2 C++ Basis

A simple C++ Program

```
#include<iostream>
using namespace std;
int main()
{
    cout << "Hello world" << endl;
    return 0;
}
```

- a program must have one and only one main function
- don't forget
- braces
- Each simple statement ends in a ";"

#include will include information of a library
#include <iostream> header file
cin: read cout: print, cerr: print error
#include "drawing.h" a user-defined header files

Simple C++ Data Types

Integers: int
Characters: char
'a', 'b', '4' ← single quotes
'\t' = tab, '\n' = newline
'\b' = backspace
'\0' = null character
Character strings: not a basic data type.
"hkust", "How are u?".

Binary numbers

A binary digit is "bit"
8 bits = 1 byte.
In C++, a char datum is represented by 1 byte (8 bits)

	ASCII Code	Integer Value
'0'	00110000	48
'1'	00110001	49
⋮		

C++ Variables

Rules for identifier:
0-9, a-z, A-Z, _
The first character cannot be 0-9.
C++ keywords are reserved.

Variables

A named memory location for a value.

```
int radius = 10, sum = 0;
```

- initialize when defined.
- data type
- identifier

```
int radius, num_of_words;
```

Defining several variables of the same type at once

```
int radius;
radius = 5;
```

Initialization by assignment

Operators

```
int a, b, x = 2, y = 4, z = 4;
```

= assignment operator

Arithmetic operator
-, +, *, /, %, ++, --

```
++x; x = x + 1. use the result
x++; use current x, then x = x + 1.
n += 2; n -= 2; n *= 2; n /= 2; n %= 2.
```

Associativity

-(minus), ++, --	Right to left
*, /, %, +, -	left to right
=	right to left

```
y = z = 5;
w = x = y + z;
```

cascading assignment.

expression: has a value
4 x-y 2-a-(b*c)
statement: a command.

```
cin >> x;
x = 5;
int x;
```

More Basic Data Types

Integers

short ≤ int ≤ long ≤ long long.
default → signed int: represent both +ve and -ve
unsigned int: only +ve

Floating-Point

float single precision
double double precision

Integer and Floating-Point Arithmetic

For division

```
10/2 = 5   10.0/2.0 = 5.0
 9/2 = 4    9.0/2.0 = 4.5
 4/8 = 0    4.0/8.0 = 0.5
```

If 1 operand is float/double, the result is float/double.

3 Program Flow Control

Boolean Data Type

```
bool
```

zero - false 0
non-zero - true 1

Type Checking and Type Conversion

Coercion

```
3 + 2.5 ⇒ 3.0 + 2.5
char small_y, big_y;
cin >> small_y;
big_y = small_y + 'A' - 'a';
```

coerced to int → converted back to char

```
int k = 5;
float x = static_cast<double>(k)
```

for manual type casting.

Constant

Literal constants:
fix/permanent values, cannot be modified.

Symbolic constants;

```
const char BACKSPACE = '\b';
```

If

```
int x;
cin >> x;
if (x < 0)
{ not neccessary.
    x = -x;
}
```

remember !!

```
int x, y, larger;
cin >> x >> y;
if (x > y)
    {larger = x;}
else
    {larger = y;}
```

not necessary

```
if (...)
    <statement>
else if (...)
    <statement>
⋮
else
    <statement>
```

Rational operator

==, <, <=, >, >=, !=.

x = y An assignment
x == y An equality comparison

Logical operators

!	NOT
||	OR
&&	AND

true: 1 ; non-zero number also consider true
false: 0;

Associativity

=, ++, --, !, - Right to left
others: Left to right

x = y result: final value of x
if x ≠ 0, true. x = 0, false

x == y a boolean expression

if-else operator

```
larger = (x > y) ? x : y;
```

"Dangling else" Problem

```
int x = 15;
if (x > 20)
    if (x > 30)
        x = 8;
    else
        x = 9;
```

groups a dangling else with the most recent if.

Loops or Iterations

```
int factorial = 1;
int number;
cin >> number;
while (number > 0)
{
    factorial *= number;
    number --;
}
```

① Necessary

Verification for a loop

Try: ① first iteration
② Second iteration
③ Last iteration.

```
for (int j=1; j<=number; j++)
{
    factorial *= j;
}
```

only defined inside the loop

Nested Loops

break : exist innermost enclosing loop
continue: cause the next iteration of the enclosing loop.

continue; in for loops: excute the update statement.

Switch

```
int section;
cin >> section;
switch (section)
{
    case 1:
        cout<<...; break;
    case 2:
        cout<<...; break;
    default:
        cerr<<...;
        break;
}
```

REMEMBER!

```
char bloodtype; cin >> bloodtype;
switch (bloodtype)
{
    case 'A':
        ...
        break;
    case 'B':
        ...
        break;
    case 'C':
    case 'D':
        ...
        break;
}
```

an integral value (int, char, bool)

When a const case matched stop when: a break, a return, the end of the switch statement

several cases may share the same action statements

enum

represented by integer

```
enum weekday {MON, TUE, WED, THUR};
                0    1    2    3
enum color {RED=1, GREEN, BLUE};
                   2      3
enum blood {A, B, AB=10, O};
            0  1     11
```

act like integer constant

Function definition & Call

```
int max (int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
}
int main()
{
    cout << max(5,8) << endl;
    return 0;
}
void func(void) {cout<<"hkust"<<endl;}
```

return type | function name | formal parameter list | function header | function body | return value

run this first | "argument" | actual parameter list | function call | "main" is paused. run "max" and pass values

could also have a return statement. return;

Pass By Values & Pass By Reference

PBV:

actual parameter { literal constant; variable: only value }

Would be copied, would not be modified by the function

PBR:

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x=10, y=20;
    swap(x,y);
}
```

local variables are destroyed on return of the function call.

For reference variable, only itself is destroyed.

10
int &a = x = 10
int &b = y = 20
a = x = 20
y = b = temp = 10

Formal parameter change ⇓ share memory
Actual parameter change.

x = 20, y = 10

const References as Func Arguments.

```
Void func(cost int &x) {x += 10}
```

Error

to keep a reference argument unchanged, could not be modified.

For reference parameter	const func argument	non-const func argument
literal const	✓	✗
const object	✓	✗
non-const object	✓	✓

4
Functions I

5 Array

1-Dimensional Array

```
const int NUM_STUDENTS=5;
float score[NUM_STUDENTS];
```

data type · array name · positive constant or constant expression

`x[0] x[1] … x[N-1]`

subscript starts from 0.

Define and Initialize a 1D array:

```
int x[6]={1,2,3,4,5};
int x[]={1,2,3,4,5};
int z[5]={0,0,0,0,0};
```

The elements of array is PBR !!! Changes to the elements inside the function will persist.

Arrays as Function Argument

```
float mean_score(float score[], int size){…}
```

Function definition — array identifier — array size

```
mean_score(score,size);
```

Function call — array identifier

```
const int x[]={1,2,3,4};
```

disallow modification when passed to a function

```
float mean(const float score[], int size).
```

Multi-dimensional Array

Initialize a 2D-array

```
int point[5][2]={
  {1,1}
  {2,4}
  {3,9}
  {4,16}
  {5,25}
};
```

5 rows and 2 columns

row · column

```
int point[5][2]={1,1, 2,4, 3,9, 4,16, 6,25};
```

A 2-D arry A[M][N], its i-th row could be treated as a 1-D arry. Say A[i] is a 1-D array with N elements.

Definition of N-Dimensional Array.

```
int a[2][2][2]={1,2,3,4,5,6,7,8};
```

Pass a nD array to the function

Say an array: `int array[5][6][7];`

```
void f(int array[][6][7], int size1…
```

first one is emty · others need the size

`f(array, 5, 6, 7)` Function call

C string

Representing a character string by a 1D character array with end-marker '\0'

null character ASCII code = 0.

"hkust"='h''k''u''s''t''\0'

```
char s1[6]={'h','k','u','s','t','z'};
s1[3]=0; for(int i=0; i<6; i++)
cout<<s1[i] //result: hkutz
cout<<s1; // result: hku.
```

Initialization

```
char a[20]="Albert"; char b[20]="Einstein".
```

Reading C strings with cin.

cin would skip all white spaces before reading data of the required type until it sees the next white space. ' ' '\t' '\n'

```
char x; cin>>x;   " hkust "=>'h'
char a[20]; cin>>a;   " hkust "=>"hkust"
cin.getline(char s[]; int max_num_char, char terminator);
```

to put in a '\0'

stop: ① (max_num_char-1) are read or ② reach terminator.

6 Function II: Recursion

Programming Recursion in C++

Recursion: A function calls itself

Probems —Decompose→ Subproblems.

Smallest sub-problem has a non-recursive solution

e.g. Factorial function

```
int factorial (int n)
{
  if (n<0)
    return -1;          Error checking
  else if (n==0)
    return 1;           Base case
  else
    return n*factorial(n-1);   Recursive case
}
```

factorial 2

```
2<0 false
2=0 false
2*factorial(1)
    1<0 false
    1=0 false
    1*factorial(0)
        0<0 false
        0=0 true
        return 1;
    return 1;
return 2;
```

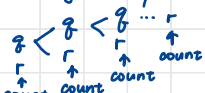
A Recursion should eventually end up with a non-cursive base case.

e.g. Fibonacci Function

```
int fibonacci(int n)
{
  if (n=0)
    return 0;          Base case #1
  if (n=1)
    return 1;          Base case #2
  return fibonacci(n-1)+fibonacci(n-2);   ↓Recursive
}
```

e.g. Counting zeros in an integer.

Idea: counting zeros in quotient & remainder.




```
int num_zeros(int n)
{
  if (n==0)
    return 1;                    Base case #1
  else if (n<10 && n>0)
    return 0;                    Base case # 2.
  else
    return num_zeros(n/10)+num_zeros(n%10).
}
```

e.g. Factoring

factor m, integer n

```
int factor(int n, int m)
{
  if (n%m!=0)
    return 0;          not a factor  Base case
  else
    return 1+factor(n/m,m);
}
```

count · Find in the quotient

e.g. Tower of Hanoi


```
void hanoi(int num, char pA, char pB, char pC).
{
  if (num==0)
    return;
  hanoi(num-1, pA, pC, pB)
  cout<<"move disc"<<num
      <<"from peg"<<pA<<"to peg"<<pC<<endl;
  hanoi(num-1, pB, pA, pC)
}
```

meaning: moving (num) discs from pA to pC.

move (num-1) discs from pA to pB

move (num-1) discs from pB to pC;

Advantage: greater productivity

Disadvantages:

More memory, more computational time

Scope

Scope is the region of codes in which an identifier declaration is active.

Location of declaration —determine→ scope

2 kinds of scope:
- global scope: outside any function
- local scope: inside a function.

3 kinds of scope:
- file scope
- function scope
- block scope.

e.g. File/function/block scope.

```
void my_print(const int b[], int size)
{
    for (int j=0; j<size; j++)
    {
        int k=0;
    }...
}
int a[] = {1,2,3,4,5};
int main(){
    int num_array = ...;
}
```

b, size: local, function scope
j: local, block scope.
k=0: local, block scope.
a[]: global, file scope.
num_array: local, function scope.

7 Scope

C++ Structures

Struct: collection of heterogeneous objects.

Struct definition

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

REMEMBER! Point ← Struct identifier; x, y ← identifier for the member; double ← data type

Define/Declare a struct variable

```
Point a,b;
```

a, b contain garbage

8 Struct

.operator

```
a.x = 24.5;
a.y = 123.0;
```

Initialize a Point struct by memberwise assignments

```
cin >> a.x >> a.y;
cout << b.x << b.y;
```

Struct-struct assignment

Done by memberwise copy.

Even a member array can be copied.

```
struct Example
{
    int x;
    float y[5];
};
Example a,b;
b=a;
```

b.y=a.y is not allowed, however

no b.y=a.y

File scope = global scope.

global variables —access→ any function

initialized to 0 without an explicit initializer.

```
int number; =0.
```

All function identifiers have file scope (global)

⇒ Try to avoid using global variables

⇒ Use only local variables, pass between functions

Function Scope

One kind of local scope.

(local) variables/const:
- formal parameter list → have function scope.
- inside function body → have function scope.
- function call: created.
- function return: destructed.

access within function

Block Scope

Also a kind of local scope.

A block of codes.

for, while, do-while, if, else, switch, { }

Variable/Const (local) —use→ within block
- enter the block: created.
- finish the block: destructed.

Initialization

```
Point a = {24.5, 123.0};
```

initialized when defined.

If not initialized, then

```
b.x = 24.5; b.y = 123.0;
```

or `b=a;`

PBR

```
void print_record(const Student_Record& x)
{
    ⋮
```

data type

Arrays of Structures

```
int main()
{
    Student_Record sr[] = {
        {"Adam", 12000, 'M', CSE, {2006,1,10}},
        {"Bob", 11000, 'M', MATH, {2005,9,1}}
    };
...
```

array type; array name

Identifiers of the same name.

An identifier can only be declared once in the same scope.

Same name may be reused in different scope.

Compiler Scope Rule: When an identifier is declared more than once but under different scopes, the compiler associates an occurance of the identifier with its declaration in the innermost enclosing scope.

e.g.

```
int main()
{
    int j; ⇒ S1, S5, S6
    int k; ⇒ S1, S2, S3, S4, S6
    S1;
    for(...)
    {
        int j; ⇒ S2, S4
        S2
        while(...)
        {
            int j ⇒ S3
            S3;
        }
        S4;
    }
    while(...)
    {
        int k; ⇒ S5
        S5;
    }
    S6;
}
```

10 Pointer & Dynamic Data

Part 1 lvalue (address) and rvalue (content)

A variable is addressable
literal constant
lvalue : location of the memory storage (read/write)
rvalue : value in the storage (read only)

```
x = x + 1
↑   ↑
lvalue rvalue
```

++x: pre-increment
PBR, return by reference (lvalue)
x++: post-increment
PBR, return by value (rvalue).

```
++++++x ✓
 lvalue
x++++++ ✗
 rvalue
```

Syntax: getting address of a variable
`&<variable>`

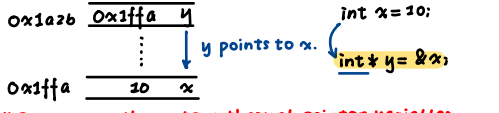
```
int x=10; cout << &x << endl;
```

PBR: create a new variable name sharing the same address.

Part 2 Pointer

Syntax: Pointer Variable definition
`<type>* variable>;`

Pointer variables store the address of another variable



Syntax: get the content through pointer variables
`*<pointer variable>.`

```
int x=10, int *y=&x;
z = *y; // z=x, ⇒ z=10.
```

e.g. pointer manipulation

```
int x1=10, x2=20;
int *p1=&x1; int *p2=&x2;
*p1=5; // x1=5
*p2+=1000; // x2+=1000 ⇒ x2=1020
*p1=*p2; // x1=x2 ⇒ x1=x2=1020
p1=p2; // p1=p2=&x2.
```

e.g. pointer to pointer to pointer

```
int x=10;
int *xp=&x;      xppp→xpp→xp→x.
int **xpp=&xp; *xp=**xpp=***xppp.
int ***xppp=&xpp;
```

e.g. reference variable & pointer variable *xptr=x =xref

```
int x=5; int &xref=x; int *xptr=&x;
```

Syntax: const Pointer definition
`<type>* const<pointer variable>=&<variable>.`

① must be initialized when defined.
② once initialized, cannot changed to point sth else
③ free to change the content in the address it points to

```
int x=10, y=20; int* const xcp=&x;
xcp=&y; Error  *xcp=5; x=5.
```

Syntax: Definition of pointer to a const object
`const <type>*<pointer variable>;`

① no necessary to initialize a pointer to const object when defined.
② Free to change the pointer to different objects
③ Content of the object cannot be changed through pointer. But can be changed by the object directly.

```
int x=10, y=20; const int* pc=&x;
pc=&y; point to y *pc=5; Error  y=8; okay.
const int* const p;
```

PBR = PBV + pointer

```
void swap(int *x, int *y)
{
  int temp=*x; *x=*y; *y=temp;
}
int a=10, b=20; swap(&a, &b);
```

Part 3 : Pointer to Structure

```
struct Point
{
  double x;
  double y;
};
Point a;
Point* ap=&a; ap→a.
```

Two ways access struct:

```
(1) (*ap).x=3.5;
    (*ap).y=9.7;
(2) ap->x=3.5;
    ap->y=9.7;
```

Part 4 Dynamic Memory / Objects Allocation and Deallocation

Static objects	managed by stack
Dynamic objects	allocated at runtime explicity persist even the object goes out of scope deallocated at runtime explicity managed by heap

```
program codes
(global)
static data
stack
  ↓
  ↑
heap
```

Syntax: Dynamic Memory Allocation Using new
`<type>*<pointer variable>=new <type>;`

```
int *ip=new int;  *ip=5;
```

① Computer finds from heap an amount of memory equals to sizeof(int)
② new returns the address of the memory
③ The memory is unnamed, the only way to access it is the pointers.

```
Student_Record* srp=new Student_Record;
```

static object — dynamic object

Syntax: Dynamic Memory Deallocation using delete
`delete <pointer variable>;`

```
delete ip;     // ip: a dangling pointer.
ip=nullptr;    // ip: a null pointer.
```

Bug 1 : Dangling pointer.
Modify the object a dangling pointer points to → crash.
① Always initialize a pointer to nullptr when defining.
② Always check a pointer if nullptr before use.

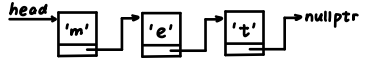
Bug 2 : Memory leak
Lose track of an unnamed memory → memory leak.
Leak a lot of memory → runtime error.

Part 5 A Dynamic Data Structure : Linked List

A linked list links object together by pointers ⇒ each object points to the next object in sequence.
Objects in a linked list are called nodes.

```
struct ll_node
{
  <type> data;
  ll_node *next;
};
ll_node *p=new ll_node; // create a node
cout << p->data; cout << (*p).data;
cin >> p->data; p->next=nullptr;
// set the head.
ll_node *head=nullptr; head=p;
// delete
delete p;
p=nullptr;
```

e.g. LL-String




```
struct ll_cnode
{
   char data;
   ll_node *next =nullptr;
};
const char NULL_CHAR= '\0';
int main(){
   // creat the LL-string "met"
   ll_cnode *mp= new ll_cnode; mp->data='m';
   ll_cnode * ep= new ll_cnode; mp->data='e';
   ll_cnode * tp= new ll_cnode; mp->data='t';
   // link them in required order
   mp->next=ep; ep->next=tp; tp->next=nullptr;
   // print out the data
   for(ll_cnode *p= mp; p; p= p->next)
      cout<< p->data;
   // clean up
   delete mp; delete ep; delete tp; return 0;
}
```

ll_create.cpp

```
// create a ll_cnode and initialize the data
ll_cnode* ll_create(char c)        p->next=nullptr
{   return type is a pointer          V
  ll_cnode*p= new ll_cnode; p->data=c; return p;
}
ll_cnode* ll_create(const char s[]) // create a linked list
{
    if (s[0]==NULL_CHAR)
       return nullptr;          a pointer to the struct whose
                                   data is s[0].
   ll_cnode * head = ll_create(s[0]);
   ll_cnode * p= head;
   for(int j=1; s[j]!= NULL_CHAR; j++)
   {
      p->next= ll_create(s[j]);
      p= p->next;
   }
   return head;
}
```

Now the whole LL can be accessed from the head.

ll_length.cpp

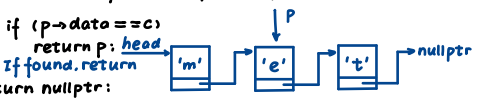
```
int ll_length(const ll_cnode * head)
{                          go through the whole linked list
   int length=0;                  V
   for(const ll_cnode *p=head; p!=nullptr; p=p->next)
      ++length;
   return length;
}
```

ll_print.cpp

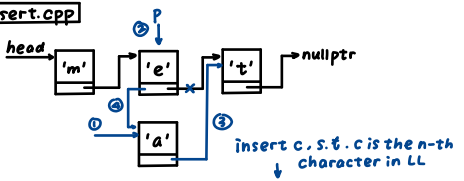
```
void ll_print (const ll_cnode * head)
{

  for(const ll_cnode *p=head; p!=nullptr; p=p->next)
     cout<< p->data;
  cout<<endl;
}
```

ll_search.cpp

```
ll_cnode * ll_search(ll_cnode * head, char c)
{
   for(ll_cnode * p= head; p!=nullptr; p= p->next)
   {
      if (p->data==c)
         return p;          head
   }  If found, return
   return nullptr;
}
```



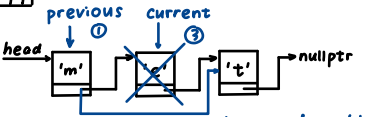
ll_insert.cpp



insert c, s.t. c is the n-th character in LL

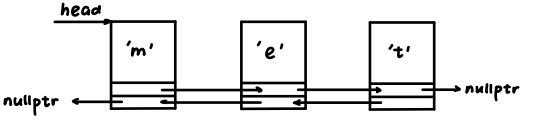
```
void ll_insert( ll_node* &head, char c, unsigned n)
{   Step 1:                  PBR
    ll_node * new_cnode = ll_create(c)
    if (n==0 || head==nullptr)
    {                                  change the head
      new_cnode->next = head;          pointer.
      head = new_cnode;
      return;
    }
    Step 2:
    ll_cnode *p=head;
    for(int position=0; position < n-1 && p->next != nullptr;
          p=p->next, position++)
          ; find the n-1 position, now p is pointing to the
            (n-1)th node
    new_cnode->next = p->next; Step 3
    p->next = new_cnode; Step 4
}
```

LL_delete.cpp

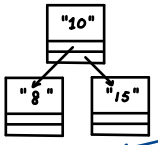



```
void ll_delete(ll_cnode * &head, char c)
{
   ll_cnode * prev = nullptr;
   ll_cnode * current = head;
   Step 1:
   while (current!=nullptr && current->data !=c)
   {                                  Two cases:
      prev= current;                  ① c is in the list.
      current = current->next;        ② c is not in the list. then
   }                                    current would run to the
   //now current is found              end of the list.
   if (current != nullptr)
   {
     if (current == head)
        head= head->next;
     else
        prev->next = current->next;
     delete current;                  PBR
   }
   void ll_delete_all (ll_cnode *&head)
   {
      if (head == nullptr)    // Base case
         return;
      ll_delete_all (head->next)   Recursion
      delet head;
      head= nullptr;
   }
```

Double linked list



Part 6 Binary Tree




```
struct btree_node
{
    int data;
    btree_node * left;
    btree_node * right;
};
```

the return type is a structure

```
btree_node * creat(int data)
{
    btree_node * node = new btree_node;
    node -> data = data;
    node -> left = node->right = nullptr;
    return node;
}
```

PBR, do operations on "root"

```
void delete_btree(btree_node * &root)
{
    if (root == nullptr)
        return;
    delete_btree(root->left);
    delete_btree(root->right);
    delete root;
    root = nullptr;
}
```

Recursion

in case changing the data

```
void print_btree(const btree_node *root, int depth)
{
    if (root == nullptr)
        return;
    print_btree(root->right, depth+1);
    for (int j=0; j<depth; j++)
        cout << '\t';
    cout << root->data << endl;
    print_btree(root->left, depth+1);
}
```

Base case

Recursion

To make it like a tree.

Recursion

Part 7 Array as a Pointer

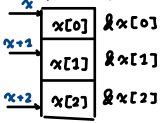
Pointer Arithmetic

<type> x; <type> * xp = x;

xp + N is &x + sizeof(<type>) × N

xp - N is &x - sizeof(<type>) × N

The result should be a valid address, or else: segementation fault.



int x[3];

The array identifier can be treated as a pointer to the first array element.

The variable x is like int * const.

int x[] = {11, 22, 33}; int * xp = x;

xp == x == &x[0], *xp == *x == x[0]

&x == x == &x[0].

```
int x[] = {1, 2, 3, 4};
for (int *xp = x, j=0; j < sizeof(x)/sizeof(int); ++j, ++xp)
    cout << *xp << endl;
```

```
char s[] = "hkust";
for (const char *sp = s; *sp != '\0'; ++sp)
    cout << *sp;
```

Syntax: new a dynamic array

<type> * <pointervariable> = new <type>[<integer>];

```
int array_size; cin >> array_size;
int * x = new int [array_size];
for (int j<0; j< array_size; ++j)
    x[j] = j;
```

treat the pointer as an array

Syntax: delete a Dynamic Array

delete[]<pointer-variable>;

```
delete []x; x = nullptr;
```

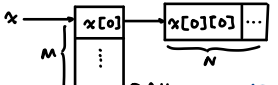
For C-string: const char * s1 = "creative";

e.g.

```
bool palindrome(const char *first, const char *last)
{
  if (first >= last)
    return true;
  else if (*first != *last)
    return false;
  else
    return palindrome(first+1, last-1);
}
```

Base case: they swap order ⇒ true

Part 8 Multi-dimensional Array and Pointer



① Allocate a 1D array of M int *

② For each int *, creat another 1D array of N int.

e.g. Operations of a Dynamic 2D array

```
int ** create_matrix(int num_rows, int num_columns)
{
    int ** x = new int* [num_rows];
    for (int j=0; j<num_rows; j++)
        x[j] = new int[num_columns];
    return x;
}
```

create a 1D int * array

```
void print_matrix(const int * const x, int num_rows,
    int num_columns){
    for (int i=0; i< num_rows; ++i)
    {
        for (int j=0; j<num_columns; ++j)
            cout << x[i][j] ;
        cout << endl;
    }
}
```

```
void delete_matrix(const int * const x, int num_rows,
    int num_columns){
    for (int j=0; j<num_rows; ++j)
        delete []x[j];
    delete []x;
}
matrix = nullptr;
```

A C++ Class contains:
data member, member functions (public, private)

e.g student_record /* student-record.h */

```
const int MAX_NAME_LEN=32;
class student_record
{
  private:
    char gender;
    unsigned int id;
    char name[MAX_NAME_LEN];
  public:
    // ACESSOR member functions: won't modify data
    // members
    char get_gender() const {return gender;}
    unsigned int get_id() const { return id;}
    void print() const
        {cout<< name << id << gender << endl;}
    // MUTATOR member functions
    void set(const char my_name[], unsigned int my_id
    char my_gender)
      {strcpy(name, my_name); id=my_id;
       gender=my_gender;}
    void copy(const student_record &r)
    {set(r.name, r.id, r.gender);}
};

#include <iostream>  /* student-record.cpp */
using namespace std;
#include "student-record.h"
int main()
{
  student_record amy, bob; //create 2 static objects
  amy.set("Amy", 12345, 'F');
  bob.set("Bob", 34567, 'M');
  cout << amy.get_id() << endl;
  amy.copy(bob);
  amy.print();
  return 0;
}
```

e.g. temperature class definition

```
#include <iostream>    /* temperature.h */
#include <cstdlib>
using namespace std;
const char KELVEN='k', CELSIUS='C',
    FAHRENHEIT='F';

class temperature
{
  private:
    double degree;
  public:
    //CONSTRUCTOR
    temperature();  // Default constructor
    temperature(double d, char s);
    //ACCESSOR
    double kelvin() const;
    double celsius() const;
    double fahrenheit() const;
    // MUTATOR
    void set(double d, char s);
};
/* temperature_functions.cpp */
#include "temperature.h"
//CONSTRUCTOR
temperature::temperature() {degree=0.0;}
temperature::temperature(double d, char s) {set(d,s);};
//ACCESSOR
double temperature::kelvin() const {return degree;}
double temperature::celsius() const
  {return degree-273.15;}
double temperature::fahrenheit()const
  {return (degree-273.15)*9.0/5.0+32.0;}.
//MUTATOR
void temperature::set(double d, char s)
{
  switch(s):
  {
    case KELVIN: degree=d; break;
    case CELSIUS: degree=d+273.15; break;
    care FAHRENHEIT:degree=...; break,
    default: ...
  }
  if (degree<0.0)
  {
    cerr<<...<<endl;
    exit(-2);
  }
}
/* temperature_test.cpp */
#include "temperature.h"
int main()
{
  char scale;
  double degree;
  temperature x;  //default constructor
  cout<<"Enter temperature:"
  while (cin>>degree>>scale)
  {
    x.set(degree, scale);
    cout<<x.kelvin()<<"K"<<endl;
    cout<<x.celsius()<<"C"<<endl;
    cout<<x.fahrenheit()<<"F"<<endl;
    cout<<endl<<"Enter temperature:";
  },
  return 0;
}
```

An object is constructed when it is
(1) defined in a scope (static)
(2) created with the new operator (dynamic)
An object is destructed when
(1) it goes out of scope (static)
(2) it is deleted by the delete operator (dynamic)
By creating an object, a constructor may initialize the content.
A class may have more than 1 constructor, but only 1 destructor.
The default constructor just reserves a big enough memory (no intialization)
The default destructor relased the memory acquired by the object.

```
temperature::temperature() {}
temperature::~temperature() {}.
```

e.g. Books

```
/* book.h */
class Book
{
  private:
    char * title;
    char * author;
    int num_pages;
 public:
    Book(int n=100) {title=author=nullptr;
      num_pages=n;}
    Book(const char * t, const char a, int n=5)
      {set(t,a,n);}
    ~Book();
    {
      cout << "Delete the book titled \"" << title
       << "\"" << endl;
    delete [] title; delete [] author;
    }
```

```
  void set(const char *t, const char * a, int n)
  {
      title = new char [strlen(t)+1]; strcpy(title,t);
      author = new char [strlen(a)+1]; strcpy(author, a);
      num_pages = n;
  }
};
/* book.cpp */
# include "book.h"
void make_books()
{
   Book y("Love", "HKUST", 88);
   Book * p = new Book [3];
   p[0].set("book 1", "author 1", 1);
   p[1].set("book 2", "author 2", 2);
   p[2].set("book 3", "author 3", 3);
   delete []p; // Call the class destructor
   return;     // delete in reverse order
}
int main()
{
   Book x("Sapiens", "Y", 1000);
   Book * z = new Book("Outliers", "Gladwell", 300);
   make_books(); cout << endl; delete z;
   return 0;
}
```

e.g. lamp and bulb

```
/* bulb.h */
class Bulb
{
  private:
    int wattage;
    float price;
  public:
    int get_power() const;
    float get_price() const;
    void set(int w, float p);
};
/* bulb.cpp */
# include "bulb.h"
int Bulb::get_power() const {return wattage;}
float Bulb::get_price() const {return price;}.
void Bulb::set(int w, float p)
    {wattage = w; price = p;}
```

```
/* lamp.h */
class lamp
{
  private:
    int num_bulbs;
    Bulb * bulbs;  // A dynamic array
    float price;  // price of the lamp
  public:
    Lamp(int n, float p)
    ~Lamp();
    int total_power() const;
    float total_price() const;
    void print(const char * prefix_message) const;
    void install_bulbs(int w, float p);
};
/* lamp.cpp */
# include "lamp.h"
# include <iostream>
using namespace std;
Lamp::Lamp(int n, float p)
   {num_bulbs = n; price = p; bulbs = new Bulb[n];}
Lamp::~Lamp() {delete [] bulbs;}
int Lamp::total_power() const
    {return num_bulbs * bulbs[0].get_power();}
float Lamp::total_price() const
    {return num_bulbs * bulbs[0].get_price();}
void Lamp::print(const char * prefix_message) const
{
   cout << prefix_message << ": total power = " << total_power()
        << "W" << ", total price = $" << total_price() << endl;
}
void Lamp::install bulbs(int w, float p)
{
   for(int i=0; i<num_bulbs; i++)
     bulbs[j].set(w, p);
}
/* lamp_test.cpp */
int main()
{
   Lamp lamp1(4, 100.5); Lamp lamp2(2, 200.6);
   lamp1.install_bulbs(20, 30.1);
   lamp1.print("lamp 1");
   return 0;
}
```

Part I Declaration and Definition

A function prototype

```
float euclidean_distance(float, float)
```

return type — function name — number, data type of formal parameters

A function is declared by writing down its function prototype; A function is defined by writing down its function header + body

Declaration can be made many times, but definition can be made only once.

```
bool even(int);
bool odd(int x)
    {return (x==0)? false : even(x-1);}
bool even(int x)
    {return (x==0)? true : odd(x-1);}.
```

9
Function
II

Part II Function overloading

Two C++ functions can have the same name but different signature ⇒ function overloading.

Function resolution: decide which function should be called. Best match:

① exact match

② match after promotion:
char/bool/short → int. float → double

③ match after standard type conversion

```
int test(int a, double b);
int test(double a, int b);
call test(3.2, 4.6) → compilation error
```

Default argument: at the end of the formal parameter list.

```
void func(int, float &, char = 'M', bool = true);
```

The default arguments must be specified in a declaration/definition, but not both