)_{16} = (0100)_2$.
$(5)_{16} = (0101)_2$.
$(6)_{16} = (0110)_2$.
$(7)_{16} = (0111)_2$.
$(8)_{16} = (1000)_2$.
$(9)_{16} = (1001)_2$.

○ The units of information (data)

Bit (**比特/位**): a binary digit which takes either 0 or 1

Bit is the smallest information unit in computer programming

Byte (**字节**): **1 byte = 8 bits**, every English character is **represented by 1 byte.**

KB/Kibibyte (**千字节**):1 KB = 2^{10} B = 1024 B

MB/Mebibyte (**兆字节**):1MB = 2^{20} B = 1024 KB

GB/Gibibyte (**千兆字节、吉字节**):1GB = 2^{30} B = 1024 MB

TB/Tebibyte (**兆兆字节、太字节**):1TB = 2^{40} B = 1024 GB

PB/Pebibyte (**拍字节**):1PB = 2^{50} B = 1024 TB

EB/Exbibyte (**艾字节**):1EB = 2^{60} B= 1024 PB

ZB/Zebibyte (**泽字节**):1ZB = 2^{70} B= 1024 EB

YB/Yobibyte (**尧字节**):1YB = 2^{80} B= 1024 ZB

*If a data reaches "PB" level, we can call it a big data.

○ Memory and addressing

A computer's memory consists of an **ordered sequence of bytes** for storing data. Every location in the memory has a **unique address.**

"N" bit computer means using N bits to represent memory.

The **key difference** between high and low level programming languages is whether the programmer has to deal with the memory addressing directly.

Low level language can deal with the memory directly.

High level language must deal with the memory with the help of OS

Specially, C&C++ can use either OS or directly do memory addressing.

CSC 1001 Week II&III

- **Python** is the language of Python interpreter and those who can converse with them. An individual who can speak Python is known as a **Pythonista**. It is very uncommon skill, and may be hereditary. Nearly all known Pythonistas use software initially developed by Guido van Rossum.
- **Interpreter v.s. Compiler**

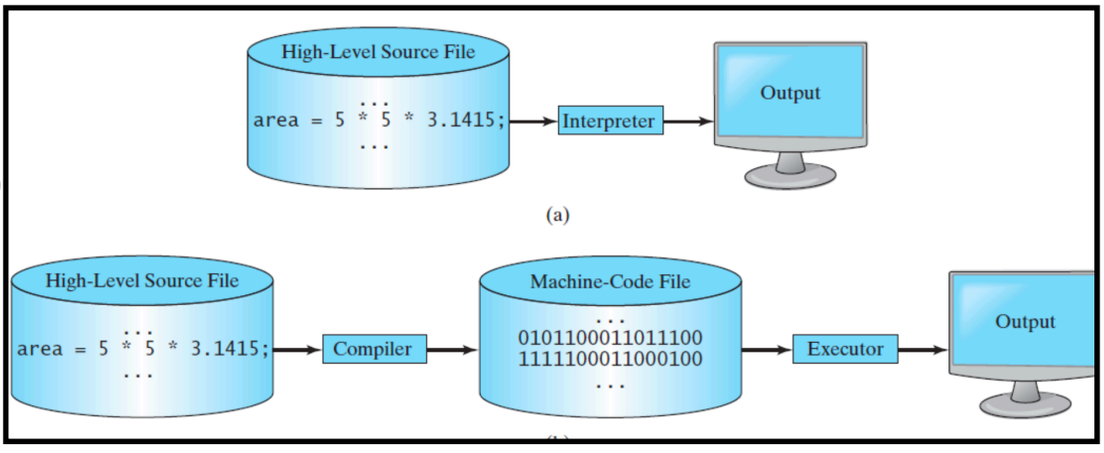
Interpreter (解释器) is a computer program that **directly** executes (i.e. performs, instructions written in a programming or scripting language) **without previously compiling** them into a machine language program.

Compiler (编译器) is a computer program (or a set of programs) that transforms source code written in a programming language (the **source language**) into another computer language (the **target language**), with the latter often having a binary form known as **object code**.

Complier works faster than interpreter.

source file(源文件)

error message (错误信息)



- **Early learner:**

syntax error

When you make a mistake, the computer says "**syntax error**" – given that it "knows" the language and you are just learning it. It seems like Python is cruel and unfeeling. The computer is simple and very fast **but cannot learn.**

- **Elements of Python Language**

Vocabulary/words – **Variables, Reserved words and Operators**

Sentence structure – valid **syntax patterns**

Story structure – constructing a **meaningful program** for some purpose.

Symbols (letters, numbers, operators)—Words—Lines—Program for Python.

- **Variables & Reserved words/key words(保留字/关键字)**

We can set anything as variables but reserved words

```
False    None     True     and      as       assert   break
class    continue def      del      elif     else     except
finally  for      from     global   if       import   in
is       lambda   nonlocal not      or       pass     raise
return   try      while    with     yield
```

O Using Python as a calculator

(1) + means plus; (2) — means minus; (3) * means times; (4) / means division; (5) ** means power; (6) // means floor division; (7) % means remainder ; (8) "+=", "-=", "*=", "/="……

O Sentences or lines

A line or sentence is an element of program with variables, operators, constants and reserved words.

We have types like: Assignment statement; Assignment with expressions; Print statement (output statement)

O Programming scripts

Interactive Python is good for experiments and programs of 3-4 lines long. Interactive mode /shell(**交互式**)

Most programs are **much longer**, so we have to type them **in a file** and **execute them all together.**

In this sense, we are giving Python a **script/file(文件)**

As convention, ".py" is added as the **suffix/extension/filetype(文件类型)** on the end of these files.

O Program steps/flow(程序流)

Program is a **sequence of steps** to be done in **pre-determined order.**

(1) **Sequential flow(顺序流)**: from top tp bottom, from one step to the next

(2) **Conditional flow(条件流)**: If sth satisfy the condition, it will be excused.

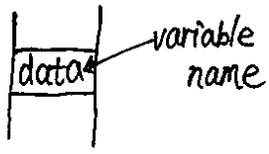
(3) **Repetitive flow(重复流)**: **Loops (repeated steps)** have **iterative variables** that change each time through a loop. Often these iterative variables go through a **sequence of numbers.** **indentation(缩进)**

O Constant(常量) & Variable(变量)

Fixed values such as numbers and letters are called **constants**, since their values won't change.

String(字符串) constants use single-quotes (') or double-quotes (").

A variable is a **named space** in the **memory** where a programmer can store data and later retrieve the data using the **variable name.**



Variable names are determined by programmers.

The value of a variable can be **changed later** in a program.

O Rules for defining variables in Python

(Must start with a letter or underscore '_')

- Case **sensitive**
- Can only contain letters, numbers and underscore
- Must **start with** letters or underscore '_'
- Different: apple, Apple, APPLE

O Tips for defining variables in Python

- Use **meaningful words** as variable names
- Start with a **lower letter**
- Capitalize the **first letter** of each word
- Example: myBankAccountID, numOfCards, salaryAtYear1995...

○ **Assignment statement**

We assign a value to a variable using the **assignment operator (=)**

An assignment statement consists of an **expression on the right hand side,** and a **variable** to store the result.

There is a location in the memory for x.

Whenever the value of x is needed, it can be retrieved from the memory.

After the expression is evaluated, the result will be put back into x.

Augmented assignment: <lvalue><operator> = <expression>. Namely <lvalue>=<lvalue><op><expression>

- **Cascaded assignment**

We can set multiple variables into the same value using a single assignment statement (i.e. x=y=z=.....)

- **Simultaneous assignment**

The values of two variables can be exchanged using simultaneous assignment. (i.e. a, b=b, a)

- **Highest to lowest precedence rule**

✓ Parenthesis are always with highest priority
✓ Power
✓ Multiplication, division and remainder
✓ Addition and subtraction
✓ Left to right

○ **Order evaluation/ Operator precedence**

*NOTE: divmod()= division & mod—a function like print and exit. two arguments

divmod(a,b)
return (a//b, a%b)
a tupple

○ **Tips for calculating in Python**

- Keep mathematical expressions **simple** so that they are easy to understand
- **Break up** long series of math expressions to make them easy to understand
- Use parenthesis(括号)

○ **Functions**

print(): output of something
input(): stop and take user inputs as a string
exit(): exit from python
divmod(): division & mod
type(): find the type of sth
int()/float()/bool()/str(): transfer the types
eval(): take a string argument and evaluates that string as a Python expression(表达式), returning the result of that expression

O String operations

Some operators **apply to strings**

✓"+": concatenation

✓"*": multiple concatenation Debug(调试).

O Data type(数据类型)

'Float' floating point number (浮点数)

'Complex' (复数) j = (-1)^(1/2)

'Bool' true or false

O Others

"#" as comment symbol-its usage: (1) help read the program (2) remove some lines.

Ending symbol 'end=' can prevent changing line.

Comparison operators: boolean expression—布尔表达式(logical)

* x!=y "Is x not equal to y?"

== comparison & = assignment.

Order: numerical operator > comparison operator > assignment operator

1.e-2=1*10^(-2) (Note 1.e(-) negative number or positive number)

m.en = m×10^n

O String Order

ASCII table

ASCII 码	键盘	ASCII 码	键盘	ASCII 码	键盘	ASCII 码	键盘
27	ESC	32	SPACE	33	!	34	"
35	#	36	$	37	%	38	&
39	'	40	(	41	)	42	*
43	+	44	'	45	-	46	.
47	/	48	0	49	1	50	2
51	3	52	4	53	5	54	6
55	7	56	8	57	9	58	:
59	;	60	<	61	=	62	>
63	?	64	@	65	A	66	B
67	C	68	D	69	E	70	F
71	G	72	H	73	I	74	J
75	K	76	L	77	M	78	N
79	O	80	P	81	Q	82	R
83	S	84	T	85	U	86	V
87	W	88	X	89	Y	90	Z
91	[	92	\	93	]	94	^
95	_	96	`	97	a	98	b
99	c	100	d	101	e	102	f
103	g	104	h	105	i	106	j
107	k	108	l	109	m	110	n
111	o	112	p	113	q	114	r
115	s	116	t	117	u	118	v
119	w	120	x	121	y	122	z
123	{	124	|	125	}	126	~

The bigger ASCII number is, the larger the string is.

(Especially, small letter > Capital letter)

When comparing the string, compare the first letter first, then second ..... until the last one.

(According to sequence / order.)

use "chr" & "ord"

(Other way — UTF-8)

ord('') ← strings

chr( ) ← numbers

CSC 1001 Week IV Flow Control

Topic I Comparison Operators

1. *Boolean expressions* ask a question and produce a *Yes/No* result, which we use to *control program flow.*
2. *Boolean expressions* use *comparison operators* to evaluate Yes/No or True/False.
3. *Comparison operators* check variables but do not change the values of variables
4. *Careful!!* "=" is used for assignment, while "==" is used for comparison.
5. **Boolean type:** Python contains a *built-in Boolean type*, which takes *two values True/False*; *number 0* can also be used to represent False. *All other numbers* represent True.

Topic II Indentation

1. **Increase indent**: indent after an if or for statement *(after :)*; Aim: entering body of a statement
2. **Maintain indent**: to indicate the scope of the block *(which lines are affected by the if/for)*; Aim: still in body of a statement
3. **Decrease indent**: to back to the level of the if statement or for statement to indicate the end of the block. Aim: exit the body of a statement.
4. *Blank lines are ignored* – they do not affect indentation.
5. Comments on a line by themselves are ignored w.r.t. indentation.

Topic III Decisions In 'If' Statement

1. **One-way decisions**: just use one 'if' with nothing.

```
if x==5:
    print('Is 5')
```

2. **Nested decisions:** There are other 'if's under one 'if'.

Note that we should avoid too many 'if's. It is well when nested 'if's are fewer than 3.

When the condition of the first 'if' fails, the nested ones also fail.

```
x=42
if x>1:
    print('More than 1')

    if x<100:
        print('Less then 100')

print('Finished')
```

3. **Two-way decisions:** Do one thing when 'true', do another when 'false'. It is like a fork in the road, we need to *choose one or the other path*, but *not both*.

Using 'if' and 'else' statement. ('if' can appear alone, but 'if' is indispensable for 'else'.)

Use indentation to match 'if' and 'else'. 'else' must come after 'if'.

```
x=1

if x>2:
    print('Bigger')
else:
    print('Smaller')
```

4. **Multi-way decisions:** There are multiple ways under different conditions.

Use 'if', 'else' and 'elif' to realize it. (We can also just use 'if' and 'elif'.)

```
x=2
if x<2:
    print('Small')
elif x<10:
    print('Medium')
else:
    print('Large')
```

Topic V Logical operators

1. In Python, we use 'and', 'or', 'not' to serve as logical operators.
2. *Logical operators* can be used to *combine* several logical expressions into *a single expression.*
3. It has same logic with math: (i) 'and' means '$\wedge$'. If one fails, then all fail.
   (ii) 'or' means '$\vee$'. If one succeeds, all succeed.
   (iii) 'not' means '$\neg$'. It just means the opposite.

Topic VI Try & Expect Statement

1. 'try' and 'except' belong to conditional flows, but they are different from others.
2. Surrounding a *dangerous part of code* with 'try/except': if the code in 'try' block *works,* the except block *will be skipped;* if the code in try block *fails,* the except block *will be executed* (Then it get into *the code following 'except'*, the program *will not be terminated.*).

```
astr = 'Bob'
try:
    print('Hello')
    istr = int(astr)
    print('There')
except:
    istr = -1
print('Done', istr)
```

3. 'try' and 'except' must appear in pair.

* An example of a ~~tupple~~ tuple (**元组**):

```
>>> a=(10,20,30)
>>> a[0]
10
>>> a[1]
20
>>> a[2]
30
>>> a[-1]
30
```

a tuple can have different types of elements
a=(1, 2, 'a', ('b', True))

Topic VII Repeated Flows

1. Repeated Flows: ***Loops (repeated steps)*** have *iterative variables* that change each time through a loop. These iterative variables often *go through a sequence of numbers.*
2. Infinite loops: (i) made by mistake: the change of variables must be in the body of 'while' statement. (ii) made deliberately: use 'True' as initial condition.

```
n=5
while n>0:
    print('Lather')
    print('Rinse')
n=n-1
print('Dry off!')
```

error type ⇒ infinite

```
while True:
    line = input('Input a word:')
```

3. Break out of an infinite statement -- 'break' and 'continue' statements: The 'break' statement *ends the current loop,* and ***jumps to the statement which directly follows the loop.*** The 'continue' statement *ends the current iteration,* and ***start the next iteration immediately.*** Iteration variable (**循环变量**).

(continue = does not do following (all) steps)

4. Definite vs. Indefinite Loops:

(i) Indefinite loop:

- 'while' loops are called "indefinite loops", since they keep going until a logical condition *becomes false.*
- Sometimes it can be hard to determine whether a loop will terminate.

(ii) Definite loop:

- Quite often we have *a finite set of items.*
- We can use a loop, each iteration of which will be executed for each

item in the set, using the for statement.

- It is said that "definite loops iterate through the members of a set".

5. Useful definite loop -- 'for' & 'in' statement:

(i) 'for' loops (definite loops) have explicit iteration variables that change each time through a loop.

(ii) These iteration variables move through a sequence or a set.

(iii) The iteration variable "iterates" through a sequence (ordered set).

(iv) The block (body) of the code is executed once for each value in the sequence.

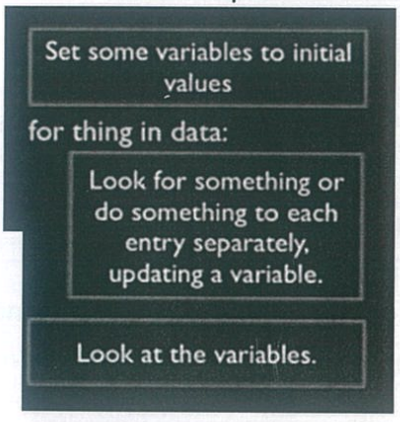
(v) The iteration variable moves through all of the values in the sequence.

(vi) Making "smart" loops:

6. Samples:

(i) Filtering in a loop:

We can use an 'if' statement in a loop to catch/filter the values we are interested at.



Example

```
print('Before')
for value in [23, 3, 43, 39, 80, 111, 99, 3, 65]:
    if value>50:
        print('Large value:', value)
print('After')
```

Output

```
Before
Large value: 80
Large value: 111
Large value: 99
Large value: 65
After
```

(ii) Search using a Boolean variable: If we want to search in a set and double check whether ***a specific number is in that set***. We can use a Boolean variable, set it to ***False at the beginning***, and assign ***True*** to it as long as the ***target number is found.***

(iii) Finding the smallest number:

- We still use a variable to store the smallest value seen so far.
- In the first iteration, the smallest value is **'None'**, so we need to use an 'if' statement to check this. (set 'None' to the initial value)

6. The **'is'** and **'is not'** operator:

(i) Python has a ***"is"*** operator which can be used in ***logical expression***

(ii) Implies "***is the same as***"

(iii) Similar to, but ***stronger than*** '==':

(iv) ***"is not"*** is also an operator.

```
>>> a=['w','y','z']
>>> b=['w','y','z']
>>> print(a is b)
False
>>> print(a == b)
True
```

a=10
b=10
print(a is b)
→ True

(v) 'id()' function
(address of a variable)
id() shows

'is' — compare their address
'==' — compare their values

CSC 1001 Week V Function

Topic I Meaning of Functions

1. A ***function*** is a paragraph of a reusable function code.
2. Construct a function: use ***'def',*** with the function name after, then a colon following.
3. Types of functions in Python: (i) ***Built-in functions*** which are part of Python, such as print(), int(), float(), etc. (ii) Functions that we define ourselves and then use.
4. The names of built-in functions are usually considered as ***new reserved words.*** (i.e. we do not use them as variable names.)
5. Function definition

- In Python, a function is some reusable code which can *take arguments as input*, perform some computations, and then *output some results*.
- Functions are defined using reserved word ***'def'***
- We call/invoke a function by *using the function name*, parenthesis and arguments in an expression.

6. ***'max()', 'min()'*** functions: pick up the largest one in any type of collections.
7. Try to write everything into functions, and break the complicated codes into different functions. This is because others can reuse it when only knowing name, input and output.

Topic II Arguments & Parameters

1. **Arguments:** An argument is a ***value*** we pass into the function as its input when we call the function--

- We use arguments so we can direct the function to do different kinds of work when we call it at different times.
- We put the argument in parenthesis *after the name of the function*.

2. **Parameters:** A parameter is a ***variable*** which we use in the function definition that is a 'handle' that allows the code in the function to access the arguments for a particular function invocation. (be memorized in the function definition step)
3. **Arguments and Parameters must satisfy one-to-one relationship in number.**
4. Multiple parameters/arguments:

- We can define more than one parameter in a function definition.
- We simply add more arguments when we call the function.
- We match the number and order of arguments and parameters.

5. Default Arguments: Default argument

- Python allows you to define functions with ***default argument values***.
- The default argument values will be passed to the function, when it is *invoked without arguments*.

```
def printArea(width = 1, height = 2):
    area = width * height
    print("width:", width, "\theight:", height, "\tarea:", area)

printArea() # Default arguments width = 1 and height = 2
printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
printArea(height = 5, width = 3) # Keyword arguments width
printArea(width = 1.2) # Default height = 2
printArea(height = 6.2) # Default width = 1
```

Topic III Return Values

1. Return values

- A fruitful function is one that ***produces a result (or return value).***
- The return statement ends the function execution and ***'sends back' the result*** of the function.

2. For using return, one must **assign variables to the function.**

3. **Void functions:**

- When a function ***does not return a value***, it is called a "void" function
- Functions that return values are "fruitful" functions.
- Void functions are "not fruitful".

Use indentation to match 'if' and 'else'. 'else' must come after 'if'.

4. Functions without return: When a function has no return statement, it will return ***'None'.***

5. ***'Return'*** does not print the output, but it has outputs and memorize them. ***'Return'*** also means the end of the function definition.

6. Return multiple values:

- Python allows a function to return multiple values.
- The sort function returns two values; when it is invoked, you need to pass the returned values in a ***simultaneous assignment***.

```
def sort(number1, number2):
    if number1 < number2:
        return number1, number2
    else:
        return number2, number1

n1, n2 = sort(3, 2)
print("n1 is", n1)
print("n2 is", n2)
```

Topic V Scope (作用范围) of Variables

1. **Scope of variables**: The scope of a variable is the part of program where this variable can be accessed

2. ***Local variables:*** variables created inside functions

3. ***Global variables:*** variables created outside all functions and are accessible to all functions in their scope

4. Different variables can share the same name when having different scopes

5. Transform between scopes:

In a function, you can use keyword global to specify that a variable is a global variable.

Be very careful when define and use global variable.

Topic VI String Type

1. String type:

- A string is a sequence of characters.
- A string literal uses quotes ' or "".
- For strings, + means "concatenate".
- When a string contains numbers, it is still a string.
- We can convert numbers in a string into a number using int() or float().

2. Reading and converting:

- We prefer to read data in using strings and then parse and convert the data as we need.
- This gives us more control over error situations and/or bad user inputs.
- Raw input numbers must be converted from strings.

3. Looking inside strings:

- We can get any character in a string ***using an index*** specified in square brackets.
- The index value must be an integer which ***starts from zero.***
- The index value can be an expression.

You will get ***a Python error*** if you attempt to index beyond the end of a string, so be careful when specifying an index value.

4. Look deeper into *'in'*:

- The iteration variables "iterates" through the sequence (ordered set).
- The ***block (body)*** of the loop is executed once for each value in the sequence.
- The iteration variable moves through all the values in the sequence.
- The *'in'* keyword can also be used to check whether one string is in another string.
- The *'in'* expression is a logical expression and returns True or False.
- It can be used in if or while statement.

5. Slicing strings:

- We can also look at any continuous section of a string using colon operator.
- The second number is one beyond the end of the slice – i.e. "***up to but not including***".
- If the second number is beyond the length of the string, it stops at the end.
- If we leave off the first or second number of the slice, it is assumed to be the beginning or end of the string respectively.

6. String library:

- Python has a number of string functions which are in the string library.
- These functions are built-into every string, we invoke them ***by appending the function to the string variable.(appending function after variable name/constant & '.')***
- These functions ***do not modify*** the original string, instead they return a new string altered from the original string.

(we can use *'dir()'* & *'help()'* to find them)

(i) Searching a string

- We can use the *'find()'* function to search for a substring in a string
- *'find()'* finds the first occurrence of the target sub-string
- If the ***sub-string is not found, it returns -1***

(ii) Making everything upper or lower case:

- You can convert a string into upper case or lower case
- Hint: often when we use *'find()'* to find a substring, we convert the original string into lower case first, so that we don't need to worry about case.

(iii) Search and replace:

- The *'replace()'* function is like a "search and replace" operation in a word processor.
- It replaces all occurrences of the search string with the replacement string.

(iv) stripping whitespace

- *'lstrip()'* and *'rstrip()'* to the left and right respectively only.
- *'strip()'* removes both beginning and ending whitespaces.

(v) Prefixes:

- *'startswith()'* function checks whether a string is starting with a given string.

(*'endswith()'* has the similar utility to it.)

Topic VII File processing

1. A ***text file*** can be thought of as ***a sequence of lines.***

2. Opening files: (take ".txt" → text file as an example)

- Before we can read the contents of a file, we must tell Python which file we are going to work with and what we will do with that file.
- This is done with the *'open()'* function.
- *'open()'* returns a "file handle" - a variable used to perform operations on files.
- Kind of like "File -> Open" in a word processor.

Using *'open()':* path + filename

- handle = open(filename, mode). file handle (文件手柄)
- Returns a handle used to manipulate the file.
- Filename is a string.
- Mode is optional, use 'r' if we want to read the file, and 'w' if we want to write to the file. reading as a string — file.read()

2. The newline character:

- We use a new character to indicate when a line ends called "newline"
- We represent it as '\n' in strings
- Newline is still one character, not two

{ no file — writer → create new one
{ no file — read → error

3. Writing to a file

- To write a file, use the *'open()'* function with 'w' argument
- Use the *'write()'* method to write to the file

read txt
file.read() → as strings
file.read(num) → read num length characters.

different files have different ways to process.

☆Remember to close the file. → otherwise, file.seek() it'll prevent others from using "open()"

Addition from CSC 1002 Lecture:

'ord()': the order in ASCLL table of sth;

'chr()': transform the order into signs

File (文件) { text file (文本文件) → oldest files' type
{ binary file (=进制)文件)

text file is a type of host file — used to climb the firewall.

CSC 1001 Week VI List

Topic I List — a powerful type of collection

1. A ***list*** is nothing but a collection containing *many values* in a single *"variable".*
2. Uniqueness of list in Python: ***list* (列表); *array* (数组)** in C & C++.
3. Most of our variables have *only one value* in them – when we put a *new value* in the variable, the *old value* will be *over-written.*
4. ***List constants:*** surrounded by square brackets and the elements in the list are separated by commas. (A list element can be **any Python object**(even another list) and it list can also be empty.)
5. Just like strings, we can access any single element in a list *using an index* specified in *square bracket.*
6. ***Lists are mutable:*** we can change an element of a list using index operator.
7. The ***'len()'*** function takes a list as input and returns the number of elements in that list, more powerfully it tells us the number of elements in any sequence (e.g. strings).
8. The ***'range()'*** function returns a list of numbers, it returns a ***pseudo list*(伪列表)**. type('range') = 'class range' – meaning it is not a real list.
9. Lists can be sliced similar to strings, the second number is "up to but no including".
10. We can ***concatenate*** lists with operators '+' & '*'.

Topic II List Methods

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__'
, '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__'
, '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__'
, '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__s
izeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'ex
tend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

1. Two ways to build empty lists: x = []; x = list()

Python can create an empty list using list(), and then add elements using ***'.append()'*** method.

The list stays in order, and new elements are added at the end of the list.

2. Python provides logical operators that ***return True or False*** to check whether an item is in a list. They do not modify the list.
3. A list is an **ordered sequence:**

- A list can hold many items and *keeps them in the order* until we do something to change the order.
- A list can be sorted (i.e. change the order)
- The ***'sort()'*** method means "sort yourself". ***'sort()'*** works only if all elements in the list have the same type. ***'sort(reverse = True)'*** will return in descending order.

4. Built-in functions and lists:

```
(len(numbers))
(max(numbers))
(min(numbers))
(sum(numbers))
```

5. Transformation between strings and lists:
(i) Use the ***'split()'*** method to break up a string into a list of strings. If () has nothing, Python splits by blank.
(ii) When you do not specify a delimiter, multiple spaces are treated like "one" delimiter.
(iii) You can specify what delimiter character to use in splitting.

Topic III Dictionaries

1. Dictionary:
(i) Dictionaries are Python's ***most powerful data collection.***
(ii) Dictionaries allow us to do fast database-like operations in Python, but they require the most storage space.
(iii) Dictionaries have different names in different languages: Associative arrays – Perl/PHP; Properties or Map or HashMap – Java; Property Bag – C#/.Net
2. Dictionaries ***index the elements*** we put in the dictionary with a "lookup tag", called ***key/tag/label.***
3. Two ways to create an empty dictionary: {} and ***'dict()'***
4. Properties: (i) no order! (ii) Tags have some real meanings.
5. Dictionary tracebacks: we can use the ***'in'*** operator to see if a key is in the dictionary. (cannot check a value)
6. The ***'get('key', default value)'*** method: check to see if a key is already in a dictionary, if in, return the corresponding value, if not, return a default value.
7. Even though dictionaries are not stored in order, we can write a for loop that goes through all elements in a dictionary – actually it goes through all the keys in that dictionary and looks up the values.
8. Retrieving lists of keys and values:
Python can get a list of ***keys***, ***values*** or ***items (both)*** from a dictionary.
Format: ***'dictionar_name.xxx()'*** – xxx can be keys, values and items.

```
>>> counts.keys()
dict_keys(['chuck', 'fred', 'jan'])
>>> list(counts.keys())
['chuck', 'fred', 'jan']
```

Remember to initially use ***'list()'***, subsequently it can be printed.
9. Two iteration variables:
(i) We loop through the ***key-value pairs*** in a dictionary using two iteration variables.
(ii) Each iteration, the ***first*** variable is the ***key***, and the ***second*** variable is the ***corresponding value*** for the key.

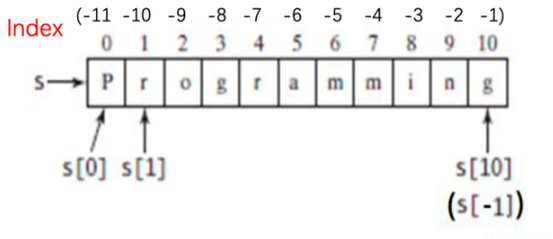
Topic IV Tuples

1. Tuples are immutable: Unlike a list, once you create a tuple, you ***cannot change its contents*** – similar to a string.
2. Tuples are more efficient:

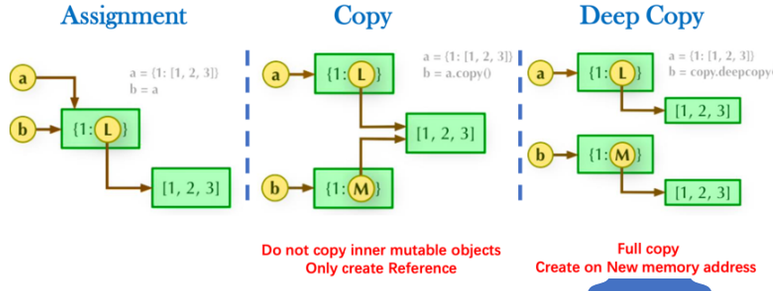
• Since Python doe not have to build tuple structures to be modifiable, they are ***simpler and more efficient*** in terms of memory use and performance than lists.
• In our program when we are making ***"temporary variables"*** we prefer tuples over lists.
3. Tuples are comparable: The ***comparison operators*** work with tuples and other sequences. If the first item is equal, Python goes on to the next element, until it finds the ***elements which are different***.
4. Sorting lists of tuples:
• We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary.
• First we sort the dictionary by the key using the ***'items()'*** method
5. Using ***'sorted()':***
We can do this even more efficiently using ***a built-in function sorted()*** which takes a sequence as a parameter and returns a sorted sequence.

Knowledge Addition for CSC 1001 Tut:

1.



2. ***'s.swapcase()'***: returns a copy of the string s in which lowercase letters are converted to uppercase and uppercase to lowercase.
s.rfind(s1) : returns the highest index where the substring s1 starts.
3. String-Judgement:
i) ***'s.isalnum()'***, return True if s contains only letters and numbers
ii) ***'s.isalpha()'***, return True if s contains only letters UTF-8/ ASCII
iii) ***'s.isdigit()'***, return True if s contains only numbers NOT only for English letters
4. Copy:
直接赋值：其实就是对象的引用（别名）。
浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。
深拷贝(deepcopy)： copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。



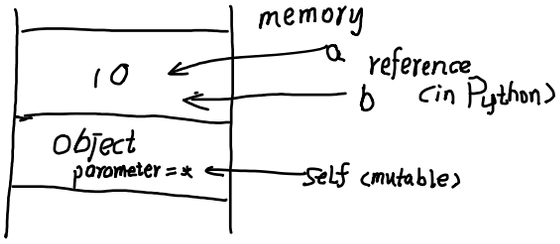
CSC 1001 Week VII, VIII & IX
Object-Oriented Programming (OOP)

Topic I Object

1. In Python, everything is an object (number, string, etc). ***Object*** is also a class name.

2. ***'id()'***: The id of an object is *an automatically assigned unique integer* by Python when the program is executed. It will not change during the execution of the program. ***'type()':*** The type for the object is determined by Python *according to the value of the object.*

3. Recognize the variable: A variable in Python is actually a reference to an object.



4. ***Methods:*** **(方法)**

You can perform operations on an object.
The operations are defined using functions.
The *functions* for the objects are *called methods* in Python.
Methods can only be *invoked from a specific object*.

5. **Why do we need object-oriented programming?**

- A sub-field in computer science called *software engineering* **(软件工程)** is invented to help with the development of large-scale software systems.
- People are always trying to invent *new ways of writing programs so that software development can be more efficient* – structural programming, OO programming, service oriented architecture, etc.
- Object oriented programing allows us to write program in a way that naturally match the problem that we are trying to solve.

Topic II Class

1. Object: An object represents an entity in the real world that can be distinctly identified. An object has a unique ***identity***, ***state/properties/attributes*** **(状态)**, and *behaviours* **(行为)**.

2. Key elements of an object:

- An object's identity is like a person's ID. Python automatically assigns each object a unique id for identifying the object at runtime.
- An object's state (also known as its properties or attributes) is *represented by variables*, called ***data fields***.

• Python uses *methods to define an object's behaviour*s (also known as its actions). Recall that methods are defined as functions. You make an object perform an action by invoking a method on that object.

3. **Class: (类)**

• Objects of the same kind are defined by using a common class

• The relationship between Classes and objects is analogous to that between an apple-pie recipe and apple pies

• A Python class uses variables to store data fields and defines methods to perform actions

• A class is a contract—also sometimes called a template or blueprint

4. Object v.s. class:

• An object is an *instance* of a class, and you can create many instances of a class.

• Creating an instance of a class is referred to as instantiation (**实例化**).

• The terms object and instance are often used interchangeably.

5. **Define class:**

class *ClassName:* (also can be written as 'class *ClassName(Object):*')

write data fields and other methods

6. **Constructing objects:**

• Once a class is defined, you can create objects from the class with a constructor.

• *'__new__()'*: The method *'__new__()'* is automatically invoked when an object is constructed. This method first creates an object in the memory, then invokes the *'__init__()'* method to initialize the object.

• *'__init__()'*: initialize the object. (i.e. create an object with given parameters)

7. **Self:**

• All methods, including the initializer, have the first parameter *self*.

• This parameter refers to the *object* that invokes the method.

• The self parameter in the *'__init__()'* method is automatically set to *reference the object* that was just created.

8. Constructor arguments:

• The arguments of the constructor match the parameters in the *'__init__()'* method *without self* (not one-to-one mapping in this case).

• The initializer in the class has a default parameter value, then the constructor *without arguments* will assign the default values to data fields.

```
class Person:
    def __init__(self, age=0, gender='Male'):
        self.age = age
        self.gender = gender
>>> a=Person(20,'Female')
>>> type(a)
<class '__main__.Person'>
>>> print(a)
The gender:Female; The age:20
>>> b=Person()
>>> print(b)
The gender:Male; The age:0
```

default value

arguments

9. Accessing member of objects:

- Data fields are also called ***instance variables***, because each object (instance) has *a specific value for a data field*.
- Methods are also called ***instance methods***, because a method which is invoked by an object (instance) will perform actions based on the data fields of that object.
- You can access the object's data fields and invoke its methods by using ***the dot operator (.)***, also known as the ***object member access operator***.

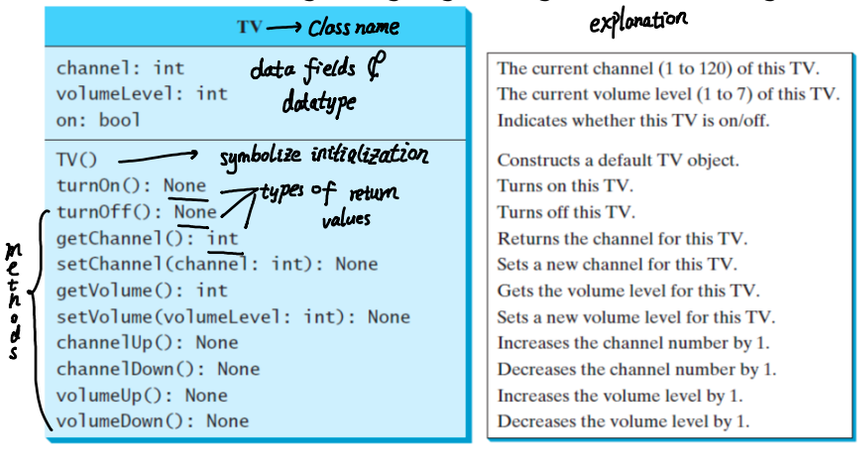
10. Scope of self:

- The scope of an instance variable is ***the entire class*** once it is created.
- You can also create local variables in a method.
- The scope of a ***local variable*** is ***within the method***.

Therefore, try to write all data fields in the overwritten '*__init__()*' method.

Topic III Understand & Use the OO Programming

1. Unified Modeling Language Diagram (UML diagram, **统一建模语言**):



The UML diagram does not show the principle of defining class.
However, it is enough for users to use the class.
It shows the advantage of OO programming.
('\' can separate one single line into two lines.)

2. Hiding data fields:

- ***Direct access*** of a data field in an object is not good.
- First, data may be ***tampered with.***
- Second, the class becomes ***difficult to maintain and vulnerable to bugs***.

3. ***Private data fields:***

- *Prevent* other programmers from directly accessing the data fields of your class is a common industrial practice, known as data hiding.
- In Python, the private data fields are defined with ***two leading/prefix underscores***, also applied for private methods.
- Private data fields and methods *can be accessed within a class*, but they cannot be accessed outside the class.

```
class Person:
    def __init__(self, age=0, gender='Male'):
        self.__age = age
        self.__gender = gender
    def getAge(self):
        return self.__age
>>> a=Person()
>>> a.__gender()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a.__gender()
AttributeError: 'Person' object has no attribute '__gender'
>>> a.getAge()
0
```

(private data fields)

• Define some methods to allow access to private data fields.

Topic IV Abstraction (抽象)

1. ***Abstraction*** means *separate the implementation* of a part of code *from* the *usage* of that code.

2. Why do we need abstraction?

• In software engineering, there are many levels of abstraction, a commonly used one is called ***function abstraction***. (low level)

• Function abstraction means separating the implementation of a function from its usage.

• Abstraction makes your code easy to ***maintain, debug and reuse***.

3. Class abstraction and encapsulation (**封装**):

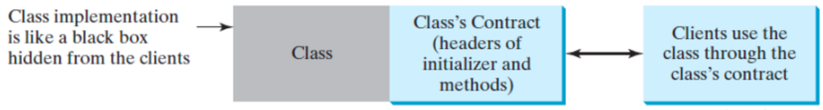
• Class abstraction means separating class implementation from the use of a class.

• The class implementation details are ***invisible*** from the user.

• The class's collection of methods, together with the description of how these methods are expected to behave, serves as the class's contract with the client.

• The user of the class does not need to know how the class is implemented. The details of implementation are ***encapsulated*** and hidden from the user, known as class encapsulation.

• In essence, encapsulation combines data and methods into a single object and hides the data fields and method implementation from the user.




```
>>> format((123-100)*100/123, '5.2f') + '%'
'18.70%'
```

```
>>> round(1.1)
1
>>> round(1.5)
2
>>> round(-3.8)
-4
```

4. ***'round(float number)':*** an integer (**取整**)

5. '***format(value, format_spec='', /)***':control output format of a floating number return as a string.

Topic V Inheritance (继承)

1. About the Inheritance

The object-oriented programming considers data and methods *together into objects.*

The object-oriented approach combines the power of the functional programming with an added dimension that integrates data with operations into objects (extension of functional programming).

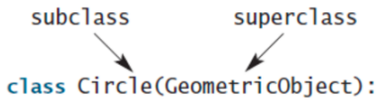
Object-oriented programming allows defining new classes from existing classes.

2. ***Superclass*** (**父类**) and ***Subclass*** (**子类**): (relative concepts)

• Inheritance enables you to define a general class (a superclass) and later extend it to more specialized classes (subclasses).

• You use a class to model objects of ***the same type***. Different classes may have some common properties and behaviours that you can generalize in a class.
• The specialized classes inherit the ***properties and methods*** from the general class.

3. The way to inherit:



class A(B): (A inherits B.)

__str__():

1) exist in every class (__...__() are Python built-in method, existing in Object Class—the superclass of all.);
2) not a private method;
3) ***print(variable name-'an object from a specific class')*** actually means printing the return value of ***__str__()*** method.

4. A subclass inherits ***accessible data fields and methods*** (cannot inherit private methods and data fields) from its superclass, but it can also have ***other data fields and methods***.

super().__init__(): use ***__init__()*** function from super class, aiming to inherit accessible data fields (methods are automatically inherited.).

```
def __init__(self, width = 1, height = 1):
    super().__init__()
```

5. Notes:

• A subclass is ***not a subset*** of its superclass; In fact, a subclass usually contains ***more information and methods*** than its superclass.
• Inheritance models is the ***'is-a relationships'***, but not all is-a relationships should be modelled using inheritance.

6. **Overriding methods (重写)**:

• Sometimes it is necessary for the subclass to ***modify the implementation*** of a method defined in the superclass. This is referred to as method overriding.
• The overriding way is to write the method ***the same name*** as super classes with ***different implementation***.

7.***The Object class:***

1) Every class in Python is ***descended from the object class***.
2) The object class is ***defined in the Python library***. If no inheritance is specified when a class is defined, its superclass is object by default.
3) Methods of the object class:

• The ***'__new__()'*** method is automatically invoked when an object is constructed. This method then invokes the ***'__init__()'*** method to initialize the object. Normally you should ***only override the '__init__()' method*** to initialize the data fields defined in the new class. (Never try to override ***'__new__()'*** method, or it will return ***'None'***.)
• The ***'__str__()'*** method returns a string description for the object. Usually you should override the ***'__str__()'*** method so that it returns an informative description for the object.
• The ***'__eq__()'*** method compares whether two objects are the same.

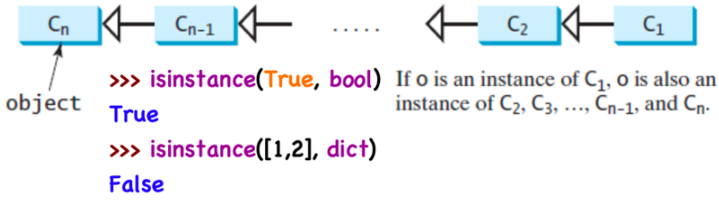
Topic VI Polymorphism (多态) and Dynamic Binding (动态绑定)

1. ***Polymorphism:***

- It means that the objects of different classes ***can be passed as arguments*** to the same function.
- A method may be implemented in several classes along ***the inheritance chain.***
- The inheritance relationship enables a subclass to inherit members from its superclass with additional new members.
- A subclass is a ***specialization*** of its superclass; every instance of a subclass is also an instance of its superclass, ***but not vice versa***.
- Therefore, when two objects share some common members, they can both be passed as arguments to the same function.

2. ***Dynamic Binding:***

- Python decides which method is invoked at runtime.
- Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class.
- In Python, C_n is the object class
- If o invokes a method p, Python searches the implementation for the method p in C_1, C_2, ..., C_{n-1}, and C_n, in this order, until it is found.



3. ***'isinstance(object, class)'*** function:

can be used to determine whether an object is an instance of a class.

****Private methods cannot be inherited and overridden.***

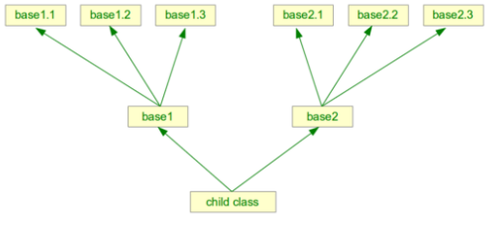
Topic VII Multiple Inheritance (多继承)

1. ***Multiple Inheritance:***

In Python, we can define ***a new class from multiple classes***.

Multiple inheritance is a feature in which a class can inherit data fields and methods from more than one parent class.

2. ***Inheritance Tree*** (继承树) :



3. The way to realize multiple inheritance:

```
class C(A, B):
    def __init__(self, a, b, c=300):
        A.__init__(self, a)
        B.__init__(self, b)
        self.c=c
```

The right way is not using super(), but directly using the superclass' name.

CSC 1001 Week X&XI Algorithm (算法)

Topic I Introduction to algorithm

1. Algorithm: a ***step-by-step procedure*** for performing some tasks in a finite amount of time.
2. The importance and properties of algorithm:
- Plays a key role in modern technological innovation.
- Not language dependent.
3. ***NP-hard problem:*** All current computers cannot solve this type of problems.

Topic II Justification of programs

1. The way to judge whether an algorithm is good or not:
- The primary analysis of algorithms involves characterizing the ***running times*** and ***space usage*** of algorithms and ***data structure operations***.
- ***Running time*** is a natural measure of "goodness," since time is a precious resource—computer solutions should run as fast as possible, it is also the **most important** factors in measuring.
- ***Space usage*** is another major issue to consider when we design an algorithm, since we only have limited storage spaces.

2. Two approaches for measuring:

1) ***Experimental Approach:*** directly use time module to measure the running time, visualizing the running time.

```
from time import time

startTime = time()
''' program body '''
endTime = time()

timeElapsed = startTime-endTime
```

Limitations of this approach:

(i) **Size of the input** dependence. (Experiments can be done only on a limited set of test inputs.)

(ii) **Hardware and Software Environments** dependence. (Other programs will affect the running time because of multi-tasking.)

(iii) An algorithm must be **fully implemented** in order to execute it to study its running time experimentally. (the worst disadvantage.)

2) ***Theoretical Approach:*** calculate the ***time*** complexity (**时间复杂度**) of a program

Principle 1: Counting **primitive operations**

✓The primitive operations can be variously defined by different programmers.

Principle 2: Measuring Operations as a **Function of Input Size**

✓Associate each algorithm with a function f (n) that characterizes **the number of primitive operations** that are performed as a function of **the problem size** n.

✓7 functions used in algorithm analysis, **which can cover all situations: Constant** (a legend); **Logarithm** (based on 2, i.e., a*logx = a*$\log_2$x), **Linear, N-log-N, quadratic, cubic and other polynomials, exponential** (cannot execute)

most

Principle 3: Focusing on the **Worst-Case** Input

✓If the worst case works, all other cases work.

✓The **big Oh notation** is used for estimating the upper bound, i.e., the worst case.

3. ***Asymptotic Analysis*** **(渐进分析)**:

In algorithm analysis, we focus on the **growth rate** of the running time as a function of the input size n, taking a "big-picture" approach

"Sweet spot" for high-level reasoning about algorithms, i.e., neglect the details such as the small number n, just consider when n goes to infinity.

Coarse enough to **supress unnecessary details**, e.g., architecture; language; compiler...

Sharp enough to make meaningful comparisons between algorithms.

Asymptotically better: means that better when n goes to infinity, i.e., the content in big Oh notation is smaller.

4. The line of tractability

- To differentiate efficient and inefficient algorithms, the general line is **between polynomial time algorithms and exponential time algorithms**.
- The distinction between polynomial-time and exponential-time algorithms is considered **a robust measure of tractability**.

Topic III Recursion

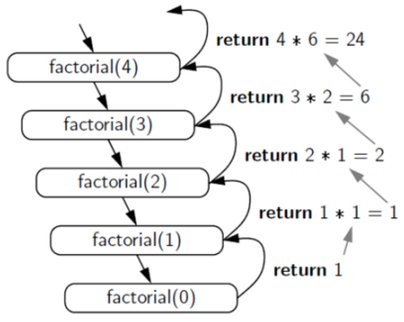
1. Introduction of **recursive algorithm**:

- Recursion is a technique by which a function makes **one or more calls to itself during execution**.
- Recursion provides an elegant and powerful alternative for **performing repetitive tasks**.

2. **The Recursive Definition:**

- Contents: (i) one or more ***base cases***, defined non-recursively in terms of fixed quantities; (ii) one or more ***recursive cases***, defined by appealing to the definition of the function being defined.
- The most important part is redefining and designing the function.

3. **The Recursive Tree:** (as right hand side shows)

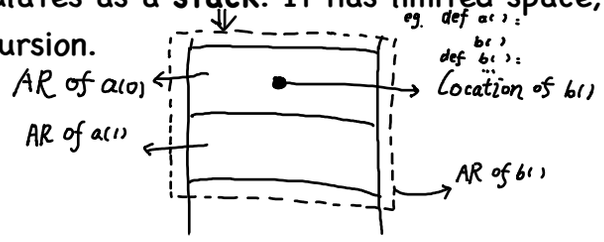


4. How Python implements recursion

- Each time a function (recursive or otherwise) is called, a structure known as an ***activation record*** or ***frame*** is created to store information about the progress of that invocation of the function
- This activation record stores the function call's parameters and local variables.

• When the execution of a function leads to a nested function call, the execution of the former call is suspended and its activation record stores the place in the source code at which the flow of control should **continue upon return of the nested call**.

• The activation records of the function cumulates as a **stack**. It has limited space, not supporting infinite/too many steps of recursion.



5. Summary types of recursion:

(i) **Linear Recursion**:

• Each invocation of the body makes **at most one** new recursive call.

• Finding the factorial and binary search are both linear recursive algorithms.

(ii) **Multiple recursion:**

• A function makes **two or more recursive calls**.

• Drawing the English ruler (showing in lecture notes) is a multiple recursion program.

Topic IV Searching Algorithm

1. ***Sequential Search*** (**顺序搜索**):

A common and standard way to search in an unsorted sequence.

Use a loop to examine every element, until either finding the target or exhausting the data set.

The complexity is **linear**.

2. ***Binary Search*** (**二分搜索**):

A classic and useful **recursive algorithm**, which can be used to efficiently locate a target value within a **sorted** sequence.

✓How to realize the binary search:

• Candidate – whether it matches the target is not sure.

• low and high – two maintained parameters, s.t. all the candidate entries have index at least low and at most high.

• low = 0 and high = n−1 initially, then compare the target value to the median candidate, then let mid = (low+high)/2.

• If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully.

If target < data[mid], then we recur on the first half of the sequence, that is, on the interval of indices from low to mid−1.

If target > data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high.

✓**Time complexity of binary search** runs in O(logn) time for a sorted sequence with n elements. (proof by hand)

```
def binarySearch(data, target, low, high):
    if low>high:
        print('Cannot find the target number!')
        return False
    else:
        mid = (low+high)//2
        if target==data[mid]:
            print('The target number is at position',mi
            return True
        elif target<data[mid]:
            return binarySearch(data, target, low, mid-1)
        else:
            return binarySearch(data, target, mid+1, high)
```

Consider every time the total program running as a primitive operation

$len = mid - low = \lfloor \frac{low+high}{2} \rfloor - low \leq \frac{high-low+1}{2} = \frac{1}{2}$ original length

$or = high - mid \leq high - \frac{low+high-1}{2} = \frac{1}{2}$ original length, $\Rightarrow$ total times r satisfying $\frac{n}{2^r} \leq 1$ & $\frac{n}{2^{r-1}} > 1$. $r = O(\log n)$

(consider the 8-queens problem and the recursive solution)

Topic V Sorting Algorithm

1. ***Bubble Sort*** (**冒泡排序**):

✓Procedure:

1) Iterate over a list of numbers, compare every element i with the following element i+1, and **swap them if i is larger.**

2) Iterate over the list again and repeat the procedure in step 1, but **ignore the last element** in the list.

3) Continuously iterate over the list, but each time ignore one more element at the tail of the list, **until there is only one** element left. (The last several numbers are already sorted.)

(*It can also be implemented by linked list.)

```
def bubble(bubbleList):
    listLength = len(bubbleList)
    while listLength > 0:
        for i in range(listLength - 1):
            if bubbleList[i] > bubbleList[i+1]:
                buf = bubbleList[i]
                bubbleList[i] = bubbleList[i+1]
                bubbleList[i+1] = buf
        listLength -= 1
    return bubbleList
```

✓Time complexity: quadratic time algorithm.

2. ***Quick sort*** (**快速排序**):

✓Procedure:

1) **Pivot**: an element from the array randomly picked.

2) **Partitioning**: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal

```
def quickSort(L, low, high):
    i = low
    j = high
    if i>=j:
        return L
    pivot = L[i]

    while i<j:
        while i<j and L[j]>=pivot:
            j = j - 1
        L[i] = L[j]

        while i<j and L[i]<=pivot:
            i = i + 1
        L[j] = L[i]
    L[i] = pivot

    quickSort(L, low, i-1)
    quickSort(L, j+1, high)
    return L
```

values can go either way). After this partitioning, the pivot is in its final position, whichis called the **partition operation.**

3) **Recursively apply** the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

(*It can also be complemented by linked list.)

✓Time complexity: quadratic time algorithm for the worst case, N-log-N time algorithm for the average cases.

TOPIC VI Sorting Algorithm for Tree

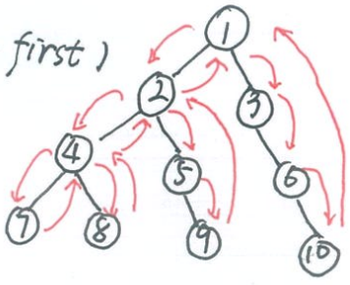
1. Depth First Search (DFS) over a tree:
(深度优先搜索)

one starts at the root and explores as deep as possible along each branch before backtracking

(For binary tree: from left to right; when deeper, deeper first)

(For unordered tree: randomly choose a child to start)

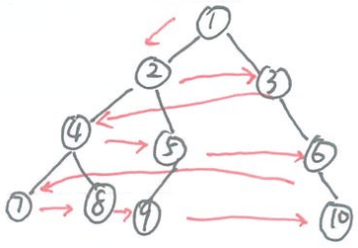
Example: search a path in a maze (迷宫)



2. Implement of DFS in a binary tree:

```
def DFSearch(t):
    if t:
        print(t.element)
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left)
        if t.right is not None:
            DFSearch(t.right)
```

} an algorithm using recursion



3. Breath First Search (BFS) over a tree:
(宽度优先搜索)

one starts at the root & we visit all positions at depth d before visiting all positions at depth d+1

Example: finding game's strategy — DFS can just find global optimal solution (hard to achieve)

BFS can find nearly optimal solution, eg. first several steps.

4. Implement of BFS in a binary tree:

```
def BFSearch(t):
    q = ListQueue()
    q.enqueue(t)
    while q.isempty is False:
        cNode = q.dequeue()
        if cNode.left is not None:
            q.enqueue(cNode.left)
        if cNode.right is not None:
            q.enqueue(cNode.right)
```

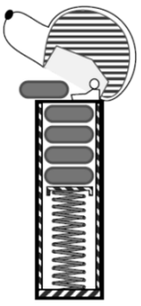
an algorithm using queue

CSC 1001 Week XII&XIII Data Structure

Topic I Introduction to Data Structure

1. ***Data Structure:*** a systematic way of **organizing and accessing** data.
2. Basic Types of Data Structure: **stack, queue, linked list, tree.**

Topic II Stack (堆栈)



1. **Stack**: a collection of objects that are inserted and removed according to the **last-in, first-out (LIFO)** principle.

A user may **insert** objects into a stack **at any time**, but may only **access or remove** the most recently inserted object that remains (at the so-called "top" of the stack).

2. Instances of using stack: web browser (history function); text editor ('undo' & 'redo' functions).
3. Implement of stack class: (use list)

S.push(e):	Add element e to the top of stack S.
S.pop( ):	Remove and return the top element from the stack S; an error occurs if the stack is empty.
S.top( ):	Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.
S.is_empty( ):	Return True if stack S does not contain any elements.
len(S):	Return the number of elements in stack S; in Python, we implement this with the special method __len__.

```
class ListStack:

    def __init__(self):
        self.__data = list()

    def __len__(self):
        return len(self.__data)

    def is_empty(self):
        return len(self.__data) == 0

    def push(self,e):
        self.__data.append(e)

    def top(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data[self.__len__()-1]

    def pop(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data.pop()
```

4. The usage of the stack:

Reverse a list using stack.

Brackets match checking.

Transfer a recursive algorithm to non-recursive ones.

Matching Tags in HTML Language.

(html language:

html tags: as right </h1>…<h1>)

- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

Topic III Queue (队列)

1. **Queue**: a collection of objects that are inserted and removed according to the **first-in, first-out (FIFO)** principle.

Elements can be inserted **at any time**, but only the element that has been in the queue **the longest** can be next removed.

2. Instances of using queue: service queue, call center.

3. Implement of queue class: (use list)

Q.enqueue(e): Add element e to the back of queue Q.

Q.dequeue( ): Remove and return the first element from queue Q; an error occurs if the queue is empty.

Q.first( ): Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.

Q.is_empty( ): Return True if queue Q does not contain any elements.

len(Q): Return the number of elements in queue Q; in Python, we implement this with the special method __len__.

```
class ListQueue:
    default_capacity = 5

    def __init__(self):
        self.__data = [None]*ListQueue.default_capacity
        self.__size = 0
        self.__front = 0
        self.__end = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size ==0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.__data[self.__front]
```

```
    def dequeue(self):

        if self.is_empty():
            print('Queue is empty.')
            return None

        answer = self.__data[self.__front]
        self.__data[self.__front] = None
        self.__front = (self.__front+1) \
                       % ListQueue.default_capacity
        self.__size -=1
        return answer

    def enqueue(self,e):
        if self.__size == ListQueue.default_capacity:
            print('The queue is full.')
            return None

        self.__data[self.__end] = e
        self.__end = (self.__end+1) \
                     % ListQueue.default_capacity
        self.__size += 1

    def outputQ(self):
        print(self.__data)
```

Static datafields **(静态)**: the variable outside every method, but inside the class.

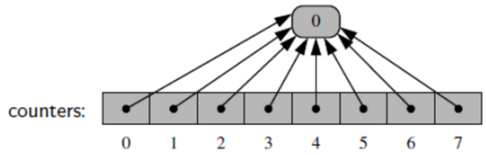
Topic IV Linked List (链表)

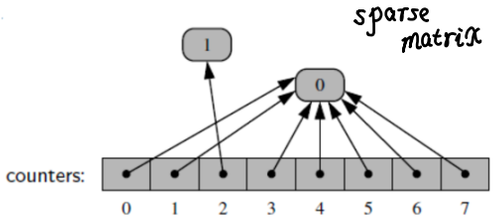
1. Referential Structure:

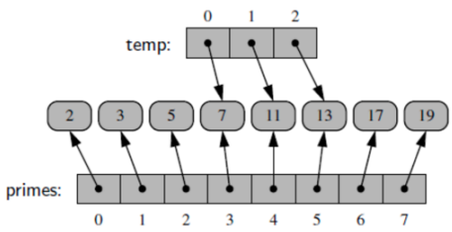
- List in Python is a referential structure.
- Consequently analyze.
- The new inserted one will have another location.
- The **order** in the list ≠ The order in the memory.
- The **order** in the compact array = The order in the memory

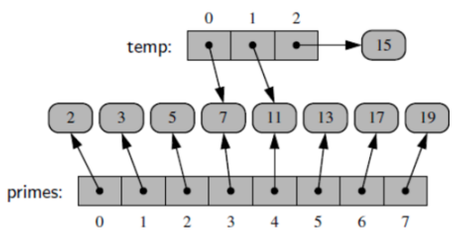
(works in C/C++/Java)

- The overall memory usage will be **much lower for a compact structure**





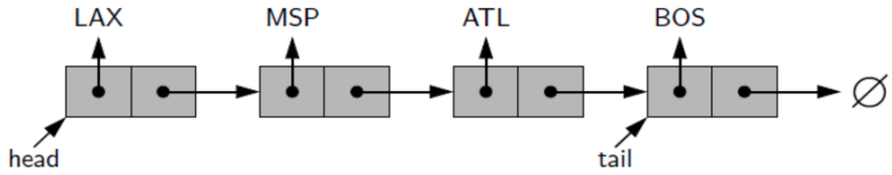




because there is no overhead devoted to the explicit storage of the sequence of memory references (in addition to the primary data).

2. ***Singly Linked List*** **(单链表)**:

- A singly linked list, in its simplest form, is a collection of **nodes (节点)** that collectively form a linear sequence.
- Each node stores a reference to an object that is **an element of the sequence**, as well as **a reference to the next node** of the list.
- The **first and last nodes** of a linked list are known as the ***head*** and ***tail*** of the list, respectively.
- We can identify the tail as **the node having None** as its next reference. This process is commonly known as ***traversing*** **(遍历)** the linked list.
- Because the next reference of a node can be viewed as a link or pointer **(指针)** to another node, the process of traversing a list is also known as ***link hopping or pointer hopping***.



- Implement of singly linked list class:

```
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

class LinkedList:

    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
```

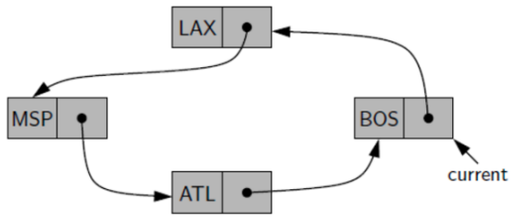
```
def add_first(self, e):
    newest = Node(e, None)
    newest.pointer = self.head
    self.head = newest
    self.size = self.size+1

    if self.size == 1:
        self.tail = newest
```

```
def add_last(self, e):
    newest = Node(e, None)
    if self.size > 0:
        self.tail.pointer = newest
    else:
        self.head = newest
    self.tail = newest
    self.size = self.size+1
```

```
def remove_first(self):
    if self.size == 0:
        print('The linked list is empty')
    elif self.size == 1:
        answer = self.head.element
        self.head = None
        self.tail = None
        self.size -= 1
        return answer
    else:
        answer = self.head.element
        self.head = self.head.pointer
        self.size = self.size - 1
        return answer
```

- About removing from the head:
In Python & Java, because of **memory judgement**, users can just change the pointer.
In C/C++, users need more steps to eliminate the head.
- Why not remove from the tail?
If so, it has **a linear time complexity**, while others are all constant.
(Because the program should first go through the whole list to find the last two.)

3. ***Circularly Linked List*** **(循环链表)**:



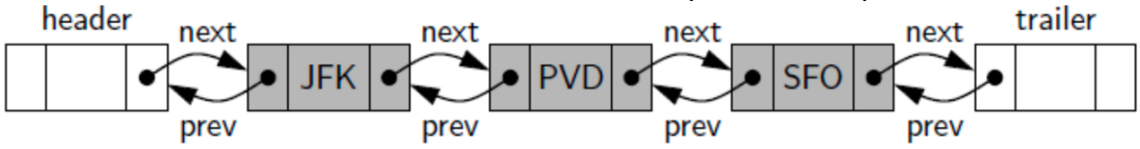
- The **tail** of a linked list can use its **next reference** to point back to the **head** of the list.
- Example: ***Round-robin scheduler***

A round-robin scheduler iterates through a collection of elements in a circular fashion and "serves" each element by performing a given action on it, often used to allocate **slices of CPU time** to various applications running concurrently on a computer. (Because CPU supports one program at one period.)

It can be achieved by using **circular queue**. (implemented by circularly linked list)

4. ***Doubly Linked List*** **(双向链表)**:

- A linked list in which each node keeps an **explicit reference** to the node **before** it and a reference to the node **after it**.
- In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add **dummy nodes/sentinels** at both ends of the list: **a header node at the beginning of the list, and a trailer node at the end of the list,** which do not store elements of the primary sequence.



- Implement of doubly linked list class

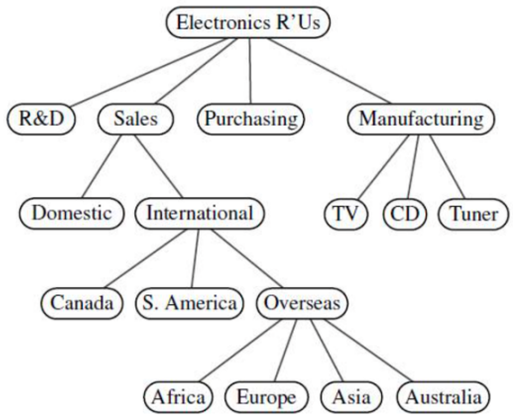
```
class Node:
    def __init__(self, element, prev, nxt):
        self.element = element
        self.prev = prev
        self.nxt = nxt

class DLList:

    def __init__(self):
        self.header = Node(None, None, None)
        self.trailer = Node(None, None, None)
        self.header.nxt = self.trailer
        self.trailer.prev = self.header
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0
```

```
def insert_between(self, e, predecessor, successor):
    newest = Node(e, predecessor, successor)
    predecessor.nxt = newest
    successor.prev = newest
    self.size+=1
    return newest

def delete_node(self, node):
    predecessor = node.prev
    successor = node.nxt
    predecessor.nxt = successor
    successor.prev = predecessor
    self.size -=1
    element = node.element
    node.prev = node.nxt = node.element = None
    return element

def iterate(self):
    pointer = self.header.nxt
    print('The elements in the list:')
    while pointer != self.trailer:
        print(pointer.element)
        pointer = pointer.nxt
```

Topic V Tree (树)

1. Introduction of Tree:

- A tree is a data structure that **stores elements hierarchically**.
- With the exception of the **top node/root (根节点)**, each element in a tree has a **parent node (父节点)** and zero or more **children nodes (子节点)**.



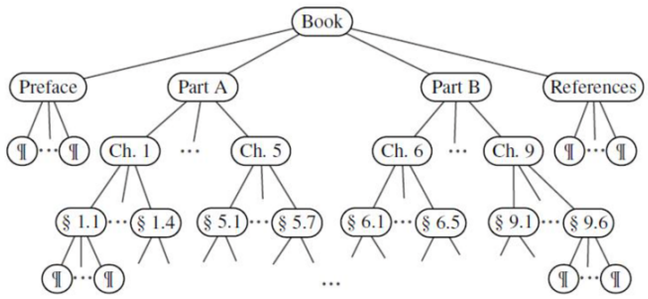
- Formally, we define **a tree T** as **a set of nodes storing elements** such that the nodes have a **parent-child relationship** that satisfies the following properties:

If T is nonempty, it has a special node, called the root of T, which has no parent.

Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

2. Definitions in Tree:

- ***Edge*** **(边)**: **a pair of nodes** (u, v) such that u is the parent of v, or vice versa.
- ***Path*** **(路径)**: a sequence of nodes such that any two consecutive nodes in the sequence form an edge.
- ***Depth*** **(深度)** of a node v: the length of the path **connecting root node and v**.
- ***Leaf Node*** **(叶节点)**: the nodes who have **no child**.
- ***Internal Node*** **(内部节点)**: the nodes who have **at least one child**.
- ***Ordered Tree*** **(有序树)**: There is a **meaningful linear order** among the children of each node; such an order is usually **visualized by arranging children** from left to right, according to their order.



- ***Binary Tree*** **(二叉树)**: an **ordered tree** with the following properties:

1) Every node has **at most two** children.
2) Each child node is labeled as being either a left child or a right child.
3) A left child precedes a right child in the order of children of a node.

- ***Subtree*** **(子树)** rooted at a left or right child of an internal node v is called a **left subtree** or **right subtree**, respectively, **of v**.
- ***Full Binary Tree:*** A binary in which each node has **either zero or two children**.

3. Implement of binary tree class:

(delete a Node--delete all the subtrees rooted in this Node.)

```
class Node:

    def __init__(self,element, parent = None,\
        left = None, right = None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right

class LBTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size
```

```
    def find_root(self):
        return self.root

    def parent(self,p):
        return p.parent

    def left(self,p):
        return p.left

    def right(self,p):
        return p.right

    def num_child(self,p):
        count = 0
        if p.left is not None:
            count+=1
        if p.right is not None:
            count+=1
        return count
```

```
def add_root(self,e):
    if self.root is not None:
        print('Root already exists.')
        return None
    self.size = 1
    self.root = Node(e)
    return self.root

def add_left(self,p,e):
    if p.left is not None:
        print('Left child already exists.')
        return None
    self.size+=1
    p.left = Node(e,p)
    return p.left
```

```
def add_right(self,p,e):
    if p.right is not None:
        print('Right child already exists.')
        return None
    self.size+=1
    p.right = Node(e,p)
    return p.right

def replace(self,p,e):
    old = p.element
    p.element = e
    return old

def delete(self,p):
    if p.parent.left is p:
        p.parent.left = None
    if p.parent.right is p:
        p.parent.right = None
    return p.element
```

CSC 1001 &1002 Tut 1

1. How to print things on separate lines?
Using Functions:
(i) ***'\n':*** a line break
(ii) ***'\t':*** same as 'Tab' on your keyboard, which is four blank spaces
(iii) ***'\x00':*** nothing but has a length
(iv) ***r' ':*** eliminate the impact of '\'
(v) ***''' ___ ''':*** docstring, which is to show the usage of one module or function. It is similar as #, but # is mainly used to explain specific code.

2. Difference between ***'if'*** and ***'=='***:
"==" is to compare ***values***, while ***"is"*** is to compare ***the memory address***. In python, small integers and short strings share the same address, so ***"is"*** may appear to be the same as ***"=="***, but for other types like tuple and list, two variables can have different address even with the same value.

3. For ***tuples***:
(i) List Indexing:
(ii) String Slicing & Indexing--***':'***
(iii) Add a third number--step number -- ***': :'***

```
a = [5, 10, 15, 20, 25, 30, 35, 40]
print(a[1:4]) # include the beginning, but not include the end
print(a[:])   # from the beginning until the very end
print(a[:-1]) # the last index can be either 7, or -1. it is not included.
```

```
>>> [10, 15, 20]
>>> [5, 10, 15, 20, 25, 30, 35, 40]
>>> [5, 10, 15, 20, 25, 30, 35]
```

```
a = [5, 10, 15, 20, 25]
print(a[0])  # count from 0
print(a[3])
```

```
>>> 5
>>> 20
```

```
a = [5, 10, 15, 20, 25, 30, 35, 40]
print(a[1:5:2])
print(a[3:0:-1])
```

```
>>> [10, 20]
>>> [20, 15, 10]
```

([1:5:2] -- means from the first to the fourth, every two)

(iv) ***'%'***: placeholder
'%d': accept a number, return an integer;
'%f': accept a number, return a float number;
'%s': accept a string or a number, return a string.

```
x = 1
y = 2.2
print("The length of %s is %d" % (x,y)
```

```
>>> The length of 1 is 2
```

Application:

```
s = list(range(100))
print(s)
# even numbers
print(s[::2])
# odd numbers
print(s[1::2])
# odd numbers in reverse order
print(s[::-2])
# even numbers in reverse order
print(s[-2::-2])
# last 10 numbers in reverse order
print(s[:-10:-1])
```

```
import string
s3 = string.ascii_lowercase
# string.ascii_uppercase: ABCDEFGHIJKLMNOPQRSTUVWXYZ
print("s3 is \n%s"%s3)
```

```
>>> s = '1 2 3 4 5'
>>> s[0]=0
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    s[0]=0
TypeError: 'str' object does not support item assignment
```

4. Are there any differences between strings and lists?
Strings are ***immutable (cannot be changed).***
Lists are ***changeable***.

```
>>> l = ['1','2','3','4','5']
>>> l[0] = '0'
>>> print(l)
['0', '2', '3', '4', '5']
```

5. ****String Methods:**
(i) ***'s.center()':*** put sth. in the center
(ii) ***'s.isdigit()':*** check whether a string is a digit (usually check user input)
(iii) ***'s.lower()':***

```
for i in range(1,12,2): print(('p'*i).center(11))
```

```
>>>       p
>>>      ppp
>>>     ppppp
>>>    ppppppp
>>>   ppppppppp
>>> ppppppppppp
```

5. **String Methods: (String Library)

(i) *'s.center()'*: put sth. in the center

```
for i in range(1,12,2): print(('p'*i).center(11))
>>>       p
>>>      ppp
>>>     ppppp
>>>    ppppppp
>>>   ppppppppp
>>> ppppppppppp
```

(ii) *'s.isdigit()'*: check whether a string is a digit (usually check user input)

(iii) *'s.lower()'*:

's.upper()':

```
S = "ONLY UPPER LETTERS"
print(S)
print(S.lower())

>>> ONLY UPPER LETTERS
>>> only upper letters
```

(iv) *'s.endswith()'*: check whether a string is ended with 'xx'. (another: *'s.startswith()'*)

(v) *'s.join()'*:

```
print('-'.join('new spam'))

>>> n-e-w- -s-p-a-m
```

(vi) *'s.rstrip()'*: strip characters from the end of the string

(vii) *'s.rjust()'*:

```
print('apple'.rjust(10,'-'))

>>> -----apple
```

(viii) *'s.find()'*:

```
s3 = 'one two three'
print(s3.rstrip('e'))      # from the right
print(s3.rstrip('er'))
print(s3.rstrip('erh'))
print(s3.rstrip('erht'))
print(s3.rstrip('erhto'))  # why this doesn't work? the space!
print(s3.rstrip('erht o')) # this one works

>>> one two thr
>>> one two th
>>> one two t
>>> one two
>>> one two
>>> one tw
```

```
print('apple'.find('app')) # return the begin index
print('apple'.find('le'))
print('apple'.find('c'))  # no 'c' in 'apple', return -1

>>> 0
>>> 3
>>> -1
```

give the first string index

(ix) *'s.split()'* and *'s.partition()'*: (functions & differences)

```
print('apple-orange-grape'.split('-'))  #return a list
print('apple-orange-grape'.partition('-'))  #return a tuple

>>> ['apple', 'orange', 'grape']
>>> ('apple', '-', 'orange-grape')
```

(x) *'s.join()'*:

```
print('-'.join(['apple','orange','grape']))

>>> apple-orange-grape
```

(xi) *'s.replace()'*:

```
s3 = 'one two three'
print(s3.replace('one','ten'))

>>> ten two three
```

two arguments

(if the to does not exist, will happen)

(xii) *'%'* and *'s.format()'*:

```
number = '1'
sentence = "This number is %s" % number
print(sentence)

number = '1'
sentence = "This number is {}".format(number)
print(sentence)

>>> This number is 1
>>> This number is 1
```

6. **List Operations:

(i) *'range()'* & *'list()'*: turn a range of numbers into a list.

```
print(range(10))
print(list(range(10)))

>>> range(0, 10)
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(ii) *'l.insert()'*: insert sth into a list at a certain index.

(iii) ***List Comprehensions***

Python supports a concept called "list comprehensions". It can be used to ***construct lists in a very natural, easy way,*** like a mathematician is used to do.

```
# Using list comprehension
>>> M = [x for x in S if x % 2 == 0]
>>> M
>>> [0, 4, 16, 36, 64]
```

```
# Using list comprehension
>>> V = [2**i for i in range(13)]
>>> V
>>> [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

Note: The '*list()***' function can turn a string into a list.

(iv) List can hold things in order, and we can change the order by ***'.sort()'*** method. Recall that lists are mutable, and ***'.sort()'*** method does change the contents of a list.

```
>>> L = [3, 1, 5, 9, 2]
>>> L.sort()
>>> L
[1, 2, 3, 5, 9]
>>> L = ['c', 'r', 'a', 'e', 'k']
>>> L.sort()
>>> L
['a', 'c', 'e', 'k', 'r']
```

(v) To learn more about lists, use '***dir()***' to see all the methods of list, and use '***help()***' to see the function and usage of each of them. Similar for strings.

```
>>> mylist = [1, 2, 3]
>>> dir(mylist)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Addition

(1) The number ('8') before 'd' in print('%8d'%v) means **8 spaces** for the integer v.

(2) The number ('8') before 's' in print('%8s'%v) means **8 spaces** for the string v.

(3) The parameters usually used in '***print()***' function is '***print(value, sep=xx, end=xx)***'

(4) More complex for ***float***:

➢print***('%7.3f'%v***): 7.3 means **7 spaces for the floating point number v (including the integer part, fractional part and the decimal point)** while keeping **3 digits for decimal part of v** (Rounding operation).

➢print***('%.3f'%v***): **.3 means keeping 3 digits for decimal part of v** (Rounding).

➢print***('%7.f'%v***): **Same as print('%7.0f'%v),** 7. means 7 spaces for the integer part while keeping 0 digits for decimal part of v (Rounding).

➢To **align the result leftward**, also add the symbol '-': print('%-7.2f'%v)

➢Only %9f , expressing print length of 9 (including decimal point)

%f express the initial value (5 digits after decimal point)

```
>>> v1='Jasmine'
>>> v2=4
>>> v3=12.5
>>> print('%s has %d petals and is worth %f yuan a bunch.'%(v1, v2, v3 ))
```

CSC 1001 &1002 Tut 2

1. The ***'split()'*** method will break up a string into a list of strings by the delimiter string (argument). Also, ***'list()'*** function can be a special case that splits a string into a list of single strings.

```
>>> s='abbbba'
>>> s.split('b')
['a', '', '', '', 'a']
```

2. The ***'join()'*** method returns a string where the string elements of sequence have been joined by separator.

```
>>> S = ['I', 'am', 'here.']
>>> ''.join(S)  ## .join func
'Iamhere.'
>>> ' '.join(S)  ## Add ' ' i
'I am here.'
>>> '**'.join(S)  ## Add '**'
'I**am**here.'
```

3. ***'.format()'***

```
>>> print('{1} {0}'.format(1, 2))
2 1
>>> print('{a} {b}'.format(a = 'c', b = 'y'))
c y
>>> print('{} {}'.format('one', 'two'))
one two
```

4. ***f'string'*** (faster)

```
>>> name = 'Xiaoming'
>>> institute = 'CUHK(SZ)'
>>> example = f'{name} is a from {institute}'
>>> print(example)
Xiaoming is a from CUHK(SZ)
```

5. Loop control: ***'pass'*** is mostly used as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The ***'pass'*** is silently ignored.

6. ***Variable-length arguments:***

Sometimes we may process a function for ***more arguments than we specified*** when defining the function. These arguments are called ***variable-length arguments*** and are not named in the function definition.

An ***asterisk (*)*** is placed before the variable name that holds the values of all non-keyword variable arguments (i.e., the un-named arguments). These argument values are ***stored in a tuple***, and thus are ***positional sensitive***. Additionally, ***this tuple remains empty if no additional arguments*** are specified during the function call.

```
def print_info(arg1, *vartuple):
    print("Output is: ")
    print(arg1)
    for var in vartuple:
        print(var)
    return
```

```
>>> print_info(10)
Output is:
10
>>> print_info(10,20,30)
Output is:
10
20
30
```

7. When ***two asterisks*** are placed before a variable name, the *****keywordargs*** form will grab the ***mapping information*** in the argument and behave like a dictionary.

```
def printdict(**kwargs):
    print(kwargs)
    return
```

```
# Call the function
>>> printdict(john=10, jill=12, david=15)
{'john': 10, 'jill': 12, 'david': 15}
```

8. Basic rules:

Always use ***lowercase for coding***, and ***avoid to use uppercase letters except for constants*** (They use UPPERCASE LETTERS like PI = 3.14159265)

For global variables, ***start with 'g_'***

For parameters of a function, ***start with 'p_'***

For a variable or a function with multi words inside, we ***use _ to separate them***.

9. ***Functions are Python objects.***

First of all, notice that functions are also objects in Python, and thus we are able to:

(1) assign a function as ***the value of a variable***.

(2) pass a function as ***a parameter*** of another function;

(3) return a function.

```
>>> def now():
...     print('current month: 2018-3')
...
>>> f = now
>>> f()
current month: 2018-3
```

10. Iteration is a general term for taking each item of something, one after another. Any time you use a loop, explicitly or implicitly, to go over a group of items, that is called iteration. In Python,

(1) An ***iterable object*** is an object that ***implements __iter__ (two underlines)***, which converts the iterable object to an iterator object.

(2) An iterator is an object that ***implements __next__ (two underlines) method***, which returns the next value from the iterator. If there are no more items to return, it should raise ***StopIteration exception***.

(3) Python has several ***built-in iterables*** such as, ***lists, tuples, strings, dictionaries and files***.

Whenever you use a for loop, or map, or a list comprehension, etc. In Python, the ***__next__ method is called automatically*** to get each item from the corresponding iterator, going through the process of iteration.

```
>>> L = iter([1, 2, 3])
>>> L.__next__() #L[0]
>>> L.__next__() #L[1]
>>> L.__next__() #L[2]
(Type one more, observe what happens)
>>> L.__next__() ——→ error
```

11. A special type of iterator: ***generator:***

It is similar to a list comprehension. But the difference is that a generator expression ***returns a generator, not a list.***

A generator expression is ***created with round brackets***. Creating a list comprehension in this case would be very ***inefficient*** because the example would occupy a lot of memory unnecessarily.

Therefore, after we create a generator, we usually do not prefer to call ***'next()'*** . Instead, we use ***a for loop to iterate it***. Meanwhile, there is no need to pay attention to the error about StopIteration.

```
>>> g = (x*x for x in range(3))
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    next(g)
StopIteration
```

12. Generator:

Generator is a 'function' that can be used to ***control the iteration behaviors*** of a loop. A generator is similar to a function returning an array. A generator can be used to ***generate a sequence of numbers***.

But unlike functions, which return a whole array, a generator ***yields one value at a time***. This requires ***less memory***. A generator returns an object on which you can call next, such that for every call it returns some value, until it raises a StopIteration exception.

Generators in Python:

Are defined with the '***def***' keyword

Use the ***'yield'*** keyword

May use several yield keywords Return an iterator

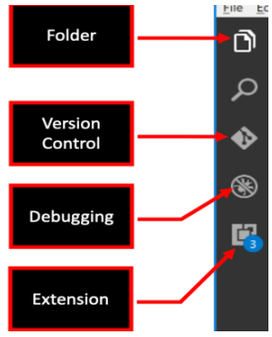
CSC 1002 SUMMARY

Topic 1 Python overview

1. IDE (Integrated Development Environment)

- uniform interface - editor
- intellisense - auto code completion, quick info, parameter info ...etc.
- code navigation - file explorer, code hierarchy, references, ...etc.
- debugging, source control, refactoring, unit testing ...etc.
- extensions (tools, language support, interfaces ...etc.

2. Activity Bar (as the right shows)



3. Core Data Types

Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None

4. List Comprehensions

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]

square_dict = { x : x * x for x in range(5) }  # { 0:0, 1:1, 2:4, 3:9, 4:16 }
square_set  = { x * x for x in [1, -1] }       # { 1 }

pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

5. zip & unpacking

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)        # is [('a', 1), ('b', 2), ('c', 3)]

pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

6. args & kwargs:

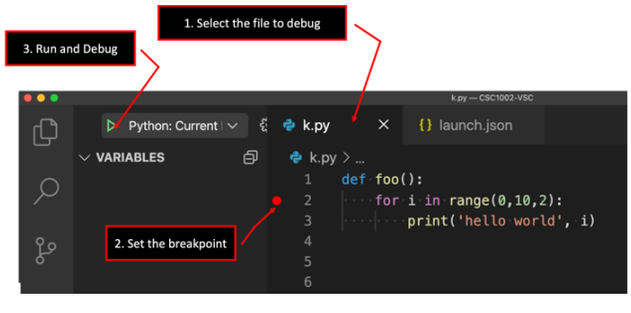
```
def magic(*args, **kwargs):
    print('unnamed args:', args)
    print('keyword args:', kwargs)

magic(1, 2, key = 'word', key2 = 'word2')
unnamed args: (1, 2)
keyword args: {'key': 'word', 'key2': 'word2'}
```

args = tuple of unnamed args

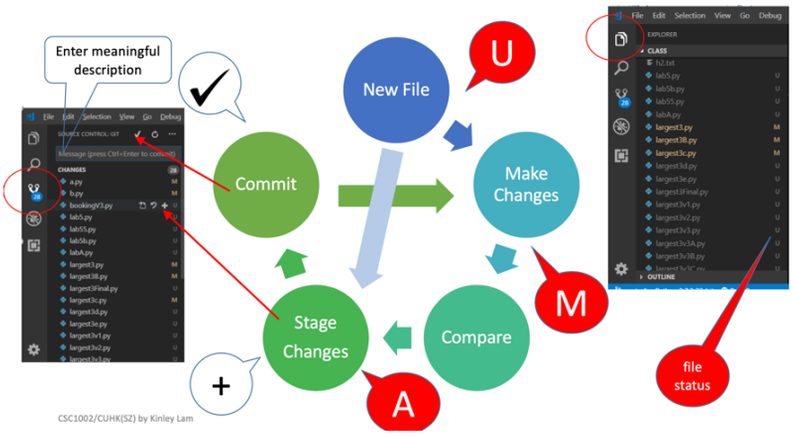
kwargs = dictionary list of named args

7. Editing

- AutoSave (especially if you have Source Control running)
- Settings - Display Language, Default Shell
- Explorer - Home Folder, Open Editors, Outline, Timeline, Script
- Status Bar - python version
- Editing - Split Windows, Zoom Level, Minimap, Column Selection
- Shortcuts
- Toggle comments
- Move + Copy line(s) up/down
- Run selected statement in terminal.



8. Version Control:



Topic II Turtle Graph

1. Turtle Class & some basic methods:

```
>>> import turtle
>>> t = turtle.Turtle()
>>> t.shape('turtle')
>>> t.color('green')
>>> t.forward(20)
>>> t.forward(100)
>>> t.backward(50)
>>> t.left(90)
>>> t.forward(30)
>>> t.goto(100,-50)
>>> t.up()
>>> t2 = turtle.Turtle()
>>> t2.shape('circle')
>>> t2.color('purple')
>>> t.clear()
>>> t.reset()
>>> t.goto(100,100)
>>> t.write('balck turtle', font=('times new roman', 20,'bold'))
>>> t.goto(-100,-100)
>>> t.shapesize(30)
>>> t.shapesize(5,5)
```

Others:

turtle.goto(x, y=None)
turtle.setpos(x, y=None)
{These are used to set the point's position.}

turtle.setx(x)
turtle.sety(y)
{These are used to set the x and y position.}

turtle.seth(to_angle)
{This is used to set the heading angle.}

turtle.home()

turtle.circle(radius, extent=None, steps=None)

Draw a circle with given radius. The center is radius units left of the turtle; extent – an angle – determines which part of the circle is drawn. If extent is not given, draw the entire circle. If extent is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if radius is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of extent.

As the circle is approximated by an inscribed regular polygon, steps determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

turtle.speed(speed=None)
"fastest": 0; "fast": 10; "normal": 6; "slow": 3; "slowest": 1.
If the number is larger than 10 or smaller than 0.5, it will be 0.
turtle.pos(); return the position x and y coordinates.
turtle.xcor() & turtle.ycor(); return the position x or y.
turtle.distance(*x, y=None*)

Programming Paradigm (with C++)

NOTEBOOK

WANG Yuzhe (Youthy)

May. 2020

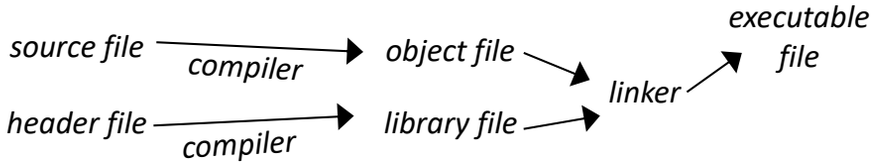
Chapter 1 Overview of C++ language

1. Comparison between C++ & Python:

	C++	Python
Execution	**Compiled** and linked into executable	**Interpreted** and executed by an engine
Memory Management	Requires much more attention to **bookkeeping** and **storage details**	Nearly no attention to be paid to memory management. **Supports garbage collection**.
Types	**Static typing**, explicitly declared, bound to names, **checked at compile time**	**Dynamic typing**, bound to values, **checked at run time**, and not so easily subverted
Language Complexity	**Complicated and full-featured**; give every language feature under the sun while at the same time never (forcibly) abstracting anything away that could potentially affect performance.	**Simple**; give only one or a few ways to do things, and those ways are designed to be simple, even at the cost of some language power or running efficiency

2. Compiler vs. Interpreter

1) A **compiler** takes entire program and converts it into object code which is typically stored in a file. The object code is also refereed as binary code and can be directly executed by the machine after linking. (e.g., C and C++.)
2) An **interpreter** directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. (e.g., Python and R.)
3) Java is a compiled programming language, but its source code is compiled to an intermediate binary form called **bytecode**. The bytecode is then executed by a Java Virtual Machine (JVM).
4) The compilation process:



5) IDE & Compiler:

- **IDE** (Integrated Development Environment): A software that provides comprehensive facilities to computer programmers for software development, and normally consists of
  - (i) a **source code editor**, which sometimes contains a **class browser**, an **object browser**, and a **class hierarchy diagram**, for use in object-oriented software development;
    GUI (Graphical User Interface) construction tools;
    a **version control system** (good for modifying)
  - (ii) build automation tools (must at least contain a **compiler** or an **interpreter**, or both, sometimes a **linker**, too)
  - (iii) a **debugger**.

3. Programs in C++

1) ***Comment:*** *Multi-line comments (*`/* Comments line 1 Comments line 2*/`*)*
Single-line comments (`// Comments line 1`*)*

2) ***Libraries*** *are collections of previously written tools that perform useful operations.*

3) ***Namespace****: code structures for ensuring that the names defined in different parts of a large system do not interfere with one another, every of which keeps track of its own set of names.*
The ***standard C++ libraries*** *use a namespace called* ***std****, which means that you cannot refer to the names defined in standard header files like "iostream.h" unless you let the compiler know to which namespace those definitions belong.*
To avoid always using the entire namespace, one can also use specific using declarations for just the names you need, e.g., "`using x::name1; using x::name2;`*"*

4) *Preprocessor directives are not program statements but directives for the* ***preprocessor****, which examines the code before actual compilation.*
Each directive has the following format:
   a. *the number/hash/pound character* ***(#)***
   b. *preprocessing instruction (one of* ***include****,* ***define****,* ***undef****,* ***if****,* ***ifdef****,* ***ifndef****,* ***else****,* ***elif****,* ***endif****,* ***line****,* ***error****,* ***pragma****)*
   c. *arguments (depending on the instruction)*
   d. *the new line character (line break)*

   *No semicolon (****;****) is expected at the end of a preprocessor directive, and one can extend a preprocessor directive through more than one line by preceding the newline character at the end of the line by a backslash (\).*

5) *Every C++ program must contain a function with the name* ***main****. It specifies the starting point for the computation and is called when the program starts up. When* ***main*** *has finished its work and returns, execution of the program ends. The* ***main*** *function can call other functions.*

(Remark: ***Imperative programming paradigm****: an explicit sequence of statements that change a program's state, specifying how to achieve the result.*
Structured: *Programs have clean,* ***goto****-free, nested control structures.*
Procedural: *Imperative programming with procedures operating on data.*
Object-Oriented: *Objects have/combine states/data and behavior/functions;*
Computation is effected by sending messages to objects (receivers).
{***Class-based****: Objects get their states and behavior based on membership in a class.*
Prototype-based*: Objects get their behavior from a prototype object.)*

6) *The "****declare-before-use****" rule: A C++ program consists of various* ***entities*** *such as variables, functions, types, and namespaces. Each of these entities must be* ***declared before they can be used****. The declaration of a function, called a* ***function prototype****, must appear before the* ***main*** *function.*

7) ***Data Types:*** *A data type is defined by a domain, which is the set of values that belong to that type, and a set of operations, which defines the behavior of that type.*

Although many data types are represented using **object classes** or other **compound structures**, C++ defines a set of **primitive types** to represent simple data.

`int`	This type is used to represent integers, which are whole numbers such as 17 or −53.
`double`	This type is used to represent numbers that include a decimal fraction, such as 3.14159265.
`bool`	This type represents a logical value (`true` or `false`).
`char`	This type represents a single ASCII character.

8) **Variables**: A **variable** is a named address for storing a type of value.
Each variable has the following attributes:
   a. A **name**, which enables you to differentiate one variable from another.
   b. A **type**, which specifies what type of value the variable can contain.
   c. A **value**, which represents the current contents of the variable.

The **address** and type of a named variable are fixed. The value changes whenever you assign a new value to the variable.

TABLE 1-1 Reserved words in C++

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

9) Variable **Scope** and **Extent**:
   a. **name binding**: an association of a name to an entity (e.g., data), such as a variable.
   b. **scope** of a name binding: the region of a computer program where the binding is valid, also known as the **visibility** of the entity.
   c. **extent** (or **lifetime**): describes when in a program's execution a variable has a (meaningful) value.
   d. Variables declared in the body of a function are called **local variables** and are accessible only inside that function; also sometimes called **automatic variables**; Variables declared as part of a class are called **instance variables**; Variables declared outside any function and class definition are **global variables**.

10) **Named Constants**: To create a named constant in C++, precede the regular variable declaration with the keyword `const`.
In C, the preprocessor directive `#define` was used to create a name for a constant.

11) **Binary operators** and **unary operator**: `(14/5 -> 2; double(14)/5 -> 2.8.`
The conversion is accomplished by means of a **type cast**, which you specify by using the type name as a function call.
If an expression contains more than one operator, C++ uses **precedence rules**.

TABLE 1-4 Operators available in C++

Operators organized into precedence groups	Associativity
`()` `[]` `->` `.`	*left*
unary operators: `-` `++` `--` `!` `&` `*` `~` (*type*) `sizeof`	*right*
`*` `/` `%`	*left*
`+` `-`	*left*
`<<` `>>`	*left*
`<` `<=` `>` `>=`	*left*
`==` `!=`	*left*
`&`	*left*
`^`	*left*
`|`	*left*
`&&`	*left*
`||`	*left*
`?:`	*right*
`=` *op*`=`	*right*

12) **L-value**: has an address, can appear on both sides of assignments.
R-value: no address, cannot appear on the left side of assignments.

13) The `?:` Operator:
`condition ? expression1 : expression2` evaluates the `?:` operator, if *condition* is **true**, evaluates *expression*$_1$; if *condition* is **false**, evaluates *expression*$_2$ instead.

14) The **switch** Statement:

```
switch ( expression ) {
  case v1:
     statements to be executed if expression = v1
     break;
  case v2:
     statements to be executed if expression = v2
     break;
  . . . more case clauses if needed . . .
  default:
     statements to be executed if no values match
     break;
}
```

15) The **for** Statement:

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

16) ***unit tests:*** create a test program that checks the correctness of that module in isolation from the rest of the code.

The text uses the `assert` macro from the `<cassert>` library to implement the unit-testing strategy. If the expression is `false`, the `assert` macro prints a message identifying the source of the error and exits from the program.

Additions to 10): The use of `const` and `static`:

For `const`:

(i) Constant definitions;

(ii) Constant call by reference:

Adding `const` to the declaration of a reference parameter signifies that the function will not change the value of that parameter.

(iii) Constant methods:

Adding `const` after the parameter list of a method guarantees that the method will not change the object. (`int CharStack::size() const`)

Classes that use the `const` specification for all appropriate parameters and methods are said to be `const`-correct.

For `static`:

Applied to	Meaning
a local variable	The variable is "permanent", in the sense that it is initialized only once and retains its value from one function call to the next. It is like having a global variable with local scope.
a global constant	Since a global constant has internal linkage by default (unlike global variables, which have external linkage by default), meaning it is only available for use in the file in which it is defined (and is therefore, in effect, static const).
a global variable or a free function	The scope of the function, or the variable, is limited to the file in which it is defined. Without a static qualifier, any free function or global variable in a file has the extern qualifier by default, making it visible from other files in the compilation unit.
a member variable of a class.	There is only one such variable for the class, no matter how many objects of the class are created. In other words, it turns the member variable from an "instance variable" into a "class variable".
a member function of a class.	The function may access only static members of the class. That is, it may not access any instance members (since there may not be any, if no objects of the class have been created).

Chapter II Functions and Libraries

II.1 The Procedural Programming Paradigm

1) Defn: A ***programming paradigm*** is a "style" or "way" of programming.

2) A language allowing programming in many paradigms is called a ***multi-paradigm*** programming language, e.g., C++, Python. You can write programs or libraries that are largely ***procedural***, ***object-oriented***, or ***functional*** (i.e., some typical paradigms).

3) About C++: C++ supports the ***procedural*** and ***object-oriented*** paradigms naturally, supports the ***functional*** paradigm through the `<functional>` interface, and supports many other paradigms through various external libraries.

II.2 The Idea of Functions in C++:

1) Advantages: ***shorten a program; easier to read; simplify maintenance*** (by decomposition).

2) ***Function Prototypes***: (see in Chapter I)
If you always define functions before you call them, prototypes are not required. (e.g., the low-level functions come at the beginning, followed by the intermediate-level functions that call them, with the main program coming at the very bottom.)

3) C++ Enhancements to Functions:

Overload: can define several different functions with the same name as long as the correct version can be determined by looking at ***the number and types of the arguments***.

The pattern of arguments (i.e., the number and types of the arguments but not the parameter names) required for a particular function is called its ***signature***.

Giving ***default values*** to the parameters makes them ***optional:*** The specification of the default values (***default arguments***) appears only in the function prototype and not in the function definition. Any optional parameters *must appear at the end* of the parameter list.

4) The Mechanics of Calling a Function:
When you invoke a function, the following actions occur:

1. Evaluates the argument expressions in its own context.
2. Copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a ***stack frame***
3. Evaluate the statements in the function body, using the new stack frame to look up the values of local variables.
4. When encountering a `return` statement, it computes the return value and substitutes that value in place of the call.
5. Discards the stack frame for the called function and returns to the caller, continuing from where it left off.

II.3 Libraries:

1) Libraries:

1. In computer science, a ***library*** is a collection of ***non-volatile*** resources used by computer programs, often to develop software.
2. A library is also a collection of ***implementations*** of behavior, written in terms of a language, that has a well-defined ***interface*** by which the behavior is invoked.

3. Libraries can be viewed from two perspectives. Code that uses a library is called a ***client***. The code for the library itself is called the ***implementation***.
4. The point at which the ***client*** and the ***implementation*** meet is called the ***interface***, which serves as both a barrier and a communication channel: Sharing and reuse; Abstraction and hiding.

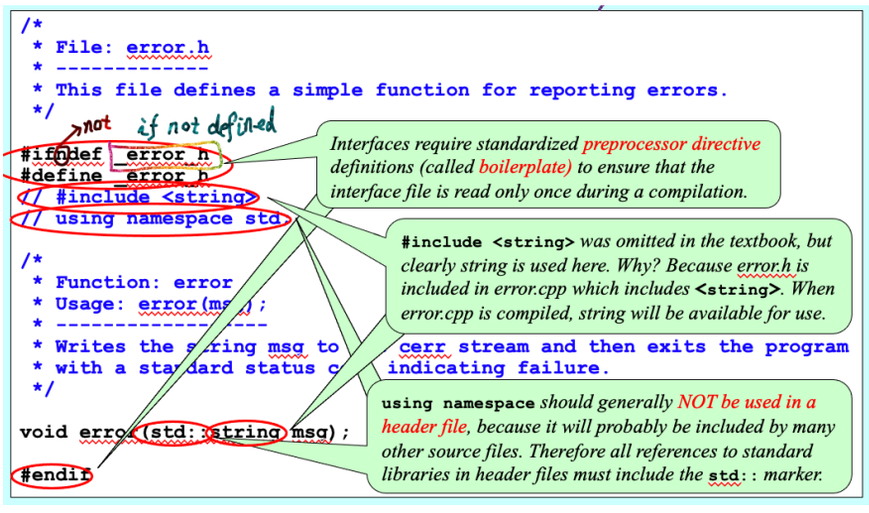
2)About ***C++ Standard Libraries***:

1. A collection of ***classes*** and ***functions***, which are written in the core language and part of the C++ ISO Standard itself.
2. Features of the C++ Standard Library are declared within the ***std namespace***: (a) Containers: vector, queue, stack, map, set, etc; (b) General: algorithm, functional, iterator, memory, etc; (c) Strings; (d) Streams and Input/Output: iostream, fstream, sstream, etc; (e) Localization; (f) Language support; (g) Thread support library; (h) Numerics library; (i) C standard library: cmath, cctype, cstring, cstdio, cstdlib, etc.
3. About ***Stanford Libraries***: include but not limit to

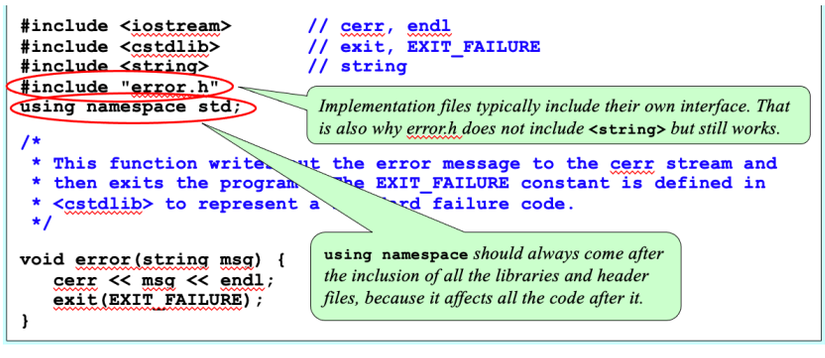
console.h	Redirects standard input and output to a console window.
error.h	Supports error reporting and recovery.
gwindow.h	Implements a simple, portable, object-oriented graphical model.
simpio.h	Suoports improved error-checking for input operations.
strlib.h	Extends the set of string operations.

3) Creating & Implementing Library Interfaces:

In C++, libraries are made available to clients through the use of an interface file that has the suffix **.h**, which designates a ***header file***, as illustrated by the following **error.h** file:



In C++, the header file contains only the prototypes of the exported functions. The implementations of those functions appear in the corresponding **.cpp** file:



Overloading operators: (example)

`std::ostream & operator<<(std::ostream & os, Direction dir);` *(Overloads the << operator so that it is able to display Direction values.)*

`Direction operator++(Direction & dir);` *(Overloads the **prefix** version of the ++ operator to work with Direction values.)*

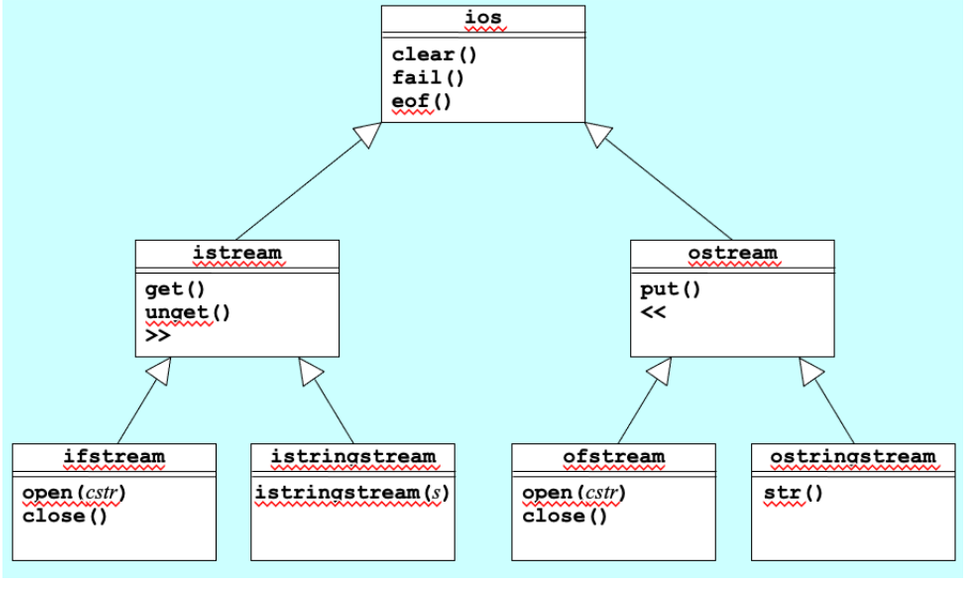
`Direction operator++(Direction & dir, int);` *(Overloads the **suffix** version of the ++ operator to work with Direction values.)* *to distinguish*

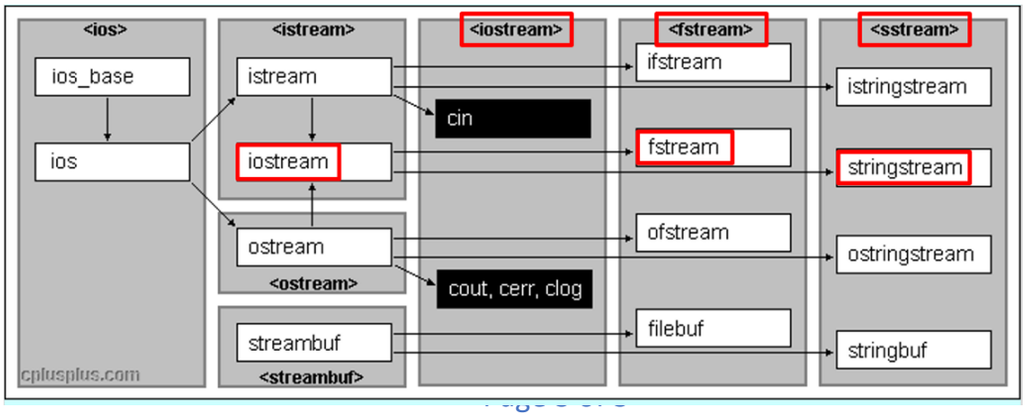
4) Principles of Interface Design:

1. ***Unified.*** *Every library should define a **consistent abstraction** with a clear unifying theme. If a function does not fit within that theme, it should not be part of the interface.*
2. ***Simple.*** *The interface design should simplify things for the client. To the extent that the underlying implementation is itself complex, it must **seek to hide that complexity**.*
3. ***Sufficient.*** *For clients to adopt a library, it must provide functions that **meet their needs**.*
4. ***General.*** *A well-designed library should be **general enough** to meet the needs of many different clients and can be **used in many different situations**.*
5. ***Stable.*** *The functions defined in a class exported by a library should **maintain** precisely the same structure and effect, even as the library evolves.*

5) Library vs. Class Hierarchy

1. *Interestingly, the C++ stream libraries are **not** organized based on the class hierarchies.*
2. *Organize libraries for the users' convenience, and design classes for the data integrity and implementation efficiency.*





Chapter III Strings and Streams

III.1 Characters in C and C++:

1) Conceptually, a ***string*** is simply ***a sequence of characters***, which is precisely how strings are implemented in C. C++ supports a higher-level view of strings, as ***objects***. The different strategies used by C and C++ on strings show the differences between different ***programming paradigms***.

2) Both C and C++ use ***ASCII*** (***American Standard Code for Information Interchange***) as their encoding for character representation. The data type `char` therefore fits in ***a single eight-bit byte*** (Totally, 2^8 = 256.).

3) In most modern language, ASCII has been superseded by ***Unicode***, which permits a much larger number of characters. The C++ libraries define the type `wchar_t` to represent "wide characters" that allow for a larger range.

4) Declaration of Single Character:

```
char ch;
char ch = 'a';
```

```
char ch = 9;   // '\t' (Tab)  (how much space depends on editors)
char ch = 10; // '\n' (New line)
```

5) The `<cctype>` (`ctype.h`) Interface [c(from C library)ctype(name of the library)]:
Including `"bool isdigit(char ch)"; "bool isalpha(char ch)"; "bool isalnum(char ch)"; "bool islower(char ch)"; "bool isspace(char ch)"; "char toupper(char ch)";`

III.2 C String Literal vs. C++ String Object:

1) ***C string literal*** (***const char[]*** type): got by putting double quotation marks around a sequence of characters. (e.g., `char cstr[] = "hello";`). The characters are stored in an ***array*** of bytes, terminated by a ***null byte*** whose ASCII value is 0.
(e.g.,

h	e	l	l	o	,		w	o	r	l	d	\0
0	1	2	3	4	5	6	7	8	9	10	11	12

).

2) The `<cstring>` (`string.h`) interface offers a lot of functions to operate C strings. (e.g., `int len = strlen(cstr); strcpy(cstr, "world");`). When using `strcpy`, pay attention to the space.
However, users cannot directly assign a value to a C string except for initialization. (Note that `sizeof x` returns the actual memory size of variable `x` including '`\0`', but `strlen(x)` do NOT.)

3) ***C++ String Object*** (a higher-level view of strings as ***objects***): library `<string>` provides a convenient ***abstraction*** for working with strings of characters by making `string` a ***class***.
Some operators in `string` class:

`str[i]`
Returns the i^{th} character of `str`. Assigning to `str[i]` changes that character.

`str.c_str()`
Returns a C-style character array containing the same characters as `str`.

Unlike most languages, C++ ***allows classes to redefine*** the meanings of the standard operators.

4) A comparison:

```
#include <cstring>      // the C string library, used in C++
#include <string.h>     // the C string library, used in C, acceptable in C++
#include "string.h"     // some compilers might still find the C string library, unless you defined string.h by yourself, which will cause conflict
#include <string>               // the C++ string library
```

```
#include "cstring.h"   // incorrect unless you defined cstring.h by yourself
#include <cstring.h>   // most likely an error even if you defined cstring.h by yourself
```

*5) **Concatenation** and C Strings: C++ automatically converts a **C string literal** to a **C++ string object** whenever the compiler can determine that what you want is a C++ string object: (e.g.,* `string str = "hello" + string(", ") + "world";` *But* `string str = "hello" + ", " + string("world");` *Is NOT allowed because + operator cannot be applied to C string literals.)*

*6) Strings as an Abstract Data Type: C++ allows clients to change the individual characters contained in a string. By contrast, Python strings are **immutable**.*

`str.erase(pos, count)`	←*Destructively changes* `str`
Deletes `count` characters from `str` starting at position `pos`.	
`str.insert(pos, text)`	←*Destructively changes* `str`
Inserts the characters from `text` into `str` starting at position `pos`.	
`str.replace(pos, count, text)`	←*Destructively changes* `str`
Replaces `count` characters in str with `text` starting at position `pos`.	

III.3 Formatted Input & Output:

Formatted input/output:

fprintf	Write formatted data to stream (function)
fscanf	Read formatted data from stream (function)
printf	Print formatted data to stdout (function)
scanf	Read formatted data from stdin (function)
snprintf C++11	Write formatted output to sized buffer (function)
sprintf	Write formatted data to string (function)
sscanf	Read formatted data from string (function)
vfprintf	Write formatted data from variable argument list to stream (function)
vfscanf C++11	Read formatted data from stream into variable argument list (function)
vprintf	Print formatted data from variable argument list to stdout (function)
vscanf C++11	Read formatted data into variable argument list (function)
vsnprintf C++11	Write formatted data from variable argument list to sized buffer (function)
vsprintf	Write formatted data from variable argument list to string (function)
vsscanf C++11	Read formatted data from string into variable argument list (function)

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

III.4 C++ Streams:

1) Introduction:

*(i) **Stream** is an important **object/abstraction** that represents **an input source or output destination of characters of indefinite length**, which C++ uses to manage the flow of information to or from some data source.*

(ii) Streams are often associated to a physical source or destination of characters, like a disk file (file streams), the keyboard (standard input stream `cin`*), or the console (standard output stream* `cout`*), so the characters gotten from or written to our abstraction called stream are physically input from or output to the corresponding physical device.*

2) Standard input/output stream: `cin` *and* `cout`*:*

*(i) The standard technique to specify formatted output in C++ uses the **insertion operator**, which is written as* `<<`*. This operator takes **an output stream on the left** and **an expression of any type on its right**. The insertion operator **returns the output stream** as its result. The advantage of this interpretation is that output operations can be **chained together**. C++ allows you to control the output by including items in the output chain called **manipulators** (* `<iomanip>` *), which affect the way how subsequent values are formatted.*

`endl`	Moves cursor to the next line.
`setw(`*n*`)`	Sets the width of the next value to *n* characters.
`setprecision(`*digits*`)`	Sets how many digits should appear.
`setfill(`*ch*`)`	Sets the fill character used to pad values.
`left`	Aligns the value at the left edge of the field.
`right`	Aligns the value at the right edge of the field.
`fixed`	Sets fixed-point output (no scientific notation).
`scientific`	Sets scientific-notation output.
`showpoint/noshowpoint`	Controls whether a decimal point must appear.
`showpos/noshowpos`	Controls appearance of a plus sign.
`uppercase/nouppercase`	Controls whether uppercase is used in hex.
`boolalpha/noboolalpha`	Controls whether `bools` appear as `true`/`false`.

(ii) For input, C++ includes the **>>** operator, which is called the ***extraction operator***. The **>>** operator is symmetrical to the **<<** operator and ***reads formatted data from the stream on the left into the variables that appear on the right***. Users can use ***manipulators*** to affect the way how subsequent values are formatted:

skipws **noskipws**	These manipulators control whether the extraction operator skips over whitespace characters before reading a value. If you specify **noskipws**, the extraction operator treats all characters (including whitespace characters) as part of the input field. You can later use **skipws** to restore the default behavior. This property is persistent.
ws	Reads characters from the input stream until some character appears that is not in the whitespace category. The effect of this manipulator is therefore to skip over any spaces, tabs, and newlines in the input. Unlike **skipws** and **noskipws**, which change the behavior of the stream for subsequent input operations, the **ws** manipulator takes effect immediately.

3) Data Files:

(i) A ***file*** is the generic name for any named collection of data maintained on the various types of ***permanent*** storage media attached to a computer. The most common type of file is probably a ***text file***, which contains character data of the sort you find in a string.

(ii) Using text files in C++: Construct a new **ifstream** (i.e., ***input file stream***) object that is tied to the data in the file by declaring a stream variable to refer to the file. Call the **open** method to open the file. For historical reasons, the argument to open is a ***C string literal*** rather than a ***C++ string object***. Break the association between the reader and the file by calling the stream's **close** method, which is called ***closing the file***.
Data in files are usually read and written ***sequentially***.

(iii) Reading Characters:

```
int ch;                                   // int get();
while(( ch =infile.get()) != EOF) {
    . . . Perform the necessary operations using the character . . .
}
```

```
char ch;                                  // istream& get(char& c);
while (infile.get(ch)) {
    . . . Perform the necessary operations using the character . . .
}
```

Type casting a stream to bool

```
file.open(filename.c_str());
file.get(c); // or c = file.get();
c = toupper(c);   → move to 2nd char
file.put(c);
```

Users can also read lines from a text file by calling the ***free function*** (unlike a ***method***, a free function is not bound to a particular class) **getline**, which takes an **ifstream** and a **string** as reference parameters.

```
istream& getline(istream& is, string& str, char delim);
istream& getline(istream& is, string& str);
```

```
string line;
while (getline(infile, line)) {
   if (line.length() > max) max = line.length();
}
```

(iv) RAM v.s. Storage: strings are stored in RAM (random access memory), will be stored temporarily; Storage stores sth permanently.

4) String Streams: A string stream is a special stream that lets you use a string as the source and destination of the flow of data. C++ provides that capability through the **<sstream>** (**istringstream**, **ostringstream**) library.

```
stringstream ss;
ss << "My favorite number is: " << i
cout << ss.str();
```

5) Copy Contents between Streams

```
void copyStream(istream & is, ostream & os) {
   char ch;
   while (is.get(ch)) {
      os.put(ch);
   }
}
```

This version is obviously better because it can be directly used in the above program.

Chapter IV Classes and Collections

IV.1 The Collection Classes:

1) Abstract Data Type (ADT): The **atomic/primitive data types** like `bool`, `char`, `int`, `double`, occupy the lowest level in the data structure hierarchy. A type defined in terms of its behavior rather than its representation is called an **Abstract Data Type (ADT)**.

2) The classes that contain collections of other objects are called **containers** or **collection** classes.
Classes that include a base-type specification are called **parameterized classes**.
The **Standard Template Library** (**STL**) is a software library for the C++ programming language that influenced many parts of the **C++ Standard Library**. **Template**s are sometimes called static (or compile-time) **polymorphism**.

3) The Stanford Collection Classes: `Vector` `Grid` `Stack` `Queue` `Map` `Set` `Lexicon`
Each class (except **Lexicon**) requires **type** parameters. To avoid copying, these structures are usually passed **by reference**.

4) The `Vector<`*type*`>` Class: `#include "vector.h";` `Vector< Grid<char> > chessPositions;`
the `Grid<`*type*`>` Class works kind of like the 2-d vector. (`Stack<`*type*`>` and `Queue<`*type*`>` skipped.)

5) The `Map<`*type, type*`>` Class: A **map** is conceptually similar to a dictionary in real life (and in Python). `Map<`*key type*, *value type*`> map;`
Iterating over a collection: C++11 uses a **range-based for statement** to simplify iterators
`for (string key : map) {` . . . *code to process that key* . . . `}`

6) The `Set<`*type*`>` Class: model the mathematical abstraction of a **set**, which is a collection in which the elements are **unordered** and **no duplicate**. `Set` supports several mathematical operations (via **operator overloading**).

7) The `Lexicon` Class: A set of words with no associated definitions is called a **lexicon**.
`Lexicon english("EnglishWords.txt");` also supports a space-efficient precompiled binary format (.dat) of text files.

IV.2 Class and Objects:

1) Compound Data Types in C++:
(i) **Enumerated** types: An enumerated type can be roughly considered as a subset of `int` with special names for each value in the subset. (Values can be also assigned artificially.)

```
enum Direction { NORTH, EAST, SOUTH, WEST };
```
0 1 2 3

(ii) **Structures**: **compound** values in which the individual components are specified by name.

```
struct typename {
    declarations of fields
};
```

This definition creates a type, not a variable. This definition allows you to declare a **variable** like this:

```
struct Point {
    int x;
    int y;
};

Point pt;
```

2) Classes and Objects: Object-oriented languages are characterized by representing most data structures as **object**s that encapsulate both the representation (i.e., **fields** or **instance variables**) and the behavior of the objects (i.e., a set of **methods**) in a single entity.

3) The Format of a Class Definition: **public** section (available to clients); **private** section (restricted to implementation).

```
class typename {
public:
    prototypes of public methods

private:
    declarations of private instance variables
    prototypes of private methods
};
```

4) Implementing Methods: A class definition usually appears as a `.h` *file that defines the* ***interface*** *for that class. Although methods can be implemented within the class definition, it is stylistically preferable to define a separate* `.cpp` *file that hides those details. To define methods, users should write the name of the class before the name of the method. (e.g.,* `Point::toString`*).*

5) Getters and Setters: Methods that retrieve the values of instance variables are formally called ***accessors****, but are more often known as* ***getters****. Methods that set the values of specific instance variables are called* ***mutators*** *or, more informally,* ***setters****. For safety consideration,* ***setter*** *methods are far less common than* ***getters*** *in object-oriented design.*

6) Constructors: class definitions typically include one or more ***constructors****, which are used to* ***initialize*** *an object. The prototype for a constructor has no return type and always has the same name as the class. A single class can have multiple constructors (i.e., overloading). The constructor that takes no arguments is called the* ***default constructor****.*

Initializer List: contains names of fields in the class; superclass constructors.

```
Point(int xc = 0, int yc = 0) {
   x = xc;
   y = yc;
}
```

```
Point(int xc = 0, int yc = 0) : x(xc), y(yc) {}
```

IV.3 Overloading Operators:

1) Besides ***methods****, you can also implement some useful* ***operators****, even those existing ones in C++. Each operator is associated with a name that usually consists of the keyword* `operator` *followed by the operator symbol.*

2) Since stream variables cannot be copied, the `ostream` *argument must be* ***passed by reference****.*
The `<<` *operator has a chaining behavior of returning the output stream, therefore, the definition of* `operator<<` *must also return its result by reference.*

Other operators that can be overloaded:

```
ostream & operator<<(ostream &os, Point pt) {
   return os << pt.toString();
}
```

cout

any of the following operators: `+` `-` `*` `/` `%` `^` `&` `|` `~` `!` `=` `<` `>` `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<` `>>` `>>=` `<<=` `==` `!=` `<=` `>=` `<=>` (since C++20) `&&` `||` `++` `--` `,` `->*` `->` `( )` `[ ]`

3) Two approaches: (the method-based style) (the free function approach)

```
// in point.h, inside the class Point definition
bool operator==(Point pt);
// in point.cpp
bool Point::operator==(Point pt) {
   return x == pt.x && y == pt.y;
}
```

`p1` *is the receiver and the argument* `p2` *is copied to parameter* `pt`*. The private data of a class are private to the class and not to the object.*

```
// in point.h, inside the class Point definition
friend bool operator==(Point p1, Point p2);
// in point.h, outside the class Point definition
bool operator==(Point p1, Point p2);
// in point.cpp
bool operator==(Point p1, Point p2) {
   return p1.x == p2.x && p1.y == p2.y;
   return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}
```

Prototype

What if there are no public getters?

(i) ***class methods*** *(object-oriented programming style):*
(advantage) ***free access*** *to the private instance variables and other methods;*
(disadvantage) the left operand is the receiver object the right operand is a parameter.

(ii) ***free functions*** *(procedural programming style).*
(advantage) ***easier to read*** *(the operands are both passed as parameters as usual);*
(disadvantage) must be designated as a ***friend*** *to refer to the private data.*

```
// in rational.h, inside the class Rational definition
friend Rational operator+(Rational r1, Rational r2);
friend Rational operator-(Rational r1, Rational r2);
friend Rational operator*(Rational r1, Rational r2);
friend Rational operator/(Rational r1, Rational r2);
// in rational.h, outside the class Rational definition
Rational operator+(Rational r1, Rational r2);
Rational operator-(Rational r1, Rational r2);
Rational operator*(Rational r1, Rational r2);
Rational operator/(Rational r1, Rational r2);
```

Chapter V Analysis of Algorithms

V.1 Recursion:

1) Dividing it into smaller sub-problems of **the same form** is the essential characteristic of recursion; without it, all you have is a description of top-down design/stepwise refinement. **Recursion** is a special kind of divide and conquer.

In terms of the structure of the code, the defining characteristic of recursion is having **functions that call themselves**, directly or indirectly, as the decomposition process proceeds.

2) Some examples:

(i) Improve efficiency of the recursive design of Fibonacci sequence:

```
int fib(int n) {
   if (n < 2) {
      return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

An extremely inefficient design (exponential time complexity)

improve

```
int fib(int n) {
   return additiveSequence(n, 0, 1);
}
int additiveSequence(int n, int t0, int t1) {
   if (n == 0) return t0;
   if (n == 1) return t1;
return additiveSequence(n - 1, t0, t1) + additi
}                              veSequence(n - 2, t0, t1);
```

(ii) **Mutual** Recursion: need to decide the initial cases for all!!

```
bool isOdd(unsigned int n) {
   if (n == 0) return false;
   return isEven(n-1);
}

bool isEven(unsigned int n) {
   if (n == 0) return true;
   return isOdd(n-1);
}
```

```
bool isOdd(unsigned int n) {
   if (n == 1) return true;
   return isEven(n-1);
}

bool isEven(unsigned int n) {
   if (n == 0) return true;
   return isOdd(n-1);
}
```

(iii) The Subset-Sum Problem: follow the **inclusion-exclusion pattern**.

```
bool subsetSumExists(Set<int> & set, int target) {
   if (set.isEmpty()) {
      return target == 0;
   } else {
      int element = set.first();
      Set<int> rest = set - element;
      return subsetSumExists(rest, target)
          || subsetSumExists(rest, target - element);
   }
}
```

(iv) Graphical Recursion: In graphical recursion, there seems to be **no simple cases** to be solved. There must be, however, **some stopping criteria** to stop the recursion.

```
void subdivideCanvas(GWindow & gw, double x, double y,
                                   double width, double height) {
   if (width * height >= MIN_AREA) {
      if (width > height) {
         double mid = randomReal(MIN_EDGE, width - MIN_EDGE);
         subdivideCanvas(gw, x, y, mid, height);
         subdivideCanvas(gw, x + mid, y, width - mid, height);
         gw.drawLine(x + mid, y, x + mid, y + height);
      } else {
         double mid = randomReal(MIN_EDGE, height - MIN_EDGE);
         subdivideCanvas(gw, x, y, width, mid);
         subdivideCanvas(gw, x, y + mid, width, height - mid);
         gw.drawLine(x, y + mid, x + width, y + mid);
      }
   }
}
```

3) Thinking recursively: a new way of thinking about problems that focuses more on a **holistic view** of the problem than on the details by adopting the **recursive leap of faith**. Avoid **nonterminating recursion** (stack overflow), the recursive analogue of the infinite loop.

V.2 Backtracking Algorithms:

1) ***Backtracking algorithms***: working through a sequence of decision points, each of which leads further along some path. When reaching a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, backtrack to previous decisions.

A backtracking problem has a solution if and only if at least one of the smaller backtracking problems that result from making each possible initial choice has a solution

2) Example: (solving a maze)

```
enum Direction {
   NORTH, EAST, SOUTH, WEST
};

struct Square {
   bool marked;
   bool walls[4];
};

Grid<Square> maze;
Point startSquare;
bool mazeShown;
double x0;
double y0;
int rows;
int cols;

bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjacentPoint(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

3) Searching in a Branching Structure:

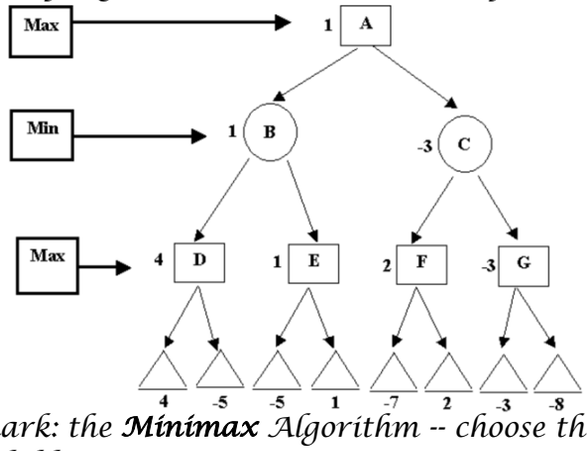
The method is applied in a wide variety of applications, characterized by the need to explore a range of possibilities at each of a series of choice points.

The historical knowledge of what choices have already been tested and which ones remain for further exploration is maintained automatically in the ***execution stack***.

4) A generalized program for two-player games:

```
struct Move {
   int nTaken;
};

void play() {
   initGame();
   while (!gameIsOver()) {
      displayGame();
      if (getCurrentPlayer() == HUMAN) {
         makeMove(getUserMove());
      } else {
         Move move = getComputerMove();
         displayMove(move);
         makeMove(move);
      }
      switchTurn();
   }
   announceResult();
}
```

5) Game Trees: As Shannon observed in 1950, most two-player games have the same basic form



Remark: the ***Minimax*** Algorithm -- choose the move that minimizes the maximum rating available to your opponent

6) AlphaGo's algorithm: a combination of ***machine learning*** and ***Monte Carlo tree search*** techniques, guided by a "***value network***" and a "***policy network***", both implemented using ***deep neural network*** technology and combined with extensive training.

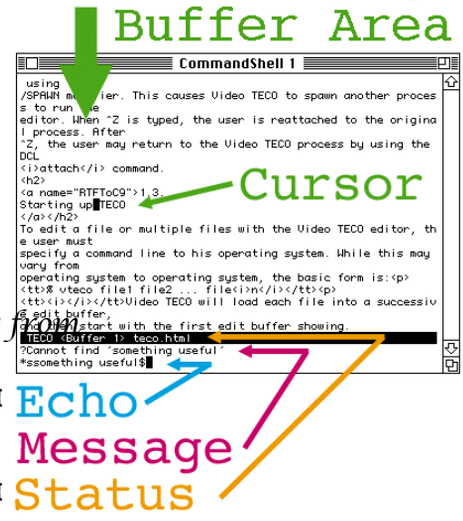
V.3 Algorithm Analyses:

1) O (bounded above) vs. Ω (below) vs. Θ (both): e.g., the worst case of Quicksort is actually both $O(N^2)$ and $\Omega(N^2)$, therefore $\Theta(N^2)$ too.

2) Some trade-offs and compromises:
- Time complexity vs. Space complexity
- In-place (at most $O(1)$ extra space) vs. Not-in-place
- Stable vs. Unstable
- Serial vs. Parallel

3) An example: `EditorBuffer` Class Design

(i) Way I: A simple ***array model*** using dynamic allocation.
(Hint: Distinguish the ***allocated size*** (`capacity`) of the array from its ***effective size*** (`length`); need to expand capacity)




```
void EditorBuffer::expandCapacity() {
char *oldArray = array;
capacity *= 2;
array = new char[capacity];
for (int i = 0; i < length; i++) {
   array[i] = oldArray[i];
}
delete[] oldArray;
}
```

Way II: A ***linked-list model*** that uses pointers to indicate the order.
(Hint: Need a ***dummy cell*** for placing cursor; represent the position of the cursor by pointing to the cell before the insertion point)

Way III: A ***two-stack model*** that uses a pair of character stacks.
(Hint: The characters before the cursor are stored in a stack called `before` and the characters after the cursor are stored in a stack called `after`.)

(ii) Efficiency Comparison:

F	**moveCursorForward()**	$O(1)$	$O(1)$	$O(1)$
B	**moveCursorBackward()**	$O(1)$	$O(N)$	$O(1)$
J	**moveCursorToStart()**	$O(1)$	$O(1)$	$O(N)$
E	**moveCursorToEnd()**	$O(1)$	$O(N)$	$O(N)$
I	**insertCharacter(ch)**	$O(N)$	$O(1)$	$O(1)$
D	**deleteCharacter()**	$O(N)$	$O(1)$	$O(1)$

4) Amortized Analysis: In an ***amortized analysis***, we average the time required to perform a sequence of data-structure operations over all the operations performed.
Amortized analysis differs from average-case analysis in that probability is **not** involved; an amortized analysis guarantees the average performance of each operation in the worst case.

(e.g., Mathematics of the Doubling Strategy:
If α is the cost of an insertion that doesn't expand the capacity and β is the per-element cost of the linear expansion, the total cost of inserting N items is:

$$\text{total time} = \alpha N + \beta\left(N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots\right)$$

$$\text{average time} = \alpha + \beta\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$$

The sum in parentheses is always less than two, so the average time of the worst case is constant.
)

Chapter VI Pointers and Dynamic Memory Management

VI.1 The Structure of Memory:

1) The fundamental unit of memory inside a computer is called a ***bit***, which is a contraction of the words binary digit. A bit can be in either of two states, usually denoted as **0** and **1**.

In most modern architectures, the smallest addressable unit on which the hardware operates is a sequence of ***eight consecutive bits*** called a ***byte***.

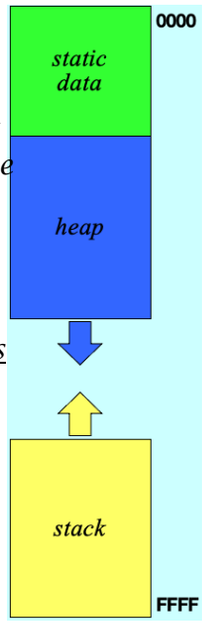
The unit that represents the most common integer size on a particular hardware (i.e., the number of bits the CPU can process at one time) is called a (hardware) ***word***. It may vary from machine to machine (e.g., a word in x86-64, the 64-bit version of the x86 instruction set, is 64-bit.)

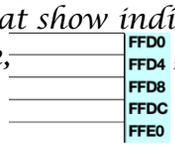
2) A word is usually the largest piece of data that can be transferred to/from the memory in a single operation of a particular processor.

The largest possible ***address length***, used to designate a location in memory, is typically a word, because this allows one memory address to be efficiently stored in one word.

However, computers can have memory addresses larger or smaller than their ***word size***.

In theory, modern byte-addressable ***N-bit*** computers can address 2^N bytes of memory. but in practice the amount of memory is limited by the CPU, the memory controller, etc.



3) Every byte inside the primary memory of a machine is identified by a numeric address. Memory diagrams that show individual bytes are not as useful as those that are organized into words. (like, 

, it includes four bytes (i.e., a 32-bit machine) in each of the memory cells.)

4) The Allocation of Memory to Variables:

One region of memory is reserved for program code and global variables/constants that persist throughout the lifetime of the program. This information is ***static data***.

Each time you call a method, C++ allocates a new block of memory called a ***stack frame*** to hold its local variables. These stack frames come from a region of memory called the ***stack***.

It is also possible to allocate memory dynamically. This space comes from a pool of memory called the ***heap***.

In classical architectures, the stack and heap grow toward each other to ***maximize the available space***.

VI.2 Pointers and Arrays:

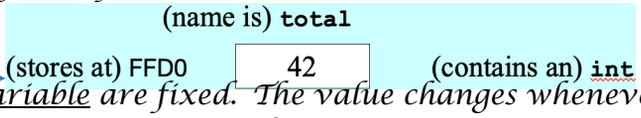
1) Data Types in C++: inherits from C, containing (i) ***Atomic (primitive) types*** (`short`, `int`, `long`, their `unsigned` variants, `float`, `double`, `long double`, `char`, `bool`); (ii) Enumerated types; (iii) Structure types; (iv) ***Array***s of some base type; (v) ***Pointer***s to a target type.

1 byte (8 bits)	2 bytes (16 bits)	4 bytes (32 bits)	8 bytes (64 bits)	16 bytes (128 bits)
`char` `bool`	`short`	`int` `float`	`long` `double`	`long double`

Enumerated types are typically assigned the space of an `int`. Structure types have a size equal to the sum of their fields. Arrays take up the element size times the number of elements. Pointers take up the space needed to hold an ***address***, which is usually the size of a hardware word. (e.g., 4 bytes on a 32-bit machine and 8 bytes on a 64-bit machine)

`sizeof(t)` returns the **actual number of bytes** required to store a value of the type `t`; `sizeof x` returns the actual memory size of the variable `x`.

2) Variables: `int total = 42;`



The **address** and **type** of a named variable are fixed. The value changes whenever you assign a new value to the variable. Note that `42` here (as **rvalue**) is in codes, but not in the memory.

Lvalue: In C++, any expression that refers to an internal memory location capable of storing data is called an **lvalue**. Once being declared, the address of an **lvalue** never changes, even though the contents of those memory locations change. The address of an lvalue is a value of a pointer variable, and can be stored in memory and manipulated as data.

3) **Pointer**: a data item **whose value is an address in memory**, which can be manipulated just like any other kind of data.

(i) Diagrams that involve pointers are typically represented in two different ways:
- Using memory addresses emphasizes the fact that pointers are just like integers.
- Conceptually, it often makes more sense to represent a pointer as an arrow.

(ii)Several Purposes Pointers Serve:
- Pointers allow you to refer to a large data structure in a compact way. (**call by pointers**).
- Pointers make it possible to reserve new memory during program execution. (**dynamic allocation**).
- Pointers can be used to record relationships among data items. (**linked structures**).

4) Declaration of a Pointer Variable:

```
type*   var;
type *  var;
type   *var;
```

(of which `type` it points to, `type` for pointer itself is `int`; the pointer itself occupies some memory space.)

- Pointer Operators:
  - the **address-of operator** (`&`) is written before a variable name (or any expression to which you could assign a value, an lvalue) and **returns the address** of that variable. (Note: the address-of operator is NOT a reference (when declaring).)
  - the **value-pointed-to operator** (`*`) is written before a pointer expression and returns the **actual value of a variable** to which the pointer points (**dereferencing**).

```
double x = 2.5;
double * px = &x;
```

```
double y = *px;
```

5) **Call by Pointer** vs. **Call by Reference** vs. **Call by Value** in C++:

- **Reference** in C++: NOT support in the original C language.
  1. A reference variable is an alias, that is, an alternative identifier for an already existing variable.
  2. The `&` characters in these declarations indicate reference variables.
  3. A reference must be initialized when it is declared. Once a reference is initialized with a variable, it cannot be reassigned to refer to a different variable.
  4. **Call by value** v.s. **Call by reference**

```
int n1 = 1, n2 = 2;
int & x = n1, & y = n2;
```

(i) C++ allows callers and functions to share information using a technique known as **call by reference**.
(ii) C++ indicates call by reference by adding an ampersand (`&`) before the parameter name. A single function often has both **value parameters** and **reference parameters**.
(iii) If you left out the `&` characters in the parameter declarations (i.e., **call by value**), calling this function would have no effect on the calling arguments because the function would exchange local copies.

```
void swap(int & x, int & y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

(iv) **Call by reference** can return more than one value.
(v) **Call by reference** has two primary purposes:
  - It creates a sharing relationship that makes it possible to pass information in both directions through the parameter list.

– It increases efficiency by eliminating the need to copy an argument.

5. In C++, a reference is a simple reference datatype that is **less powerful but safer** than the **pointer type** inherited from C. It can be considered as **a new name** for an existing object, but **not a copy** of the object it refers to.

❖ Pointer vs. Reference

	Pointer	Reference
Definition	The memory address of an object	An alternative identifier for an object
Declaration	`int i = 5;` `int * p = &i;`	`int i = 5;` `int & r = i;`
Dereferencing	`*p`	`r`
Has an address	Yes (`&p`)	No (no such `&r`)
Pointing/referring to nothing	Yes (`NULL`/ `nullptr` since C++11)	No (`r = j` means value of `i` change)
Reassignments to new objects	Yes	No
Supported by	C and C++	C++

6) Pointers to Objects: users need to dereference the pointer before invoking the method of objects, because "." takes precedence over "*", we have `(*pp).getX();`, where "->" is an alternative way. `pp->getX();`

The Keyword `this`: It is often **convenient** to use the same names for parameters and instance variables. Doing so, users must use the keyword `this` (defined as a pointer to the current object) to refer to the instance variable.

```
Point::Point(int x, int y) {
    this->x = x;
    this->y = y;
}
```

7) Simple Arrays in C++: the only operation is selection using `[]`, and arrays don't store their actual length. Thus, to determine how many elements there are in a strange array, we use `sizeof MY_ARRAY / sizeof MY_ARRAY[0]`.

The size of the array specified in the declaration is called the ***allocated size***. The number of elements actively in use is called the ***effective size***.

8) Pointers and Arrays:

❖ In C++, the name of an array is synonymous with a pointer to its first element. For example, if you declare an array `int list[100];` the C++ compiler treats the name `list` as a pointer to the address `&list[0]` whenever necessary, and `list[i]` is just `*(list+i)`, because pointer arithmetic **counts the objects** pointed to by the pointer.

❖ Although an array is often treated as a pointer, they are NOT entirely equivalent. E.g., you can assign an array to a pointer (of the same type), but not vice versa, because an array is a **non-modifiable** lvalue (so are constant variables)

VI.3 Pointer Arithmetic:

1) C++ defines the + and - operators so that they work with pointers. Dangerous, though. C++ defines pointer addition so that the following identity always holds (Note the following are not C++ statements!)

```
double arr[5];
double * dp = arr;
```

arr[i] ≡ *(arr+i) ≡ *(dp+i) ≡ dp[i]
&arr[i] ≡ arr+i ≡ dp+i ≡ &dp[i]

2) Comparison between different Element types:

```
int a[] = {0, 1, 2, 3};
int * p = a; // &a[0]
```

	a	&a	&a[0]	p
Type	array or used as pointer to an int	address of an array	address of an int	pointer to an int
Size of	16 (4 int)	8 (a word)	8 (a word)	8 (a word)
Lvalue	Yes (non-modifiable)	No	No	Yes
Value	ADDRESS1	ADDRESS1	ADDRESS1	ADDRESS1
*	0	ADDRESS1	0	0
&	ADDRESS1	N/A	N/A	ADDRESS2
+1	ADDRESS1 + 1 int	ADDRESS1 + 4 int	ADDRESS1 + 1 int	ADDRESS1 + 1 int

(jump over the full array)

Rule: Interpret `a` *as an entire array first. If it doesn't make sense, interpret it as a pointer.*

3) C Strings are Pointers to Characters:

```
char* msg = "hello, world";
```

```
char cstr[] = "hello, world";
char* msg = cstr;
```

4) The `*p++` *idiom:*

(i) `*p++` *is equivalent to* `*(p++)`*, because unary operators in C++ are evaluated in right-to-left order.*

(ii) The `*p++` *idiom means dereference* `p` *and return as an lvalue the object to which it currently points, and increment the value of* `p` *so that the new* `p` *points to the next element in the array. (Note the differences between it and* `*++p`*).*

(iii) It is, however, equally important that you avoid using it in your own code, to avoid ***buffer overflow errors****.*

VI.4 Dynamic Allocations:

1) Dynamic Allocation and Freeing Memory:

C++ uses the `new` *operator to allocate memory on the heap. Users can either allocate a single value (as opposed to an array) or allocate an array of values.*

```
int * pi = new int;          type * name = new type[size];
```

- *The* `delete` *operator frees memory previously allocated. For arrays, you need to include empty brackets, as in*

```
delete pi;
delete [] arr;
```

  - *Always pair a* `delete` *to a* `new`*, and a* `delete []` *to a* `new []`*.*
  - *No* ***size*** *needs to be specified for* `delete []`*. One of the approaches for compilers know how much memory to free is to allocate a little more memory and to store a count of the elements in a head segment (invisible to you) just before the first array element.*
  - `delete ptr` *only frees the memory space pointed by* `ptr`*, but not the memory space occupied by* `ptr` *itself.* `ptr` *is still a live variable until it is released (depending on how it is declared), though it is* ***dangling****. (i.e., does not point to a valid object of the appropriate type).*
  - *To avoid dangling pointers, after deleting a pointer (freeing the memory space it points to), one can* ***nullify*** *a pointer by*

```
ptr = NULL; // nullptr since C++11
```

2) Garbage Collection:

Programs that fail to free the heap memory you allocate have what computer scientists call ***memory leaks****.*

In Python and Java, objects are created on the heap and are automatically reclaimed by a ***garbage collector*** *when those objects are no longer accessible.*

Garbage collection frees the programmer from manually dealing with memory deallocation, eliminating or substantially reducing:

- Certain kinds of memory leaks (forget to `delete`);
- Dangling pointer bugs (forget to free the pointers of the same variable);
- Double free bugs (delete pointers of the same variable "twice");
- Efficient implementations of persistent data structures, etc.

Garbage collection, however, is not always efficient:

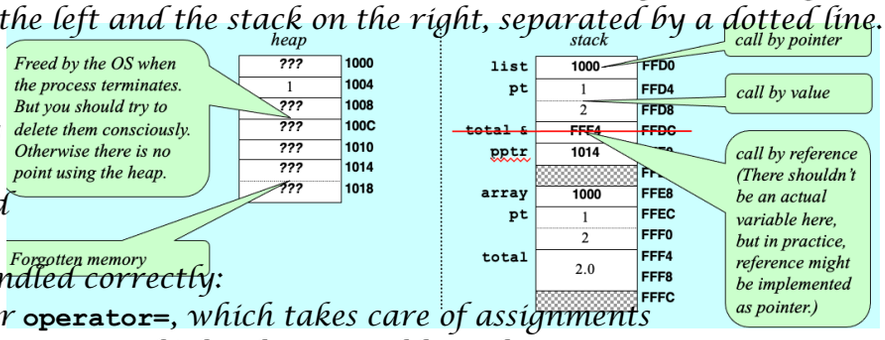
- Consuming additional resources;
- Performance impacts (inefficient);
- Possible stalls in program execution (programs need to do garbage collection);
- Incompatibility with manual resource management, etc.

In C++, class definitions often include a ***destructor***, which specifies how to free the storage used to represent an instance of that class. The prototype for a destructor has no return type and uses the name of the class preceded by a tilde (~). The destructor must not take any arguments.

```
CharStack::~CharStack() {
   delete[] array;
}
```

3) Heap-stack diagrams:

One of the most useful models to understand how C++ works is a ***heap-stack diagram***, which shows the heap on the left and the stack on the right, separated by a dotted line.

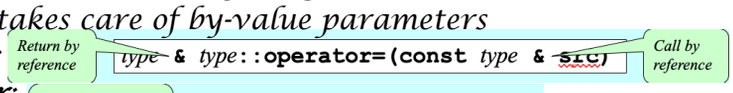


4) Copying objects:

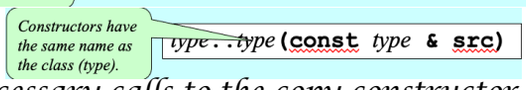
When you are defining a new abstract data type in C++, you typically need to define two methods to ensure that copies are handled correctly:

- The operator `operator=`, which takes care of assignments
- A copy constructor, which takes care of by-value parameters

The ***assignment operator*** has the form:



The prototype for the ***copy constructor***:



Call/return by value might cause unnecessary calls to the copy constructor. Constant call by reference protects the source.

An assignment operator can return anything it wants, but the standard C and C++ assignment operators return a reference to the left-hand operand (e.g., for primitive types). This allows you to chain assignments together.

Shallow Copying: the default behavior of C++ is to copy only the top-level fields in an object, which means that all dynamically allocated memory is shared between the original and the copy. The collection classes in C++ are defined so that copying one collection to another creates an entirely new copy of the collection.

Deep Copying: copies the data in the dynamically allocated memory as well.

❖ Implementing Deep Copy Semantics:

(i) Firstly, overload copy constructor to deep copy.

```
CharStack::CharStack(const CharStack & src) {
   deepCopy(src);
}
```

(ii) Overload the assignment.

```
CharStack & CharStack::operator=(const CharStack & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}
```

(iii) Write deep copy function.

```
void CharStack::deepCopy(const CharStack & src) {
   array = new char[src.count];capacity (a typo in the textbook, at least in my copy)
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
   capacity = src.capacity;
}
```

Chapter VII Additions for Data Structures

VI.1 Linear Structures:

1) **Overloading**: ***polymorphism*** is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

Function/operator ***overloading*** (a form of polymorphism) allows defining several functions with the same name as long as those functions can be distinguished by their signatures.

2) **Templates**: One of the most powerful features in C++ is the ***template*** facility, which makes it possible to define functions and classes that work for a variety of types.

The most common form of a template specification is:

`template <typename` *placeholder*`>` | `template <typename` *placeholder1*`, typename` *placeholder2*`>`

where ***placeholder*** is an identifier that is used to stand for a specific type when the definition following the **template** specification is compiled. The keyword `typename` can also be `class`

❖ Template in Functions:

The compiler will generate the code for many different versions of `max`, one for each type that the client uses.

```
template <typename ValueType>
ValueType max(ValueType v1, ValueType v2) {
   return (v1 > v2) ? v1 : v2;
}
```

❖ Templates in Class Definitions:

Templates are more commonly used to define generic classes. When they are used in this way, the **template** keyword must appear before the class definition and before each of the implementations of the member functions.

The most inconvenient aspect of using templates is that the `.h` files for template classes must contain both the prototypes and the corresponding code.

3) Redefining `operator[]`:

Returning a value by reference means putting the result into an ***lvalue***, which is an expression that can be used on the left side of an assignment.

```
template <typename ValueType>
ValueType & Vector<ValueType>::operator[](int index) {
   if (index < 0 || index >= count) {
      error("Vector selection index out of range");
   }
   return array[index];
}
```

The most important caution is that you must ***never*** use return by reference with a value that lives in the stack frame of the current method.

VI.2 Maps, Trees and Graphs:

1) Maps:

❖ Implementing maps using vectors: (`vector<structure>`).

```
struct KeyValuePair {
   std::string key;
   std::string value;
};

/* Instance variables */

Vector<KeyValuePair> bindings;
```

❖ complexity of operations:

Linear search --Keep track of all the key/value pairs in a vector. In this model, both the `get` and `put` operations run in O(N) time.

Binary search. -- `get` to…O(log N). But to keep the vector sorted, `put` runs in O(N).

Table lookup in a grid. (for some special cases).

2) The Idea of Hashing:

The `StringMap` class implemented using the hashing strategy is called a ***hash map*** (sometimes a ***hash table***). The implementation requires the existence of a free function (called a ***hash function***) that transforms a key into a non-negative integer (called a ***hash code***), preferably in constant time.

❖ About Hash Functions:

To achieve the high level of efficiency that hashing offers, a hash function must have the following two properties:

- The function must be **inexpensive to compute** (from computer's perspective of view).
- The function should distribute keys as uniformly as possible across the integer range, to minimize **collisions**.

Most implementations use techniques similar to those for generating pseudorandom numbers, to ensure that the results are **hard to predict**.

Implementations that use poorly designed hash functions run more slowly but nonetheless continue to give correct results.

❖ The Bucket Hashing Strategy:

The array of buckets is smaller than the number of hash codes, making it necessary to convert the hash code into a bucket index, typically by executing a statement like:

```
int index = hashCode(key) % nBuckets;
```

Rehashing: The ratio of the number of keys to the number of buckets is called the **load factor** of the map. The library implementation of `HashMap` increases the number of buckets when the map becomes too full. This process is called **rehashing**.

3) Trees:

❖ Implementing Maps Using BSTs:

The Stanford `Map` class (and `map` in Standard C++) uses a BST structure internally, in which `get` and `put` operate in O(logN) time while allowing iteration to proceed in order.

- **Priority Queue**: a queue in which the order in which elements are dequeued depends on a numeric priority. They are essential to both Dijkstra's and Kruskal's algorithms (examples of some very important **graph** algorithms).

  The standard algorithm for implementing priority queues uses a data structure called a **partially ordered tree**, which makes it possible to implement priority queue operations in O(logN) time.

  A partially ordered tree is a special class of **binary tree** (but not BST) with these additional properties:

  (i) The tree is complete, i.e., it is completely balanced, and each level of the tree is filled as far to the left as possible.

  (ii) The root node of the tree has higher priority than the root of either of its subtrees, which are also partially ordered trees.

  A **heap** is an array-based data structure that simulates the operation of a partially ordered tree.

`parentIndex(n)`	*is always given by*	`(n - 1) / 2`
`leftChildIndex(n)`	*is always given by*	`2 * n + 1`
`rightChildIndex(n)`	*is always given by*	`2 * n + 2`

  Heapsort: a comparison-based sorting algorithm utilizing the heap data structure.

  It can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted and an unsorted region, shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. It has **worst-case O(MogN)**, but it is **not a stable** sort.

4) Sets:

High-Level Operators in `set.h`:

Operators

s_1 + s_2	Returns the ***union*** of s_1 and s_2, which consists of the elements in either or both of the original sets.
s_1 * s_2	Returns the ***intersection*** of s_1 and s_2, which consists of the elements common to both of the original sets.
s_1 - s_2	Returns the ***set difference*** of s_1 and s_2, which consists of the all elements in s_1 that are not present in s_2.
s_1 += s_2 s_1 -= s_2 s_1 *= s_2	The +, -, and * operators can be combined with assignment just as they can with numeric values. For += and -=, the right hand value can be a set, a single value, or a list of values separated by commas.

❖ Implementing Sets:

Modern library systems adopt either of two strategies for implementing sets:

- **Hash tables**; (average O(1) performance for adding or testing membership; but do not support ordered iteration).
- **Balanced binary trees**; (O(logN) performance on the fundamental operations, but do make it possible to write an ordered iterator).

5) Character Sets and Bitwise Operations:

Representing the inclusion or exclusion of a character using a single bit makes efficient ***character sets*** (or, equivalently, ***sets of small integers***). If the bit is a 1, then that element is in the set; if it is a 0, it is not in the set. Create what is essentially an array of bits, with one bit for each of the ASCII codes. That array is called a ***characteristic vector/indicator vector***.

What makes this representation so efficient is that you can pack the bits for a characteristic vector into a small number of words inside the machine and then operate on the bits in large chunks

❖ Bitwise Operators:
(^--XOR: or but not both)

The expression `x << n` shifts the bits in the integer x leftward n positions. Spaces appearing on the right are all filled with 0s.

`x & y`	Bitwise AND. The result has a 1 bit wherever both x and y have 1s.
`x | y`	Bitwise OR. The result has a 1 bit wherever either x or y have 1s.
`x ^ y`	Exclusive OR. The result has a 1 bit wherever x and y differ.
`~x`	Bitwise NOT. The result has a 1 bit wherever x has a 0.
`x << n`	Left shift. Shift the bits in x left n positions, shifting in 0s.
`x >> n`	Right shift. Shift x right n bits (logical shift if x is unsigned).

The expression `x >> n` shifts the bits in the integer x rightward n positions. The question as to what bits are shifted in on the left depend on whether x is a signed or unsigned type:

- If x is an unsigned type, the `>>` operator performs a ***logical shift*** in which missing digits are always filled with 0s.
- If x is a signed type, the `>>` operator performs what computer scientists call an ***arithmetic shift*** in which the leading bit in the value of x never changes. Thus, if the first bit is a 1, the `>>` operator fills in 1s; if it is a 0, those spaces are filled with 0s. (Arithmetic shifts are efficient ways to perform multiplication or division of signed integers by powers of two).

Two's Complement: two's complement of an N-bit number is defined as its complement with respect to 2^N, i.e., the sum of a number and its two's complement is 2^N.

Conveniently, another way of finding the two's complement is inverting the digits and adding one.

5) Graphs:

❖ Three Strategies for Graph Abstraction:

- ***Use low-level structures***. This design uses the structure types `Node` and `Arc` to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.

```
struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

```
struct Node {
   std::string name;
   Set<Arc *> arcs;
};
```

```
struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
```

- ***Adopt a hybrid strategy***. This design defines a `Graph` class as a parameterized class using templates so that it can use any structures or objects as the node and arc types. This strategy retains the flexibility of the low-level model and avoids the complexity associated with the class-based approach.

```
template <typename NodeType,typename ArcType>
class Graph {
Set<NodeType *> nodes;
Set<ArcType *> arcs;
Map<std::string,NodeType *> nodeMap;
```

```
/* The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 */
```

- ***Define classes for each of the component types***. This design uses the `Node` and `Arc` classes to define the structure. In this model, clients define subclasses of the supplied types to particularize the graph data structure to their own application.

Chapter VIII Inheritance in C++

VIII.1 Class Hierarchies in C++

1) **Single & Multiple Inheritance**: although C++ supports ***multiple inheritance***, the vast majority of class hierarchies use ***single inheritance***. This convention means that class hierarchies tend to form ***trees*** rather than ***graphs***.

2) Representing Inheritance in C++:

```
class subclass : public superclass {
    body of class definition
};
```

can be **protected** *and* **private** *too.*

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Subclasses have access to the ***public*** and ***protected*** (but not private) members/methods in the superclass.

3) Overloading vs. Overriding: (polymorphism)

Overloading is about multiple functions/operators/methods of the same name but different signatures and possibly different implementations.

Overriding is about multiple methods of the same signature but different implementations defined in different classes connected through inheritance.

Method ***overloading*** is an example of ***static*** (or ***compile time***) ***polymorphism***, while method ***overriding*** is an example of ***dynamic*** (or ***run time***) ***polymorphism***.

4) Virtual Methods:

```
class Employee {
   virtual double getPay();= 0
};
```

In some hierarchy, a method which is ***overridden*** differently in each subclass and therefore is a ***virtual method***.

Once a method is declared as a virtual method, it becomes virtual in every subclass. In other words, it is not necessary to use the keyword virtual in the subclasses.

Any method that is always implemented by a concrete subclass is indicated by including `= 0` before the semicolon on the prototype line, to mark the definition of a ***pure virtual method***.

5) Abstract Classes:

Because of the ***pure virtual method*** **getPay**, the superclass **Employee** cannot be instantiated, and therefore is an ***abstract class*** that is never created on its own but instead serves as a common superclass for other ***concrete classes*** that correspond to actual objects.

Users cannot declare an object of an abstract class because some of the methods may not be properly implemented. But they can declare a pointer of an abstract class, which can be later used to point to an object of a concrete subclass inheriting from the abstract superclass.

6) The limitation of ***subclassing*** in C++

(i) Because a subclass requires at least as much space as its superclass, and typically requires more, when assigning an object of a subclass to a variable declared to be its superclass, C++ throws away any fields in the assigned object that don't fit into the superclass by default. This behavior is called ***slicing***. (More serious a problem than shallow copy!)

To avoid slicing, one approach is to define private versions of the copy constructor and assignment operator so that copying objects in that inheritance hierarchy is ***prohibited*** (just like the stream class hierarchy in C++).

(ii) Instances from different subclasses cannot be executed by a method (like **add** in a vector). One possible solution is to use ***pointer*** of the primitive superclass objects.

```
Vector<Employee *> payroll;
payroll.add(&bobCratchit);
for (Employee *ep : payroll) {
   cout << ep->getName() << ": " << ep->getPay() << endl;
}
```

(ii) Another question here is when some methods are overriding, like **`getPay`**, *what is the result of calling* **`ep->getPay`**. *In C++, a pointer can be declared as the superclass but pointed to the subclass, you have to explicitly allow for overriding by marking the method prototype with the keyword* **`virtual`**. *More precisely, if you leave out of the* **`virtual`** *keyword, the compiler determines which version of a method to call based on how the object is* ***declared*** *and not on how the object is* ***constructed***.

Unfortunately, using pointers complicates the process of memory management, which is already a difficult challenge in C++. Collections allow deep copying (by overloading assignment operator and copy constructor) while streams forbid copying.

7) Recall that ***a superclass constructor*** *is included in the initializer list.*

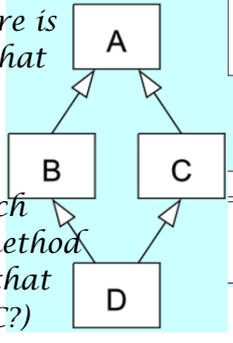
When calling the constructor for an object, the constructor ordinarily calls the ***default constructor*** *for the superclass, which is the one that takes no arguments.*

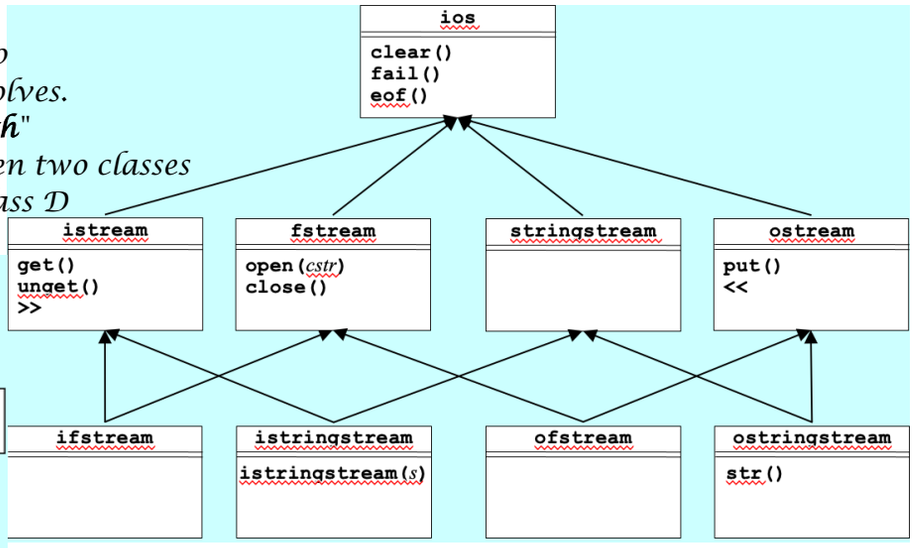
```
class GCircle : public GOval {
   GCircle(double x, double y, double r)
                     : GOval(x-r, y-r, 2*r, 2*r) {
      /* Empty */
   }
};
```

8) ***Multiple Inheritance****: C++ allows one class to inherit behavior from more than one superclass.*

Multiple inheritance tends to create more problems than it solves.

*The "****deadly diamond of death****" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. (Question: If there is a method in A that B and C have overridden, but D does not override it, which version of the method does D inherit: that of B, or that of C?)*





Chapter IX Strategies for Iteration

IX.1 Iterators in C++

1) Range-based `for` statement v.s. Index-based `for` statement

Range

```
for (string key : map) { . . . code to process that key . . . }
```

```
foreach (string key in map) { . . . code to process that key . . . }
```

Index

```
for (int i = 0; i < str.length(); i++) {
    . . . Body of loop that manipulates str[i] . . .
```

Iterator (before C++11)

```
for (Lexicon::iterator it = english.begin();
     it != english.end(); it++) {
```

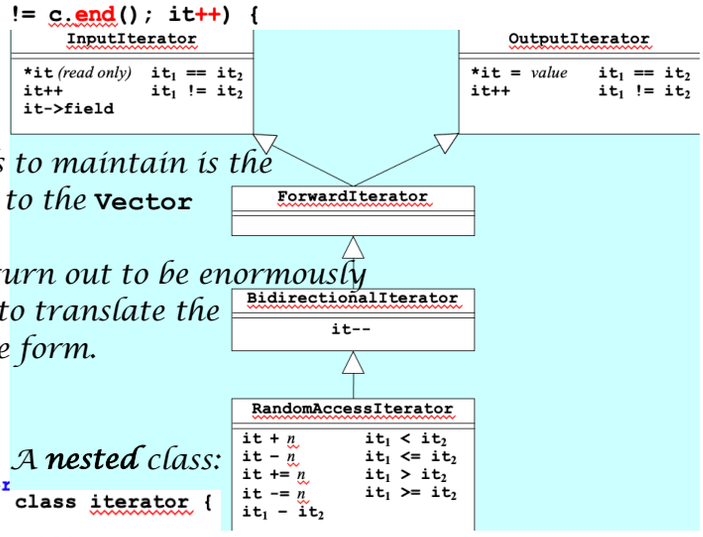
2) Iterators:

The C++ Standard Template Library makes extensive use of an abstract data type called an ***iterator***. Every collection class in the STL exports an `iterator` type (a ***nested type***) along with two standard methods that produce iterators:

- The `begin` method returns an iterator positioned at the beginning of the collection.
- The `end` method returns an iterator positioned just **past** the final element

```
for (ctype::iterator it = c.begin(); it != c.end(); it++) {
    . . . Body of loop involving *it . . .
}
```

Hierarchy: (RHS)



Implementation: (e.g., the `Vector` Iterator)

The only state information the iterator needs to maintain is the current index value, along with a pointer back to the `Vector` object itself.

Iterators for tree-structured classes like `Map` turn out to be enormously tricky, mostly because the implementation has to translate the recursive structure of the data into an iterative form.

```
iterator(const Vector *vp, int index) {
   this->vp = vp;
   this->index = index;
}
friend class Vector;
iterator begin() const { // public method of class Vector
   return iterator(this, 0);
}
iterator end() const {// public method of class Vector
   return iterator(this, count);
}
```

A ***nested*** class:

```
class iterator {

public:
```

… (need to overload some operators like `*`; `->`; `++`; `--`; etc.)

3) Another Collection Iterating Strategy:

Functions that allow you to call a function on every element in a collection are called ***mapping functions***.

Mapping functions are less convenient than iterators and are consequently used less often. They are, however, easier to implement.

IX.2 Function Pointers in C++

1) The ability to determine the address of a function makes it possible to pass functions as parameters to other functions. The function can call some functions supplied by the caller. Such functions are known as ***callback functions***.

2) Function Pointers in C++:

`double *g(double);`	Declares **g** as a function taking a **double** and returning a pointer to a **double**.
`double (*fn)(double);`	Declares **fn** as a pointer to a function taking and returning a **double**.

(e.g., one way to do selection sort `ascending` and `descending` are defined before using)

```
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int)) {
    ...
        if (comparisonFcn(array[currentIndex], array[bestIndex]))
            bestIndex = currentIndex;
```

```
selectionSort(array, 9, ascending);
// Sort the array in descending orde
selectionSort(array, 9, descending);
```

3) Mapping Functions:

The ability to work with pointers to functions offers a solution to the problem of iterating through the elements of a collection. To use this approach, the collection must export a ***mapping function*** *that applies a client-specified function to every element of the collection.*

```
template <typename ValueType>
void Vector<ValueType>::mapAll(void (*fn)(ValueType)) const {
   for (int i = 0; i < count; i++) {
      fn(array[i]);
   }
}
```

The `mapAll` *function is at the same level of the* `iterator` *class, therefore can get access to the low level data structures directly, without relying on the definition of* `iterator`*.*

Implementations: (for multi-variable functions)

```
template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::mapAll(BSTNode *t,
                                    void (*fn)(KeyType, ValueType)) const {
   if (t != NULL) {
      mapAll(t->left, fn);
      fn(t->key, t->value);
      mapAll(t->right, fn);
   }
}
```

What is the traversal order here, why?

Passing Data to Mapping Functions: 2 strategies

- *Passing an additional argument to the mapping function, which is then included in the set of arguments to the callback function (overloading* `mapAll` *in many forms).*
- *Passing a* ***function object*** *or* ***functors*** *to the mapping function. A function object is simply any object that overloads the function-call operator, which is designated in C++ as* `operator()`*.*

(e.g., in the RHS)

```
template <typename FunctorType>
void Lexicon::mapAll(FunctorType fn) const {
    for (std::string word : *this) {
        fn(word);
    }
}
```

```
class ListKLetterWords {
public:
   ListKLetterWords(int k) {
      this->k = k;
   }
   int operator()(string word) {
      if (word.length() == k) {
         cout << word << endl;
      }
   }
private:
   int k;     /* Length of desired words */
};
void listWordsOfLengthK(const Lexicon & lex, int k)
   lex.mapAll(ListKLetterWords(k));

}
```

Another Use of Iterators in C++:

Iterators have even greater importance in C++ because so many of the functions in the STL take iterators as parameters.

```
sort(v.begin(), v.end());
```

IX.3 Functional Programming in C++

Even though ***functional programming*** *was not a goal of the language design, the fact that C++ includes both templates and function objects make it possible to adopt a programming style that is remarkably close to the functional programming model, which is characterized by the following properties:*

- *Programs are expressed in the form of* ***nested function calls*** *that perform the necessary computation without performing any operations (such as assignment) that change the program state.*
- *Functions are data values and can be manipulated by the programmer just like other data values.*

STL offers rudimentary support for the functional programming paradigm through the `<functional>` *interface, which exports a variety of classes and methods, which generally fall into two categories represented by the template class* `binary_function` *that takes two arguments, and* `unary_function` *that takes a single argument.*

Student Number _____

Student Name _____

The Chinese University of Hong Kong, Shenzhen

FINAL EXAMINATION

Term 2, Spring, 2021-2022

CSC3002 Introduction to Computer Science:

Programming Paradigms

Examination Duration: 120 minutes.

This examination has 32 questions divided into 4 sections.

Exam Conditions:

This is a FORMAL Examination.

This is a RESTRICTED OPEN BOOK Examination.

Materials Permitted In The Exam Venue:

Maximum of one (1) sheet of double-sided A4 paper notes are permitted. NO OTHER MATERIALS ARE PERMITTED

Calculators are NOT ALLOWED in this examination.

2B pencils and erasers should be brought by the students and used to mark the answer sheet.

Materials To Be Supplied To Students:

This question and answer book, the answer sheet, and scratch paper.

Materials To Be Collected From Students:

EVERYTHING MUST BE RETURNED, including this question and answer book, the answer sheet, and scratch paper. Please answer the questions in the first 3 sections by marking the answer sheet, and those in the last section by writing down your answers in the specified blanks in this question and answer book.

Section 1. True or false questions. (10 × 1% = 10%)
Pick "a" for "True" or "b" for "False" for each of the following statements. Please mark your answers (only "a" or "b") on the answer sheet.

1. In C++, a **variable** is a named address for storing a type of value, and a **pointer** is a special type of variable that holds the address of another variable.

   a. True b. False

2. In C++, a **function** is a named section of code that performs a specific operation, and a function must have a parameter list as the input and return a value as the output.

   a. True b. False

3. The following two statements will generate two strings with the same content:

```
std::string str1 = "hello,world";
std::string str2 = "hello" + "," + std::string("world");
```

   a. True b. False

4. In C++, streams are complicated objects that cannot be copied. If you want to pass a stream object into a function you must use **call by reference**, and if you want to return a stream object from a function, you must use **return by reference**.

   a. True b. False

5. In an array, the size of the array specified in the declaration is called the **allocated size**. The number of elements actively in use is called the **effective size**. The C++ compiler makes sure that the effective size can never exceed the allocated size.

   a. True b. False

6. The following statements declare two pointers **`p1`** and **`p3`**, and two integers **`p2`** and **`p4`**:

```
int*  p1, p2;
int  *p3, p4;
```

   a. True b. False

7. For a variable **`x`**, the expression **`*&x`** is essentially equivalent to **`x`**; and for a pointer **`p`**, the expression **`&*p`** is equivalent to **`p`**.

   a. True b. False

8. A **hash function** should distribute keys as uniformly as possible across the integer range to avoid **collisions**, and the **`put`** and **`get`** operations can only achieve $O(1)$ performance without any collision.

   a. True b. False

9. If **`c`** is a nonempty collection in C++ STL, calling **`c.begin()`** and **`c.end()`** will return **iterators** pointing at the first and last element of that collection, respectively.

   a. True b. False

10. C++ is a **multi-paradigm** programming language. Besides the **procedural** programming paradigm originally supported by C, C++ also supports the **object-oriented** paradigm. However, C++ only supports other paradigms such as the **functional** paradigm through external libraries.

    a. True b. False

Section 2. Single choice questions. (10 × 2% = 20%)
Pick the correct option in each of the following questions. Note that only ONE option is correct. Please mark your answers on the answer sheet.

11. Calculate the result of the following expressions based on the C++ rules:

```
9.0*(8-7) / 6 + 5*4 % 3*(2+1)
```

    a. `3`
    b. `3.5`
    c. `7`
    d. `7.5`

12. A local variable in a function is usually destroyed when the function returns. If you want to retain its value until the next time the function is called, which keyword should be used before the declaration of the variable:

    a. const
    b. extern
    c. static
    d. protected

13. Suppose that you are using the **Merge sort** algorithm to sort a vector of 1,000 integers and find that it takes T milliseconds to complete the operation. How many milliseconds would you expect the running time to be if you use the same algorithm to sort a vector of 1,000,000 integers on the same machine? (Assuming there is enough memory and $\log_2 1000 \approx 10$.)

    a. 2,000 T
    b. 10,000 T
    c. 20,000 T
    d. 1,000,000 T

14. If we have built a 40-bit computer on which the memory address will be represented in one 40-bit long **machine word**, and the **smallest addressable unit** is a sequence of 8 consecutive bits called a **byte**, how much memory can this computer manage, theoretically? (Assume 1KB = 2^{10} bytes, 1MB = 2^{10} KB, 1GB = 2^{10} MB.)

    a. 40 GB
    b. 256 GB
    c. 320 GB
    d. 1024 GB

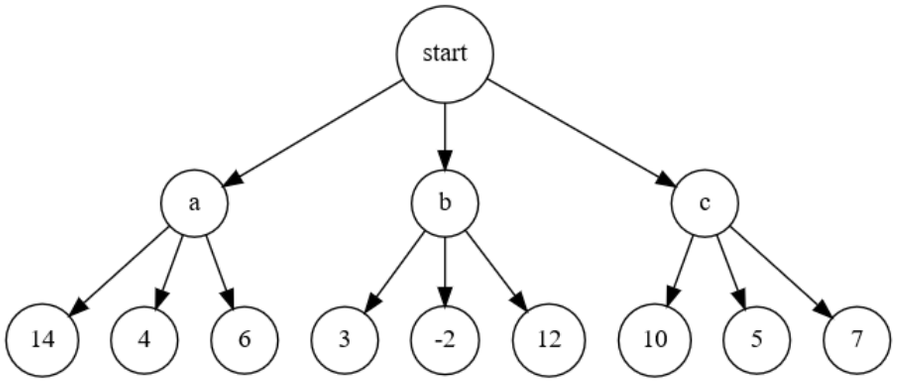
15. Assuming the following C++ variable declaration statement

```
int i = 2022, *pi = &i, **ppi = &pi;
```

leads to the addresses of **`i`**, **`pi`**, and **`ppi`** being **`0x5190`**, **`0x5198`**, and **`0x51A0`**, respectively, what are the values of **`**ppi`** and **`*ppi`**?

a. **`0x51A0`** and **`0x5198`**
b. **`0x5198`** and **`0x5190`**
c. **`0x5190`** and **`0x5198`**
d. **`2022`** and **`0x5190`**

16. In a two-player game, suppose you are in a position in which the analysis for the next two moves shows the following rated outcomes (larger values represent better outcomes for you) from your original player's point-of-view:



If you adopt the **minimax** strategy in your program, what is the best move and the rating of that move from your perspective?

a. **`a, 14`**
b. **`b, 12`**
c. **`c, 5`**
d. **`b, -2`**

17. Computers typically represent each pixel in an image as a 32-bit integer or 8 hexadecimal digits (starting with **`0x`**), in which the 4 bytes are interpreted as the transparency (α) and color (Red, Green, Blue) components. The following bitwise operations extract one of the components from the hexadecimal representation of the αRGB color stored in an integer variable **`pixel`**. Which component is extracted and what is the value of that component in **decimal** format?

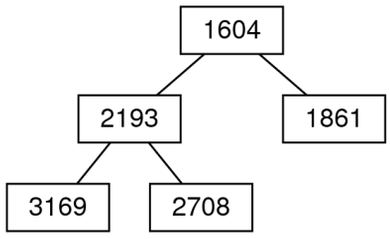
```
int pixel = 0x05192022;
int color = (pixel >> 8) & 0xFF;
```

a. α, 0
b. α, 256
c. Green, 20
d. Green, 32

18. If we build a binary search tree by inserting the nodes in increasing order (i.e., the smaller the key is, the earlier the node is inserted), and do not balance it, the result will be an extremely unbalanced tree growing linearly. If we then traverse this tree and **`cout`** the key, which one of the following traversal approaches will have different output from the others?

a. Preorder
b. Inorder
c. Postorder
d. Level-order

19. Suppose that you are modelling a **priority queue** with a **partially ordered tree** that contains the following data (assuming smaller values have higher priorities):



What is the proper content in the corresponding **heap** after deleting a node and then inserting "2022"?

a. 1604, 2022, 1861, 3169, 2193
b. 1861, 2022, 2193, 2708, 3169
c. 1861, 2022, 2708, 3169, 2193
d. 1861, 2193, 2022, 3169, 2708

20. What does the following declaration do?

```
double (*f)(double)
```

a. Declares **`f`** as a function taking a double and returning a pointer to a double.
b. Declares **`f`** as a function object or functor taking a double and returning a double.
c. Declares **`f`** as a pointer to a function taking a double and returning a double.
d. Declares **`*f`** as a function taking a double and returning a double.

Section 3. Multiple choice questions. (10 × 4% = 40%)
Pick the correct option(s) in each of the following questions. Note that there may be ONE to FOUR correct options for each question. Incomplete answers will get partial scores (2%). Please mark your answers on the answer sheet.

21. What are the **distinguishing** features of **object-oriented** programming languages like C++ (i.e., what makes object-oriented programming different from other programming paradigms)?

a. Abstraction
b. Encapsulation
c. Inheritance
d. Polymorphism

22. In C++, functions can be **overloaded**, which means that you can define different functions with the same name as long as they have different **signatures**. The signature of a function includes the following:

a. The number of the parameters
b. The names of the parameters
c. The types of the parameters
d. The type of the return value

23. Which of the following collection classes do not support the use of the range-based **`for`** loop?

a. **`vector`**
b. **`stack`**

c. **`queue`**
d. **`map`**

24. In the definition of a class, the data members can be declared as:

    a. public
    b. protected
    c. friend
    d. private

25. Normally in C++ you should not use pointers as much as in the C language, in which pointers are important for the following reasons:

    a. Pointers allow you to specify the memory addresses for dynamically allocated variables and objects in the heap.
    b. Pointers make it possible to reserve new memory during program execution.
    c. Pointers allow you to refer to a large data structure in a compact way.
    d. Pointers can be used to record relationships among data items, e.g., linked list.

26. In C++, any expression that refers to an internal memory location capable of storing data is called an **lvalue**, which can appear on the left side of an assignment statement. Which of the following terms in a C++ program **cannot** be lvalues?

    a. **`p`**
    b. **`(p+1)`**
    c. **`&p`**
    d. **`*p`**

27. What **properties** must a problem have for **recursion** to make sense as a solution strategy?

    a. There must be a function that calls itself.
    b. There must be an inclusion-exclusion pattern in the problem.
    c. There must be simple cases or exit points for which the answer is easily determined, without using recursion.
    d. There must be a recursive decomposition that allows you to break any complex instance of the problem into simpler problems of the same form.

28. A **ring buffer** is an array whose end is linked back to the beginning. In a typical ring buffer based queue implementation as follows,

```
ValueType *array;
int capacity;
int head;
int tail;
```

    the **`head`** field holds the index of the next element to come out of the queue, and the **`tail`** field holds the index of the next free slot. If we use the equality of the **`head`** and **`tail`** fields to indicate an empty queue, which of the following cases may indicate that the queue is full?

    a. **`head == tail`**
    b. **`head - tail == 1`**
    c. **`tail - head == capacity`**
    d. **`tail - head == capacity - 1`**

29. To avoid the default shallow copy or slicing in C++, which of the following functions should be overloaded when designing new classes:

    a. The copy constructor
    b. The assignment operator (**=**)
    c. The insertion operator (**<<**)
    d. The selection operator (**[]**)

30. After running the following program, which of the numbers will be seen in the output?

```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    A() { a = 1; }
    virtual void display() { cout << a << endl; }
};
class B: public A {
public:
    int b;
    B() { b = 2; }
    void display() { cout << a << b << endl; }
};
class C: public B {
public:
    int c;
    C() { a = 2; c = 3; }
    void display() { cout << a << b << c << endl; }
};
int main() {
    C oC;
    B* pb = &oC;
    pb->display();
    return 0;
}
```

    a. **1**
    b. **2**
    c. **3**
    d. None, compile error.

Section 4. Fill-in-the-Blank Questions. (12%+18% = 30%)
Answer the following questions. <u>Please write your answers on THIS EXAM PAPER in the blank space after the questions.</u>

31. What is the formatted output of the following program? Write your answer in the box, and make sure the format is correct. (12%)

```
#include <iostream>
using namespace std;
int main(void)
```