Intro

C++: Compiled Python: Interpreted source ⇒ tokens

Compiler: Takes entire program and converts it into object code and executed after linking. E.g. C、C++

Interpreter: Directly executes instructions E.g. Python、R

Java: source ⇒ intermediate binary form called bytecode, executed by JVM. Some JVM compile, some interpret.

Source / Header $\xrightarrow{编译}$ obj file (source) / obj file (header) > linker ⇒ Executable

IDE: ① Source code editor: a class browser, GUI, version control system ② Compile, interpret or both ③ A debugger

Library

Namespace: Using x::name1, using x::name2 An e.g. for x: Std

Structed: Programs have clean, goto-free, nested control structures (while, for, etc)

Procedural: Imperative programming with functions operation.

Variable name: ① Must start with a letter or '_' ② No space or special char.

Constant declaration: const double PI = 3.1415926;

Unary operators, ?:, =, op= are read from the RHS

In OOP: operator++( ) ⇒ ++x; operator++(int e.g) ⇒ x++

6 relational operators: ==, !=, <, >, <=, >= 3 logical: &&、||、!

E.g. If n=0, then n!=0 && x%n==0 only execute the left part.

When using switch: If break is not included in one case, all statements following the case are also executed, until a break is reached or to the end.

- A parameter is a variable in a function declaration and definition, and a placeholder for the argument.
- An argument is an expression used when calling the function, and an actualization of the parameter.
- When a function is called, the arguments are the data (actual values) you pass into the function's parameters (local variables).

```
int gcd(int x, int y) {
   int guess = (x < y) ? x : y;
   while (x % guess != 0 || y % guess != 0) {
      guess--;
   }
   return guess;
}
```

Brute-Force Approach

Check the number one by one to get GCD.

```
int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

Euclid's Algorithm to get GCD

Overload: 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.

The specification of the default values (***default arguments***) appears only in the function prototype and not in the function definition. Optional Parameters

Any optional parameters must appear at the end of the parameter list. No required parameters may appear after an optional one.

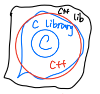
```
void setInitialLocation(double x, double y);

void setInitialLocation() {
   setInitialLocation(0, 0);
}
```

Keep the user from calling the function with only one argument.

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

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***.



Relationship between C & C++

Interface is only read once:

```
# ifndef _error_h
# define _error_h
   ⋮
# endif
```

Extern (Reserved word): e.g. To export the constant PI, you need to add extern in both .h & .cpp file. extern const double PI;

Principles of Interface Design: Unified, Simple, Sufficient, General, Stable

In-Function static: Only initialize once to record behaviors

String

C string literal: char cstr[]="hello, world", terminated by a null byte whose ASCII is 0.

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

Question: is `"a"` the same as `'a'`?

`"a"` is a string literal containing an `'a'` and a null terminator `'\0'`, and therefore is a 2-char array.

```
cstr[] = "hello";
cstr = "world";
cstr[] = "world";
```

← prohibited (初始化时已确定)

Sizeof x returns the actual memory size of x, can't be modified by strcpy

<cstring>、<string.h>、"string.h" → C string lib <string> → C++ string lib

- **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.

```
string toUpperCase(string str) {
   for (int i = 0; i < str.length(); i++) {
      str[i] = toupper(str[i]);
   }
   return str;
}
```

若(string & str), 则效率更高, 但不安全. (void, no return)

Stream

insertion operator: <<; extraction operator: >>;

A ***stream*** is an abstraction (object) that represents an input source or output destination of characters of indefinite length, on which input and ouput operations can be performed.

getline: 读整行, 不忽略空格

cin: treat space as stop sign

```
int ch;
while ((ch = infile.get()) != EOF) {
   cout.put(ch);
}
```

```
char ch;
while (infile.get(ch)) {
   cout.put(ch);
}
```

Use get to read the whole file

Cannot copy a stream, pass/return by reference, 作为参数加 "&"

e.g. stringstream ss << "abc"; cout << ss.str(); 可一次性输出string的集合

Collections

A type defined in terms of its behavior rather than its representation is called an ***Abstract Data Type (ADT)***.

ADT e.g: Vector, stack, string, Queue, etc.

Simplicity: fewer details for the client to understand.

Flexibility: free to change underlying implementation as long as the interface remains the same.

Security: prevents the client from changing the values in the underlying data structure in unexpected ways.

It is possible to create ***nested*** vectors, so that, for example, `Vector<Vector<char> >` represents a two-dimensional vector of characters. 需要留空

Also regarded as a "Grid"

```
Vector<char> vec(3);
Vector< Vector<char> > board(3, vec);
// or simply
Vector< Vector<char> > board(3, Vector<char>(3));
```

Use a Vector to Replace Grid

Range-Based for: for(string key : map)…

Lexicon: 字母顺序 (No need to specify element type)

Iterator: Grid: 逐行, Map/Set: 比较函数, Hashmap: Seemingly Random, Hashset: unordered

Lexicon: Extremely space-efficient, easy for iterators to process words in alpha order

Class

enum: Unless specified, it assigns value started at 0.

Struct: struct typename {…}. This definition creates a type, not a var. If you want to declare a var, use typename var_name;

Definition of a class: class typename { public: … ; private: … };

Initializer list: `Point(int xc = 0, int yc = 0) : x(xc), y(yc) {}` No return

⇕

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

☆ Constant members can only be initialized by Initializer list.

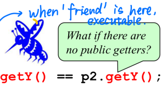
```
ostream & operator<<(ostream & os, Point pt) {
   return os << pt.toString();
}
```

↳ cout << pt; An overloading example

Defining an operator as a free function often produces code that is easier to read (the operands for a binary operator are both passed as parameters as usual), but also means that the operator function must be designated as a `friend` to refer to the private data that do not have public getters.

```
// 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) {
X return p1.x == p2.x && p1.y == p2.y;
✓return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}
```

when 'friend' is here, executable.



封装 ***Encapsulation***, 多态 ***Polymorphism***, and 继承 ***Inheritance***.

Object-Oriented Programming

继承: multiple inheritance. 比 global var safer.

多态: overloading

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

Inefficient becuz repeatly calling same functions ↑

```
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, t1, t0 + t1);
}
```

✓

Recursion

```
if (test for a simple case) {
    Compute and return the simple solution without using recursion.
} else {
    Divide the problem into one or more sub-problems that have the same form.
    Solve each of the sub-problems by calling this method recursively.
    Return the solution by reassembling the results of the various sub-problems.
}
```

Recursive strategy

```
int gcd(int x, int y) {
   if (x % y == 0) {
      return y;
   } else {
      return gcd(y, x % y);
   }
}
```

GCD Algorithm by Recursion

```
bool isSubstringPalindrome(string & str, int p1, int p2) {
   if (p1 >= p2) {
      return true;
   } else {
      return str[p1] == str[p2] &&
             isSubstringPalindrome(str, p1 + 1, p2 - 1);
   }
}
bool isPalindrome(string str) {
   return isSubstringPalindrome(str, 0, str.length() - 1);
}
```

若check 后 P1+1 > P2-1 也需返回true head tail

长度为偶数, 则在">"跳出, 长度为奇数, 则在"="跳出

Check Palindrome by Recursion

```
void moveTower(int n, char start, char finish, char temp) {
   if (n == 1) {
      moveSingleDisk(start, finish);
   } else {
      moveTower(n - 1, start, temp, finish);
      moveSingleDisk(start, finish);
      moveTower(n - 1, temp, finish, start);
   }
}
```

Recursion for Hanoi

```
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);
   }
}
```

Inclusion、Exclusion

不含目标元素 / 含目标元素

We can use multiprocessor to solve recursion problem in parallel.

```
bool solveQueens(Grid<char> & board, int nPlaced) {
   int n = board.numRows();
   if (nPlaced == n) return true;
   for (int row = 0; row < n; row++) {
      if (queenIsLegal(board, row, nPlaced)) {
         board[row][nPlaced] = 'Q';
         if (solveQueens(board, nPlaced + 1)) return true;
         board[row][nPlaced] = ' ';
      }
   }
   return false;
}
```

8皇后

Binary Search

```
int binarySearch(string key, Vector<string> & vec, int p1, int p2) {
   if (p1 > p2) return -1;
   int mid = (p1 + p2) / 2;
   if (key == vec[mid]) return mid;
   if (key < vec[mid]) {
      return binarySearch(key, vec, p1, mid - 1);
   } else {
      return binarySearch(key, vec, mid + 1, p2);
   }
}
```

floor version

ASCII 码比较大小

Recursion: ① Check the var at steps changes or not ② Check if the program can terminate.

```
Vector<string> generateSubsets(string set) {
   Vector<string> result;
   if (set == "") {
      result.add("");
   } else {
      result = generateSubsets(set.substr(1));
      int n = result.size();
      for (int i = 0; i < n; i++) {
         result.add(set[0] + result[i]);
      }
   }
   return result;
}
```

```
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;
}
```

→ 无路可走

没走通 unmarked

```
int findGoodMove(int nCoins) {
   int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
   for (int nTaken = 1; nTaken <= limit; nTaken++) {
      if (isBadPosition(nCoins - nTaken)) return nTaken;
   }
   return NO_GOOD_MOVE;
}
bool isBadPosition(int nCoins) {
   if (nCoins == 1) return true;
   return findGoodMove(nCoins) == NO_GOOD_MOVE;
```

mutual recursion

What you want instead is to choose the move that minimizes the maximum rating available to your opponent. This strategy is called the ***minimax*** algorithm.

最小化对手能得到的最高分数.

Algorithm

```
void sort(Vector<int> & vec) {
   int n = vec.size();
   for (int lh = 0; lh < n; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < n; i++) {
         if (vec[i] < vec[rh]) rh = i;
      }
      int temp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = temp;
```

Selection Sort

Disadvantages of using actual time to estimate efficiency:

① Several runs

② Several independent trials

③ Discard some measure (电脑同时运行其他后台)

Worst-case complexity: The upper bound on the computational complexity. You can ***guarantee*** that the performance of the algorithm will be at least as good as your analysis indicates.

Average-case complexity: The best statistical estimate of actual performance. Usually much more difficult to carry out and typically requires considerable mathematical sophistication.

Constant	$O(1)$	$O(Yeah)$ Finding elements in a vector using indices
Logarithmic	$O(\log N)$	$O(Nice)$ Binary search in a sorted vector
Linear	$O(N)$	$O(Fine)$ Summing a vector; linear search
$N\log N$	$O(N\log N)$	$O(K)$ Merge sort
Quadratic	$O(N^2)$	$O(Well)$ Selection sort
Cubic	$O(N^3)$	$O(My)$ Obvious algorithms for matrix multiplication
Exponential	$O(2^N)$	$O(No)$ Tower of Hanoi solution
Factorial	$O(N!)$	$O(MG)$ Brute-force search through all permutations

```
void sort(Vector<int> & vec) {
   int n = vec.size();
   if (n <= 1) return;
   Vector<int> v1;
   Vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   sort(v1);
   sort(v2);
   vec.clear();
   merge(vec, v1, v2);
```

```
void merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2)
   int n1 = v1.size();
   int n2 = v2.size();
   int p1 = 0;
   int p2 = 0;
   while (p1 < n1 && p2 < n2) {
      if (v1[p1] < v2[p2]) {
         vec.add(v1[p1++]);
      } else {
         vec.add(v2[p2++]);
      }
   }
   while (p1 < n1) vec.add(v1[p1++]);
   while (p2 < n2) vec.add(v2[p2++]);
```

merge sort algorithm

```
void quicksort(Vector<int> & vec, int start, int finish)
   if (start >= finish) return;
   int boundary = partition(vec, start, finish);
   quicksort(vec, start, boundary - 1);
   quicksort(vec, boundary + 1, finish);
 int partition(Vector<int> & vec, int start, int finish)
    int pivot = vec[start];
    int lh = start + 1;
    int rh = finish;
    while (true) {
       while (lh < rh && vec[rh] >= pivot) rh--;
       while (lh < rh && vec[lh] < pivot) lh++;
       if (lh == rh) break;
       int tmp = vec[lh];
       vec[lh] = vec[rh];
       vec[rh] = tmp;
    }
    if (vec[lh] >= pivot) return start;
    vec[start] = vec[lh];
    vec[lh] = pivot;
    return lh;
```

Quick Sort

1. Select the first element as the ***pivot*** and set it aside.
2. Keep two indices into the remaining elements, starting at each end.
3. Advance the indices until the left index is larger than the pivot and the right index is smaller.
4. Exchange the elements at the two indices.
5. Repeat the process until the indices coincide.
6. Swap the pivot and the index.

Sorting complexity	Best	Average	Worst
Selection sort	N^2	N^2	N^2
Insertion sort	N	N^2	N^2
Bubble sort	N	N^2	N^2
Merge sort	$N \log N$	$N \log N$	$N \log N$
Quicksort	$N \log N$	$N \log N$	N^2
Heapsort	NlogN	NlogN	NlogN

Pointer & Array

The smallest addressable unit on a computer is byte= 8 bits. The unit that represents the most common integer size on a particular hardware is called 'word' A computer said to be 32-bit has a hardware word size of 32-bits (4 bytes) The largest possible address length, used to designate a location in memory, is typically a word.

3 types of spaces: Static area. heap →←, stack. In classical arch, the stack & heap grow toward each other to max the available space.

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`

Enum: The same space as int.

Struct: The sum of their fields.

Array: element size x # elements Pointers: 4 bytes on 32-bit, 8 bytes on 64-bit machine

"Sizeof x" returns the actual memory size of the variable x.

Usage of pointers: ① Pointers allow you to refer to a large data structure in a compact way. E.g: call by pointers. ② Pointers make it possible to reserve new memory during execution. E.g. Dynamic Allocation. ③ It can be used to record relationships among data items: linked structures.

double x=2.5; double *px=&x; px is a pointer whose value is the address of x, *px represents x itself.

```
void swap(int * px, int * py)
   int tmp = *px;
   *px = *py;
   *py = tmp;
```

Usage: swap(&n1, &n2)

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

swap(n1, n2)

	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 (the same as `&i`)
Pointing/referring to nothing	Yes (`NULL/` `nullptr` since C++11)	No
Reassignments to new objects	Yes	No
Supported by	C and C++	C++

Point type pointer. Given pp as a

pp.getX() ✗

*pp.getX() ✗

⇕

*(pp.getX())

Right One:

(*pp).getX()

or

pp→getX()

- Parameters or local variables declared in the current method
- Instance variables of the current object
- ~~Global variables defined in this scope~~

Use 'this' to represent instance variables.

Point :: Point(int x, int y){

this→x = x; this→y = y;}

Array: ① 只用[ ]选择 ② 不检查 index 越界 ③ Declared length 不变 ④ Array 不存储 actual length，需额外传入实际使用的元素个数的参数.

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***.

```
sizeof MY_ARRAY / sizeof MY_ARRAY[0]
```

确认未知Array内的元素数量.

E.g. int list[100]; C++ treats list as a pointer to address & list[0] whenever necessary, and list[i] is just *(list+i) by pointer arithmetic. address is passed.

Array is a non-modifiable lvalue, when you pass an array to a func, only ↑

```
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 By default: Treat a as a pointer	ADDRESS1	ADDRESS1	ADDRESS1
*	0	ADDRESS1	0	0
&	ADDRESS1	N/A (越过该array)	N/A	ADDRESS2
+1	ADDRESS1 +1 int	ADDRESS1 +4 int	ADDRESS1 +1 int	ADDRESS1 +1 int

E.g. char *msg = "hello, world"

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

```
int strlen(char *str) {
   char *cp;
   for (cp = str; *cp != '\0'; cp++);
   return cp - str;
}
void strcpy(char* dst, char* src)
   while (*dst++ = *src++);
```

errors.

*p++ ⇔ *(p++) 少用, avoid buffer overflow

Dynamic Memory

int *p1 = new int; int * arr = new int [10000]; Type *pp = new Type obj;

delete p1;   delete [ ] arr;   delete pp;

Delete a ptr only frees the memory space pointed by ptr, but not the memory space occupied by ptr itself. To avoid it, we should ptr = NULL; free bugs

Garbage Collection: Reduce ① Memory leaks ② Dangling pointer bugs ③ Double

If new is used in constructors, delete should be used in destructors.

(Assignment operator)

Return by reference

e.g. a=b=c   保证在函数中src的元素不被改变

```
type & type::operator=(const type & src)
```

Make the return lvalue available

Call by reference

```
type::type(const type & src)
```

(Copy constructor)

```
CharStack::CharStack(const CharStack & src) {
   deepCopy(src);
}

CharStack & CharStack::operator=(const CharStack & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

void CharStack::deepCopy(const CharStack & src) {
   array = new char[src.~~count~~]; capacity
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
   capacity = src.capacity;
```

S1 = S2   参数是一个类

if (this != &src) → 防止 S1 = S1 时依然对S1有变化操作

delete[] array; → 防止 S1, S2 size 不匹配，故在deepcopy中重新生成

return *this; → pointer that pointing at yourself

capacity (a typo in the textbook, at least

```
#include <iostream>
using namespace std;

class A {
public:
   static const int c = 1;
   static int i;
};

int A::i = 2;
```

Initializer to a variable, not a constructor

```
int main()
{
   A a, b;
   cout << a.c << a.i << endl;
   cout << b.c << b.i << endl;
   cout << (&a.c == &b.c) << endl;
   cout << (&a.i == &b.i) << endl;
   a.i = 3;
   cout << b.i << endl;
```

1 2 / 1 2 / True (1) / True / 3 — They use the same address.

Buffer

Editor buffer: ① Array: 超capacity 后 expand ② Linked list: start后加 dummy cell ③ 2-stack based: stack中间的分界代表 cursor.

↳ e.g. a b c | d e ⇒ c b a before, d e after

	Average:	array	list	stack
F	`moveCursorForward()`	$O(1)$	$O(1)$	$O(1)$
B	`moveCursorBackward()`	$O(1)$	From start point begin $O(N)$	$O(1)$ Need to move everything
J	`moveCursorToStart()`	$O(1)$	$O(1)$	$O(N)$ from a stack to
E	`moveCursorToEnd()`	$O(1)$	From current pos $O(N)$ begin	$O(N)$ another
I	`insertCharacter(ch)` best O(1) Worst O(N)	$O(N)$	$O(1)$	$O(1)$
D	`deleteCharacter()` 挪动每个元素	$O(N)$	$O(1)$	$O(1)$

→ Depends on complexity of charstack

Linear Struct

template <typename placeholder>

Typename can be class, placeholder: specific type

```
using namespace std;
max(string("cat"), string("dog"));
```

方法已存在于 std, ambiguous in std & 自己版本.

Correct: ::max(std::string("cat"), std::string("dog")); std::max(…) 用STL Version

做 template 时需将 .h、.cpp合并进 .h 中，不然无法编译. Ring-Buffer: use % to complete size: (tail + capacity - head) % capacity. Enqueue: tail=(tail+1)% capacity. Dequeue: head=(head+1)% capacity, Expand: array[i] = old[(head+i)% oldcapa] avoid collisions.

Map

Hash functions: ① Inexpensive to compute ② Distribute keys as uniformly as possible

Use the hash code for each key to select an index into an array that will contain all the keys with that hashcode. int index = hashcode(key) % nBuckets.

Tips: Cell *① *② nBuckets; ①: 指向linked-list ② 指向bucket array, 生成指向Cells的pointer

Tree

```
struct TreeNode {
   string key;
   Vector<TreeNode *> children;
```

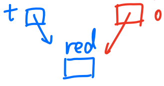
A pointer to represent tree

```
struct BSTNode {
   string key;
   BSTNode *left, *right;
```

```
Node* findNode(Node* t, string key) {
   if (t == NULL) return NULL;
   if (key == t->key) return t;
   if (key < t->key) {
      return findNode(t->left, key);
   } else {
      return findNode(t->right, key);
```

```
void insertNode(Node* & t, string key)
   if (t == NULL) {
      t = new Node;
      t->key = key;
      t->left = t->right = NULL;
      return;
   }
   if (key == t->key) return;
   if (key < t->key) {
      insertNode(t->left, key);
   } else {
      insertNode(t->right, key);
```

If no, we actually define a new tree locally



Linear search: O(N)

Binary Search: get O(logN), put O(N)

grid: O(1)

Hashing O(1).

For a leaf node (with no children), replace the pointer to the node with a NULL pointer; Removing Nodes.

If either child of the node you want to remove is NULL, replace it with its non-NULL child;

If you try to remove a node with both a left and a right child, replace it with the rightmost node in the left subtree (predecessor) or the leftmost node in the right subtree (successor). Additional operations might be needed. balance themselves

A BST is balanced if the height of left & right subtrees differs at most one and if both sub are

```
void preorderTraversal(Node *t) {
   if (t != null) {
      cout << t->key << endl;
      preorderTraversal(t->left);
      preorderTraversal(t->right);
```

In-order: Traverse by Alpha order.

Post: useful when free the nodes

Partially ordered tree: En/Dequeue O(logN), is a BT but not BST

Properties: ① 从左到右填充 ② Root 有最高 priority.

Parentindex: (n-1)/2 leftchild: 2*n+1 rightchild: 2*n+2. Heapsort worst-case O(NlogN)

Set

Implement sets: ① Hash tables O(1). but don't support ordered iteration. ② Balanced binary trees. O(log N), but possible to write an iterated order.

Character Set: Characteristic/Indicator vector ⇒ O(1) 可打包为machine words.

Bitwise operator: x&y, x|y, x^y (exclusive or), ~x, 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 of x unsign)

& : masking; | : assemble; ^ : flips; ~ : complement; Arithmetic shifts are efficient ways to perform multiplication or division of signed int by power of two.

The expression *x* << *n* shifts the bits in the integer *x* leftward *n* positions. Spaces appearing on the right are filled with 0s.

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.

Two's complement: The sum of a number and its two complement is 2^N for a N-bit number

Simpler approach: 0, 1 倒转再加 1.

e.g.

0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
−128	1000 0000
−127	1000 0001
−126	1000 0010
−2	1111 1110
−1	1111 1111

i *= 2 ⇔ i <<= 1

i *= -1 ⇔ i = ~i + 1

Graph

Adjacency list, inefficient when dense (D→N)

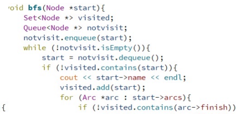
Adjacency matrix, inefficient when sparse (D<<N), set of arcs ✓

```
struct StringBasedGraph {
   Set<string> nodes;  /* A
   Set<string> arcs;   /* At
```

```
void visitUsingDFS(Node *node, Set<Node *> & visited)
   if (visited.contains(node)) return;
   visit(node);
   visited.add(node);
   for (Arc *arc : node->arcs) {
      visitUsingDFS(arc->finish, visited);
   }
}
```

dead end; like "cout"; take the finish point of the arc i.e. children of the original node

```
void dfs(Node *start){
   Set<Node *> visited;
   Stack<Node *> notvisit;
   notvisit.push(start);
   while (!notvisit.isEmpty()){
      start = notvisit.pop();
      if (!visited.contains(start)){
         cout << start->name << endl;
         visited.add(start);
         for (Arc* arc : start->arcs){
            if (!visited.contains(arc->finish)){
               notvisit.push(arc->finish);
```



```
oid bfs(Node *start){
   Set<Node *> visited;
   Queue<Node *> notvisit;
   notvisit.enqueue(start);
   while (!notvisit.isEmpty()){
      start = notvisit.dequeue();
      if (!visited.contains(start)){
         cout << start->name << endl;
         visited.add(start);
         for (Arc *arc : start->arcs){
            if (!visited.contains(arc->finish))
               notvisit.enqueue(arc->finish);
```

Inheritance

Can be protected/private

class *subclass* : **public** *superclass* {
body of class definition

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)

Overloading is about multiple functions/operators/methods of the same name but different signatures and possibly different implementations. when parameters are different, call the corresponding function

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***.

```
class Employee {
   virtual double getPay();=0
```

pure virtual method

Because of the pure virtual type method, the Superclass can't be instantiated ⇒ Abstract class (Object ✗ pointer ✓)

slicing

C++ throw away any fields in the assigned object that don't fit into superclass

```
Vector<Employee *> payroll;
payroll.add(&bobCratchit);
for (Employee *ep : payroll) {
   cout << ep->getName() << ": " << ep->getPay() << endl
```

若 pointer declared 为 superclass 指向 subclass, ① 无 virtual: superclass method called ② 有 virtual: Subclass method called.

```
class GCircle : public GOval {
   GCircle(double x, double y, double r)
                  : GOval(x-r, y-r, 2*r, 2*r){}
```

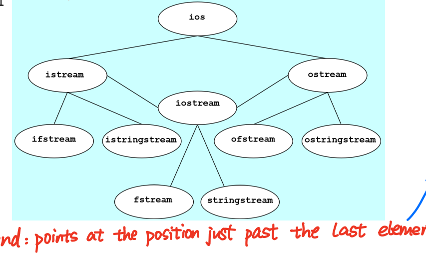
Initializer list

```
GRect::GRect(double x, double y, double width, double height) {
   this->x = x;
   this->y = y;
   this->width = width;
   this->height = height;
   filled = false;
}

void GRect::setFilled(bool flag) {
   filled = flag;
}

void GRect::draw(GWindow & gw) {
   gw.setColor(color);
   if (filled) {
      gw.fillRect(x, y, width, height);
   } else {
      gw.drawRect(x, y, width, height);
```

Multiple Inheritance



```
for (ctype::iterator it = c.begin(); it < c.end(); it++)
   . . . Body of loop involving *it . . .
```

Iterator

begin: Pointing at the beginning. end: points at the position just past the last element.

`double x;`	*Declares* `x` *as a* `double`.
`double list[n];`	*Declares* `list` *as an array of n* `doubles`.
`double *px;`	*Declares* `px` *as a pointer to a* `double`.
`double **ppx;`	*Declares* `ppx` *as a pointer to a pointer to a* `double`.
`double f(double);`	*Declares* `f` *as a function taking and returning a* `double`.
`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`.

MapAll Function

```
template <typename ValueType>
void Vector<ValueType>::mapAll(void (*fn)(ValueType)) const
   for (int i = 0; i < count; i++) {
      fn(array[i]);
```

```
template <typename FunctorType>
void Lexicon::mapAll(FunctorType fn) const
   for (std::string word : *this) {
      fn(word);
```

with overloading on ( ).

V.mapAll(printInt) ⇔ mapAll(printInt(V));

```
class ListKLetterWords {
public:
   ListKLetterWords(int k) {
      this->k = k;
   }
   int operator()(string word) {
      if (word.length() == k) {
         cout << word << endl;
```

Declarative: Programs state the logic of a computation (what is the result you want), but not its control flow (how to get it).

- **Functional**: Computation proceeds by (nested) function calls that avoid any global state (e.g., no assignments and loops).
- **Logic**: Programmer specifies a set of facts and rules, and an engine infers the answers to questions.

```
int neg = 0;
for (Vector<int>::iterator it = v.begin(); it != v.end(); it++)
   if (*it < 0) neg++;
```

int neg = count_if(v.begin( ), v.end( ), bind2nd(less<int>( ), 0));