Content

First program

```
#include <iostream>   //header file lib
using namespace std;   //use names for objects and var from the standard lib
int main(){
    cout << "Hello World!";   //output func cout; insertion operator to print text
    return 0;   //ends main func
}
```

Omitting Namespace

```
#include <iostream>
int main(){
    std :: cout << "Hello World!";   //std with two semicolons instead of namespace std lib
    return 0;
}
```

Output (print text)

cout, together with the << operator, to print the text, but no new line character with this func.

```
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World! ";
    cout << "I love u";
    return 0;
}   //this will print "Hello World! I love u"
```

New line character and escape sequence

```
#include <iostream>
using namespace std;
int main(){
      cout << "Hello World! \n";   //first way to enter the other line
      cout << "Hello World!" << endl;   //second way to enter the other line
      cout << "I love u";
      return 0;
}
```

\t creates a horizontal tab

\\ inserts a backslash character

\" inserts a double quote character

\' inserts a single quote character

\n new line

\t tab

Variables

type variableName = value;   //type is one of var types, variableName is the name

assign val before:

```
int myNum = 15;
cout << myNum;   //print 15
```

assign val later:

```
int myNum;
```

```
myNum = 15;
cout << myNum;
```

NOTICE: New assigned val to an existing var will overwrite the previews val

@What is the difference between float and double type?

Double has a precision of 15 digits, float has 6 or 7 digits. So double is safer.

Display variables and text:

```
int myAge = 35;
cout << "I am " << myAge << " years old."; //print out "I am 35 years old."
```

Add var together:

```
int x = 5;
int y = 6;
int sum = x + y;
cout << sum;
```

Declare many var:

```
int x = 5, y = 6, z = 50;  //same type for multiple var
cout << x + y + z;
int a, b, c;
a = b = c = 50;  //assign same val to multiple var
cout << x + y + z;
```

Identifiers

Names are case sensitive; can contain underscore.

Constant

Use constant to prevent override existing var val.

Always declare the var as const when have val that are unlikely to change.

```
const int myNum = 15;  //myNum will always be 15
myNum = 10;  //error: assignment of read-only var 'myNum'
```

User input

cin reads data from the keyboard with extraction operator *>>* (see-in operator)

```
int x;
cout << "Type a number: ";  //Type a number and press enter
cin >> x;  //Get user input from the keyboard
cout << "Your number is: " << x;  //Display the input value
```

C++ Data Types

boolean: true or false (1 byte); true is 1, false is 0; default is true

char: store a single character or ASCII values (1 byte)

int: store whole numbers without decimals (2 or 4 bytes)

float: store fractional num, max 7 dec digits (4 bytes)

double: store fractional num, max 15 dec digits (8 bytes)

scientific nums -> e as the power of 10

string: store a sequence of characters (text); use double quotes; need #include <string>

1.Variables and Data Types 变量与数据类型

Basic Types 基本数据类型

Identifier	Type Family	Storage Size	Value Range	printf	scanf
int	Integer Type	4 bytes (32 bits)	-2^{-31}~$2^{31}-1$	%d	%d
double	Floating-Point Type	8 bytes (64 bits)	(±) 2.2E-308 ~ 1.8E308	%f	%lf
char	Integer Type	1 byte (8 bits)	−128~127 or 0~255	%c or %d	%c
bool	Derived Type(C99)	1 byte (8 bits)	False (0) or True (1)	%d	None

⚠ You must write **#include <stdbool.h>** at the top of your program to use the Boolean type.

C++ Operators

Arithmetic Operators: +, -, *, /, %, ++(increment; unary operator), --(decrement; unary operator)

@What is the difference between ++i and i++?

++i means it will increment i then display/sent the result (already incremented). i++ means it will display/send i (without any increment) and then increment i. ++i returns the value of i+1, i++ returns the value of i.

Assignment Operators: =, +=, -=, *=, /=, %=(取余并重新赋值余数), &=(FA), |=(FB), ^=(FC), >>=(FD), <<=

FA: &, bitwise AND, takes two nums as operands and does AND on every bit of two binary nums, result of AND is 1 only if both bits are 1.

FB: |, bitwise OR, result of OR is 1 if any of the two bits is 1.

FC: ^, bitwise XOR, result of XOR is 1 if the two bits are different.

FD: <<, left shift, left shifts the bits of the first operand, the second operand decides the num of places to shift.

FE: ~, bitwise NOT, takes one num and inverts all bits of it.

Comparison Operators: ==, !=, >, <, >=, <=

Logical operators: &&, ||, !(reverse the result and returns false if the result is true)

More about String

String Concatenation: + to add two strings together to make a new string.

Append: A string in C++ is an object, contain functions that can perform certain operations on strings.

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName.append(lastName);   //use string as an object by using append func
cout << fullName;   //output "John Doe"
```

NOTICE: C++ uses the + operator for both addition and concatenation. Nums are added, strings are concatenated. So, cannot add a num to a string, it will show an error.

string Length: length( ) or size( )

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
cout << txt.length( );   //output 26
cout << txt.size( );   //output 26
```

Access strings: referring to its index num inside square brackets.

User input strings: *cin* >> to read user inputs.

@Problem encounter: *cin* considers a space (whitespace, tabs, etc.) as a terminating character, so if input is "John Doe", it will only print "John". How to solve this?

Use the *getline( )* function to read a line of text. It taks cin as the first parameter, and the string var as the second:

```
string fullName;
cout << "Type your full name: ";
getline (cin, fullName);
cout << "Your name is: " << fullName;   //output John Doe
```

Math

Other funcs such as square root, rounds a num and natural log, can be found in the <cmath> header file: *#include*

<cmath>

Max and min: max(x, y); min(x, y);

Absoute val: abs(x)

Trig: acos(x), asin(x), atan(x); cos(x), tan(x), sin(x); cosh(x), tanh(x)

Cube root: cbrt(x)

X rounded up to its nearest integer: ceil(x)

Exponential: exp(x), expm1(x)

Floating point remainder of x/y: fmod(x,y)

X rounded down to its nearest integer: floor(x)

Booleans

A data type that can only have one of two vals: yes/no, on/off, true/false; boolean var take one from true/false.

bool: true returns 1, false returns 0

Boolean and comparison operator: use >, == to find out if an expression or a var is true.

```
int x = 10;
int y = 9;
cout << (x > y);    //returns 1 (true), because 10 is higher than 9
```

Condition statements

Use if to specify a block of code to be executed, if a specified condition is true.

Use else to specify a block of code to be executed, if the same condition is false.

Use else if to specify a new condition to test, if the first condition is false.

Use switch to specify many alternative blocks of code to be executed.

If...else statement:

```
if (condition) {
//block of code to be executed if the condition is true
} else {
//block of code to be executed if the condition is false
}
```

Else...if statement:

```
if (condition1) {
//block of code to be executed if condition1 is true
} else if (condition2) {
//block of code to be executed if the condition1 is false and condition2 is true
} else {
//block of code to be executed if the condition1 is false and condition2 is false
}
```

Ternary operator (short hand if...else):

Can be used to replace multiple lines of code with a single line, often used to replace simple if else statements.

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Switch statements:

```
switch(expression) {    //evaluated once; the value of the expression is compared with the val of each case
                        //if there is a match, the associated block of code is executed
                        //the break and default keywords are optional
    case x:
        //code block
        break;
    case y:
```

```
        //code block
        break;
    default:
        //code block
}
```

Break keyword: breaks out of the switch block. It will stop the execution of more code and case testing inside the block. When a match is found, and the job is done, there needs a break.
It can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.
Default keyword: specifies some code to run if there is no case match.
It must be used as the last statement in the switch, and it does not need a break.

Loops

Loops can execute a block of code as long as specified condition is reached. They are handy because they save time, reduce errors, and they make code more readable.

While loop

It loops through a block of code as long as a specified condition is true:

```
while (condition) {
//code block to be executed; remember to increase the var used in the condition, or it will reach an infinite loop
}
```

Do/While loop

A variant of the while loop. It will execute the code block once, before checking if the condition is true then it will repeat the loop as long as the condition is true.

```
do {//code block to be executed; remember to increase the var used in the condition, otherwise reach an infinite loop
}
while (condition);
```

For loop

When know exactly how many times you want to loop through a block of code.

```
for (initialization; condition; increment or decrement){
//code block to be executed
}
```

Continue keyword: breaks on iteration in the loop, if a specified condition occurs, and continues with the next iteration in the loop.

Arrays

Used to store multiple values in a single var, instead of declaring separate var for each val.

Declaration: define the var type, specify the name of the array followed by square brackets and specify the num of elements it should store.

```
varType name[num of elements] = {element1, element2};    //character or string need to be in double quote
```

Access the elements: referring to the index number inside square brackets.

Loop through the array: by changing index num.

Omit array size: if dont specify, it will be as big as the elements that are inserted into it. Problem arise if you want extra space for future elements. If specify the size, the array will reserve the extra space.

Omit elements on declaration: declare an array without specifying the elements on declaration, and add them later.

Get the size of an array: *sizeof( )*; returns the size of a type in bytes. So to find out how many elements an array has, you have to divide the size of the array y the size of the data type it contains.

```
int myNumbers[5] = {10, 20, 30, 40, 50};
cout << sizeof(myNumbers);
```

Multi-Dimensional Array: array can have any number of dimensions. Access by specify the index. The first one is row,

second dimension is col.
It is great at representing grids.

Structures (struct)

A way to group several related var into one place. Each var in the structure is a member of the member of the structure. A structure can contain many different data types.

Create a structure: using the struct keyword and declare each of its members inside curly braces. After the declaration, specify the name of the structure var.

```
struct {
    Int myNum;   //strcture declaration
    String myString;   //member (string var)
} myStructure;   //structure var
```

Access structure members: use the dot syntax (.): *structureName.memberName*

One structure in multiple var: use a comma to use one structure in many vars

```
struct {
    int myNum;
    string myString;
} myStruct1, myStruct2, myStruct3;   //multiple structure vars separated with commas
```

Named structures: giving a name to the structure, means can treat it as a data type. To create a named structure, put the name of the structure right after the struct keyword. To declare a var that uses the structure, use the name of the structure as the data type of the var.

```
struct myDataType {
    int myNum;
    string myString;
};
myDataType myVar;   //to declare a var that uses the structure, use it as the data type
```

References

A reference var is a "reference" to an existing var, and it is created with the & operator. Change the val of the reference is the same as change the val of the original var.

```
string food = "Piazza";   //food var
string &meal = food;   //reference to food
```

Memory Address

& to get the memory address of a var, which is the location of where the var is stored on the computer. The memory address is in hexadecimal form.

@Why it is useful to know the memory address?

References and Pointers give the ability to manipulate the data in the computer's memory, which can reduce the code and improve the performance.

Pointers

A pointer is a var that stores the memory address as its val. It points to a data type of the same type, and is created with the * operator.

Creating pointers

```
string food = "Pizza";   // A food variable of type string
string* ptr = &food;   // A pointer variable, with the name ptr, that stores the address of food
cout << food << "\n";   // Output the value of food (Pizza)
cout << &food << "\n";   // Output the memory address of food (0x6dfed4)
```

```
cout << ptr << "\n";   // Output the memory address of food with the pointer (0x6dfed4)
```

Dereference

```
string food = "Pizza";  // Variable declaration
string* ptr = &food;    // Pointer declaration
cout << ptr << "\n";    // Reference: Output the memory address of food with the pointer (0x6dfed4)
cout << *ptr << "\n";   // Dereference: Output the value of food with the pointer (Pizza)
```

* sign

When used in declaration (*string* ptr*), it creates a pointer var.

When not used in declaration, it act as a dereference operator.

Modify the pointer value: change the pointer's value, but it will also change the val of the original var. Pointer type store an address, so dereference a pointer type to access the original value.

Pass by reference

Useful when you need to change the value of the arguments. Pass in by using, for example, *int &x*

Functions

A function is a block of code which only runs when it is called. The data, or parameters, can be passed into a func.

Pre-defined func: main( ), used to execute code.

Function declaration and definition

Declaration: the return type, the name of the function, and parameters (if any).

Definition: the body of the function (code to be executed).

NOTICE: if user-defined func is declared after the main( ) function, an error will occur, then separate the declaration and the definition of the func for code optimization is useful. So, declaration goes before the main function, such as void myFunc( ); before the main function.

Parameters and Arguments

Information passed to funcs as a parameter. It act as var inside the function. When a parameter is passed to the function, it is called an argument.

Default parameters (optional parameter)

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}
int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();   //will output default val
  myFunction("USA");
  return 0;
}   // output: Sweden, India, Norway, USA
```

Return keyword: void should return nothing.

Function overloading

With function overloading, multiple functions can have the same name with different parameters.

```
int plusFunc(int x, int y) {
  return x + y;
}
double plusFunc(double x, double y) {
  return x + y;
}
```

```
int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
   cout << "Int: " << myNum1 << "\n";
   cout << "Double: " << myNum2;
  return 0;
}
```

To have multiple functions has same names but different return type and parameter type. Multiple functions can have the same name as long as the number and/or type of parameters are different.

Operator Overloading

The ability to provide the operators with a special meaning for a data type. It is a compile-time polymorphism, an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

@the difference between operator functions and normal functions?

They are the same. Only differences are, the name of an operator function is always the operator keyword followed by the symbol of the operator and operator functions are called when the corresponding operator is used.

Operators that cannot be overloaded: sizeof, typeid, scope resolution (::), class member access operators (. (dot), .* (pointer to member operator)), ternary or conditional

Important points about operator overloading

At least one of the operands must be a user-defined class object.

Compiler automatically creates a default assignment operator with every class.

Conversion operators can be used to convert one type to another type.

OOP (object-oriented programming)

Procedural programming is about writing procedures or functions that perform operations on the data.

Object-oriented programming is about creating objects that contain both data and functions.

Advantages of OOP:

OOP is faster and easier to execute

OOP provides a clear structure for the programs

OOP helps to keep the C++ code DRY (dont repeat yourself), makes the code easier to maintain, modify and debug.

OOP makes it possible to create full reusable applications with less code and shorter development time.

DRY principle: about reducing the repetition of code, you should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Class and objects: a class is a template for objects, and an object is an instance of a class. When the individual objects are created, they inherit all the var and func from the class.

Class/objects

For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".

Create a class: use the class keyword

public keyword: an access specifier, which specifies that members (attributes and methods) of the class are accessble from outside the class.

Attributes: var are declared within a class.

Object: to create an object of Myclass, specify the class name, followed by the object name. To access the class attributes, use the dot on the object. Objects can be created multiple times.

Class Methods

Methods are functions that belongs to the class.

define a function outside the class definition: need to declare it inside the class and then define it outside of the class. We need to specify the name of the class, followed the scope resolution :: operator, followed by the name of the function.

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};
// Method/function definition outside the class
void MyClass::myMethod() {
   cout << "Hello World!";
}
int main() {
   MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Constructors

A special method that is automatically called when an object of a class is created.

Create a constructor: use the same name as the class, followed by parentheses ( )

NOTICE: the constructor has the same name as the class, always public, does not have any return value.

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};
int main() {
   MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

Constructor parameters: constructors take parameters can be useful for setting initial values for attributes.

```
class Car {        // The class
  public:          // Access specifier
    string brand;  // Attribute
     string model;  // Attribute
    int year;      // Attribute
     Car(string x, string y, int z) { // Constructor with parameters
       brand = x;
       model = y;
      year = z;
     }
};
int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
```

```
  Car carObj2("Ford", "Mustang", 1969);
 // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
 return 0;
}
```

Defined outside the class: declare the constructor inside the class; then define it outside of the class: className :: constructorName( ){ };

Copy constructor

A member function that initializes an object using another object of the same class; a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

Characteristics

It is used to initialize the members of a newly created object by copying the members of an already existing object.

It takes a reference to an object of the same class as an argument.

The process of initializing members of an object through a copy constructor is copy initialization.

Member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

It can be defined explicitly by the programmer.

Types

Default copy constructor: copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

User defined copy constructor: generally needed when an object owns pointers or non-shareable references, such as to a file , in which case a destructor and an assignment operator should also be written.

When is the copy constructor called:

An object of the class is returned by value.

An object of the class is passed to a function by value as an argument.

An object is constructed based on another object of the same class.

The compiler generates a temporary object.

RVO (return value optimization): a technique that gives the compiler some additional power to terminate the temporary object created which results in changing the observable behavior of the final program.

When is a user-defined copy constructor needed: if an object has pointers or any runtime allocation of the resource like a file handle, a network connection, etc.

The default constructor does only shallow copy; deep copy is possible only with a user-defined copy constructor.

Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to the copy constructor would be made to call the copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

Can we make the copy constructor private?

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like the above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

Copy constructor vs Assignment Operator

The main difference between Copy Constructor and Assignment Operator is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

Destructors

An instance member function which is invoked automatically whenever an object is going to be destroyed. It iss the last func that is going to be called before an object is destroyed.

Characteristics:

It destroys the class objects created by constructor.

It has the same name as their class name preceded by a tilde (~) symbol.

Not possible to define more than one destrctor.

It is the only one way to destroy the object create by constrctor.

Not requires argument and return val.

It release memory spaces occupied by the objects created by the constructor.

Objects are destroyed in the reverse of an object creation.

NOTICE: if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

(1) the function ends

(2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

How are destructors different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don't take any argument and don't return anything

Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function. See virtual destructor for more details.

Default Constructors

If no constructors are explicitly declared in the class, a default constructor is provided automatically by the compiler.

Can a default constructor contain a default argument?

Yes, a constructor can contain default argument with default values for an object.

A constructor without any arguments or with the default value for every argument is said to be the **Default constructor**.

What is the significance of the default constructor?

They are used to create objects, which do not have any specific initial value.

Is a default constructor automatically provided?

Private Destructor

What is the use of private destructor?

Whenever we want to control the destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.

Access Specifiers

Public: vars can be accessed and modified from outside the code.

Private: members cannot be accessed or viewed from outside the class, but can be accessed in inherited classes.

NOTICE: all members of a class are private if you dont specify an access specifier.

Encapsulation

Meaning: to make sure that "sensitive" data is hidden from users. To achieve this, we must declare class var/attributes as private. Provide public get and set methods in order to let others to read or modify the value of a private member.

@Why Encapsulation?

Since a good pratice to declare your class attributes as private, encapsulation ensures better control of your data, since you can change one part of the code without affecting other parts. It also increased security of data.

Files

fstream lib allows to work with files. Include both the standard <iostream> and the <fstream> header file.

Three classes included in the fstream lib, which are used to create, write or read files:

Class	Description
ofstream	Creates and writes to files
ifstream	Reads from files
fstream	A combination of ofstream and ifstream: creates, reads, and writes to files

Create and write to a file

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  // Create and open a text file
   ofstream MyFile("filename.txt");
  // Write to the file
   MyFile << "Files can be tricky, but it is fun enough!";
  // Close the file
   MyFile.close();
}
```

@Why close the file?

It can clean up unnecessary memory space.

Read a File:

```
// Create a text string, which is used to output the text file
string myText;
// Read from the text file
ifstream MyReadFile("filename.txt");
// Use a while loop together with the getline() function to read the file line by line
while (getline (MyReadFile, myText)) {
  // Output the text from the file
  cout << myText;
}
// Close the file
MyReadFile.close();
```

Dynamic memory

When the memory needed depends on user input.

Operators new and new[ ]: dynamic memory is allocated using operator new. It returns a pointer to the begining of the new block of memory allocated.

```
Pointer = mew type    //used to allocate memory to contain one single element of type type
Pointer = new type [num_of_elements]    //used to allocate a block (an array) of elements of type type, where num_of_elements is an integer value representing the amount of these.
Int* foo;
Foo = new int [5];
```

The system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo. So, foo now points to a valid block of memory with space for five elements of type int.

Check if the allocation was successful:

Handling exceptions: the method used by default by new, is the one used in a declaration.

```
Foo = new int [5];    //if allocation fails, an exception is thrown
```

Nothrow method: when a memory allocation fails, the pointer returned by new is a null pointer, and the program continues its execution normally.

```
Foo = new(nothrow) int [5];
```

If the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer.

```
int* foo;
Foo = new(nothrow) int [5];
If (foo == nullptr) {
      //error assigning memory, take measures.
}
```

Operators delete and delete[ ]: once the memory is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.

```
delete pointer;    //releases the memory of a single element allocated using new.
delete[ ] pointer;    //release the memory allcated for arrays of elements using new and a size in brackets.
```

The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a null pointer. (delete produces no effect to the null pointer)

ECE244 Final

OBJECT PASSING

1. const: a qualifier -> value cannot be changed
2. constexpr: a qualifier -> value is constant expression; can be evaluated at compile time.
    - Implies const, but const can only be evaluated during run-time

```
constexpr int x = 42;
int y = 1;
x = 0; // ERROR: x is const
const int& x1 = x; // OK
const int* p1 = &x; // OK
int& x2 = x; // ERROR: x const, x2 not const
int* p2 = &x; // ERROR: x const, *p2 not const
int a1[x]; // OK: x is constexpr
int a2[y]; // ERROR: y is not constexpr
```

3. Lifetime: the time between an object's creation and its destruction, aka the time period the var/obj has valid memory. (aka. Allocation method/storage duration)
    - x global scope and lifetime, j function scope and lifetime, i block scope and lifetime.

```
int x;

void wasteTime()
{
    int j = 10000;
    while (j > 0) {
        --j;
    }
    for (int i = 0; i < 10000; ++i) {
    }
}
```

4. Parameter passing
    - *Pass by value:* function given copy to object from caller

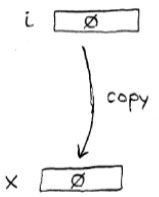
```
void increment0(int x) {
    ++x; // Increment x by one.
}

void increment(int& x) {
    ++x; // Increment x by one.
}

void func() {
    int i = 0;
    increment0(i); // i is passed by value
    // i still equals 0 (i was not incremented)
    increment(i); // i is passed by reference
    // i equals 1 (i was incremented)
}
```

```
void func() {
    int i = 0;
    increment0(i); // i unchanged
}

void increment0(int x) {
    ++x;
}
```



    - *Pass by reference:* function given reference to object from caller; pass ref type (pass the var, but parameter is type address)

```
void func() {
    int i = 0;
    increment(i); // i is incremented
}

void increment(int& x) {
    ++x;
}
```


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

x = 3;
y = 5;
swap(x, y);
```

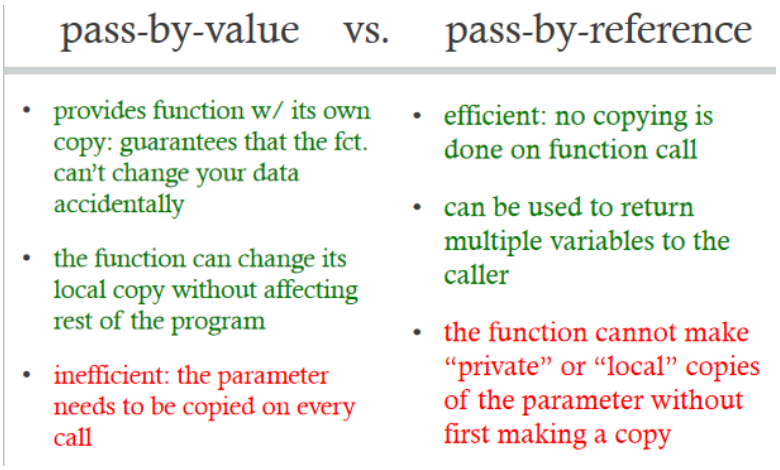
- Benefits: only a reference (sort like a ptr) is copied, the data itself is not copied; modification made in the callee remains after it returns.

- Example:

```
void increment(int& x)
  // x is passed by reference
{
    ++x;
}

double square(double x)
  // x is passed by value
{
    return x * x;
}
```

- *Compare:* object is expensive to copy, so pass by reference when needs to change value of object in caller.



5. Memory allocation
   - *new* - allocate memory; array version new[]
   - *delete* - deallocate memory allocated with new statement; array version delete[]
   - *Forms of allocation*
     - Single obj. (e.g. non-array case)
     - Array of objects
   - Match non-array and array versions of new and delete. If not, may compile fine, but crash.

```
char* buffer = new char[64]; // allocate
                             // array of 64 chars
delete [] buffer; // deallocate array
double* x = new double; // allocate single double
delete x; // deallocate single object
```

6. Propagating values: propagate the value of src to the destination, src and destination of same type obj.
   - *Copying:* without modifying the src obj.
   - *Moving:* permitted to modify the src obj.
     - More or same efficient as copying.
   - Some types that copying does not make sense but moving does
     - std::ostream
     - std::istream
7. Constructors: to initialize new obj to some known state; prevent object from accidentally being used before it is initialized.

- *Feature:*
  - a member function that called automatically when obj is created
  - Have the same name as class
  - No return type
  - Cannot be called directly
  - Can be overloaded
  - Must be called before other functions
- *Default constructor:* constructor that can be called with no arguments.
  - If no constructors specified, default constructor automatically provided that calls default constructor for each data member of class type.
- *Copy constructor:* constructor taking value reference to T as first parameter that can be called with one argument.
  - Usage: create obj by copying from existing obj; default one only perform shallow copy.
  - When invoke: istantiating one object from another of the same type; when an object is passed into a function by value; when an object is returned from a function by value.
  - Form: T(const T& other); const since non-const references cannot bind to temporary objects.
  - Feature: if no copy constructor, move constructor or move assignment operator specified, copy constructor is automatically provided that copies each data member.
  - Using copy constructor for class, and bitwise copy for built-in type.

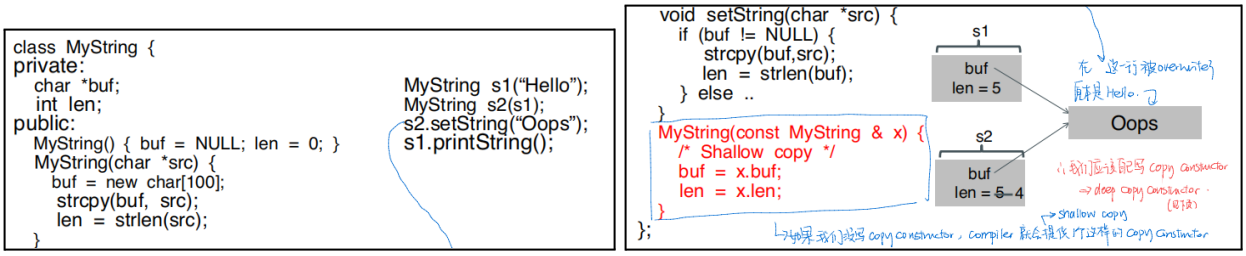
```
class Vector { // Two-dimensional vector class.
public:
    // ... (e.g., default constructor)
    Vector(const Vector& v) { // Copy constructor.
        x_ = v.x_; y_ = v.y_;
    }
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};

Vector v;
Vector w(v); // calls Vector(const Vector&)
Vector u = v; // calls Vector(const Vector&)
```

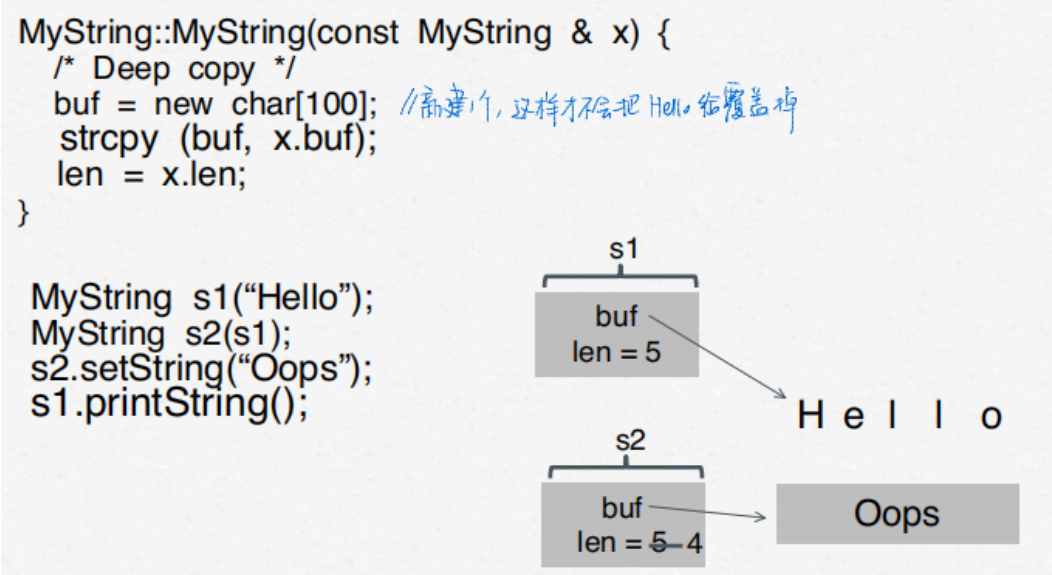
8. Return an object by value
   - Another copy of the object could be created via copy constructor.
   - *In g++*: this copy is omitted as an optimization by default.
     - To disable it: -fno-elide-constructors
   - Create 7 obj in this example.

```
                                  ④        ⑤
Complex operator+ (Complex x, Complex y)
{
  Complex d(x.getReal() + y.getReal(),   ⑥
            x.getImag() + y.getImag());
  return d; ⑦
}
Complex a(3.2, 1.0); ①
Complex b(4.0, 0.5); ②
Complex c; ③
c = a + b;
```

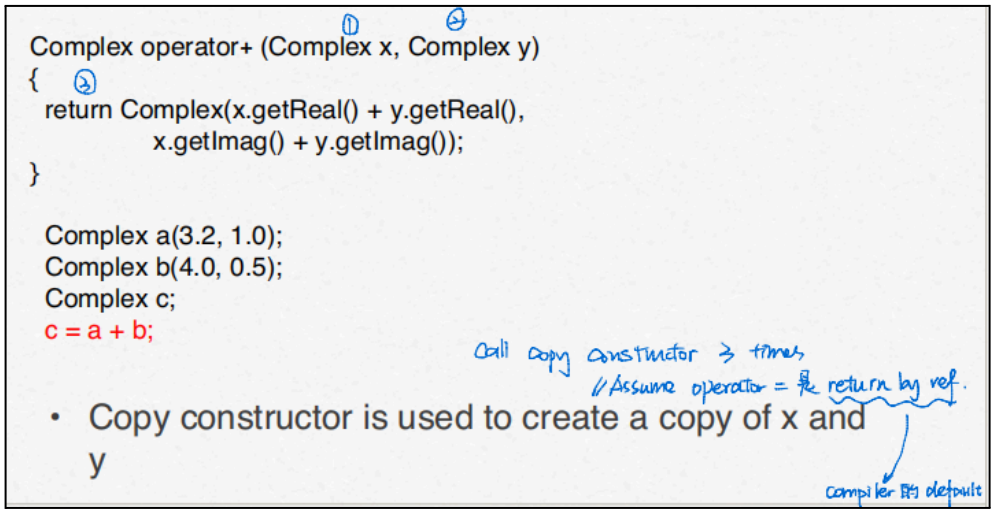
9. Shallow copy versus deep copy
   - *Shallow copy* (by default); overwrite the original value



- *Deep copy* (need to be defined by user)
  - The original val will not be overwrote.



10. Passing obj into func by value (operator overloading type)
    - Since non-const references cannot bind to temporary obj., the copy constructor need to pass in const.
    - Copy constructor is invoked when declare and initializr an obj at the same time, but not invoked when declare and initialize separately.
    - *If some data members are pointers to other data:*
      - Provide own default constructor (to allocate the data)
      - Own copy constructor (deep copy)
      - Own assignment operator (deep assignment)
      - Destructor (explicitly delete dynamically allocated data)



LINKEDLIST

Lec16

1. Linked list: A collection of nodes that are linked to one another via pointers.
   - Usage: store data in some order.
   - Feature: last node point to NULL
2. Code class - define a node, hence a list

```
class Node {
  private:
    int data;
    Node *next;
  public:
    Node() { data = 0; next = NULL; }
    Node(int d) { data = d; next = NULL; }
    Node(int d, Node *n) { data = d; next = n; }
    int getData() const { return data; }
    void setData (int _d) { data = _d; }
    Node *getNext() const { return next; }
    void setNext(Node *_n) { next = _n; }
}
```

```
class List {
  private:
    Node *head;
  public:
    List();
    ~List();
    void insertValue (int);
    bool valueExists (int);
    bool deleteValue (int);
    List (const List & original);
    List & operator= (const List & original);
    .. ..
};
```

3. Features of the list
   - *Keep the list sorted.*
   - *Keep track of "head" of the list.*
   - *Support basic operations.*
     - Inserting (keep the list sorted)
     - Deleting
     - Searching
     - Copying one list from another list
   - *Have a destructor* that properly deletes the list.
4. Class list - member functions
   - *Searching*
     - When the list is empty.
     - When the searched value is not in the list.
     - Special cases: sorted nums, so if num > (or <) searched value, return false.

```
bool List::valueExists(int _data)
{
  for (Node *cur = head; cur != NULL; cur = cur->getNext())
  {
    if (cur->getData() == _data)
    {
      return true;
    }
    if (cur->getData() > _data)
    {
      return false;
    }
  }
  return false;
}
```

5. Class list - member functions
   - *Insert at the head:* change the head to the new inserted one; use temp to access.
   - *Insert at the end:* the new node after inserted node points to NULL; use temp to access.
   - *Insert in middle:* ensure the "next" pointers are properly updated; use temp to access.

- Do not change the head ptr, use temp to track.

```
void List::insertValue (int _data)
{
  Node *n = new Node(_data);
  Node *p = head;
  Node *prev = NULL;
  while (p != NULL && p->getData() < _data) {
    prev = p;
    p = p->getNext();
  }
  n->setNext(p);
  if (prev == NULL) // head of the list!
    head = n;
  else
    prev->setNext(n);
}
```

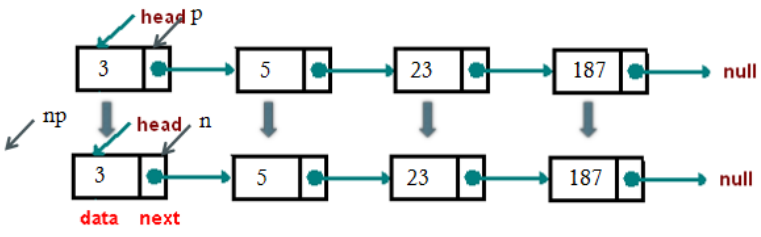
6. Class list - deleting, and example for destructor
   - True if data is found, return false if data is not found.
   - When list is empty.
   - When data is not in list.
   - If data as the first node/end node/middle.

```
bool List::deleteValue(int _data) {
  Node *p = head, *prev = NULL;
  while (p != NULL && p->getData() != _data) {
    prev = p;
    p = p->getNext();
  }
  if (p == NULL) // _data is not in the list
    return false;
  /* Remove the node pointed by p from the list. */
  if (prev == NULL) // p is the head of the list!
    head = p->getNext();
  else
    prev->setNext(p->getNext());
  delete p;
  return true;
}
```

```
List::~List() {
  Node *p;
  while (head != NULL) {
    p = head;
    head = p->getNext();
    delete p;
  }
}
```

7. Class list - copy constructor
   - *Reason:* C++ does shallow copy by default.
   - *Copy constructor to do a deep copy.*

```
List::List (const List & original) {
  Node *p = original.head;
  Node *np = NULL;
  head = NULL;
  while (p != NULL) {
    Node *n = new Node (p->getData());
    if (np == NULL) head = n;
    else np->setNext(n);
    p = p->getNext();
    np = n;
   }
}
```

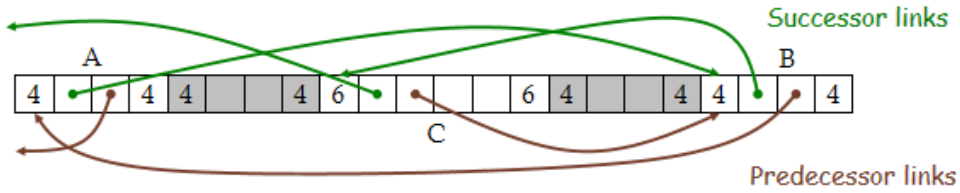


8. Class list - operator=
   - Similar to copy constructor, but need to deal with cases when copy(lhs) is not empty, and the original(rhs) is the same list as copy(lhs).

- Lhs = rhs; copy = original

```
List & List::operator=(const List & original) {
  if (&original == this)
    return (*this);
  if (head != NULL)
  { // empty the list as in destructor
    .. ..
  }
  // copy list as copy constructor
  return (*this);
}
```

9. Ordered vs. unordered
   - *Ordered list with non-exist data:* best case - one node deep, worst case - all, average - half of the list
10. Dynamic memory management example: free blocks are modeled as linked list



RECURSION

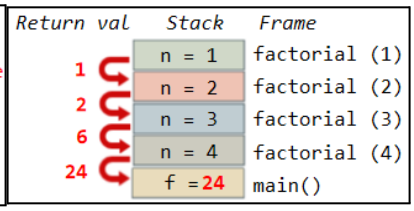
Lec17: Recursion

1. Recursion: an algorithm which breaks down problem to be solved into smaller, discrete steps that are solved using the same algorithm.
    - *Recursive func:* an implement of a recursive algorithm. E.g. factorial, n*f(n-1) if n>1, 1 if n=1
        - Makes calls to itself to perform each step.

```
void rprint (Node *p) {
    if(p == NULL) return;
    cout << p -> data;
    rprint(p -> next);
}
```

```
int factorial (int n) {
  if (n==1) return 1; // Basis
  return (n*factorial(n-1)); // Recursive
}

int main() {
  int f = factorial(4);
}
```



    - *Recursion on a linked list* - print the list

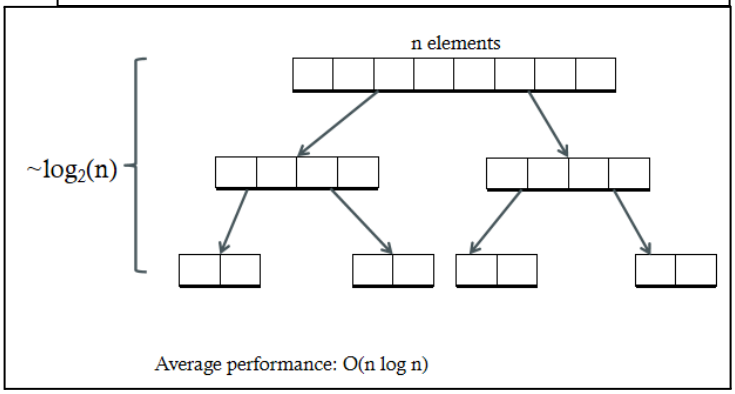
Lec18: quick sort

1. Sorting: given a list, put elements in a certain order.
2. Bubble sort: 从第一个开始，两两比较后交换，一直到最后一个。
    - Complexity: n + n-1 + n-2 … + 1 = n(n-1)/2 = O(n^2)
    - Slow
3. Quicksort: pick a pivot val from an array; move ever val < pivot to the left, >= pivot to the right; Quicksort sub-array to the left of pivot; quicksort sub-array to the right of pivot.
    - fastest general sorting algorithm
    - qsort in stdlib.h
    - *implementation:*
        - selectAndShuffle: Pick the first element as pivot; move elements > pivot to its right; < pivot to its left; return the new index of pivot.

```
void qsort(int *a, int begin, int end){
    int pivotIndex = selectAndShuffle(a, begin, end);
    if(pivotIndex - 1 > begin)
        qsort(a, begin, pivotIndex - 1);
    if(pivotIndex + 1 < end)
        qsort(a, pivotIndex + 1, end);
    return;
}
```

```
int selectAndShuffle(int *a, int begin, int end){
    int pivotValue = a[begin];
    int current = begin;
    for(int i = begin + 1; i <= end; i++)
        if(a[i] < pivotValue){
            current++;
            swap(a[current], a[i]);
        }
    swap(a[begin], a[current]);
    return current;
}
```

At the end of each iteration	i	current
6 3 9 8 5 2 7 1 4	-	0
6 3 9 8 5 2 7 1 4	1	1
6 3 9 8 5 2 7 1 4	2	1
6 3 9 8 5 2 7 1 4	3	1
6 3 9 8 5 2 7 1 4	4	2
6 3 5 8 9 2 7 1 4	5	3
6 3 5 2 9 8 7 1 4	6	3
6 3 5 2 9 8 7 1 4	7	4
6 3 5 2 1 8 7 9 4	8	5
6 3 5 2 1 4 7 9 8	9	5
4 3 5 2 1 6 7 9 8	-	5



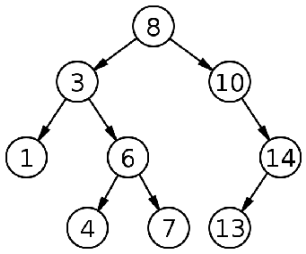
    - *Pivot point:* always the median of three.
        - median(a[begin], a[middle], a[end])
    - *Other methods*: merge sort, parallelize qsort.

Different methods of sorting and function implementations

BINARY SEARCH TREE

Lec19

1. Tree: a data structure made up of nodes and edges.
   - features:
     - No cycle or loop.
     - Only one root.
     - Each node can have multiple children, but only one parent.
   - usage:
     - Function call tree: to find where this function comes from; The function you are interested in is placed at tree root. Then, a search is performed through entire project to find all functions that call it.
       ~a useful tool for refactoring and visualizing dependencies in the code.
     - Hierarchical debugging.
     - Directory and files, assume no links.
2. Binary tree: each node has at most 2 children.
   - Binary search tree: a binary tree that for ever node,
     - The nodes in the left subtree have vals <= val of node.
     - The nodes in the right subtree have vals > val of node.
     - Min val: leftmost node without a left child.
     - Max val: right most node without a right child.
3. Tree traversal: visit every element of a tree.
   - methods:
     - Pre-order: node, left subtree, right subtree.
       8, 3, 1, 6, 4, 7, 10, 14, 13
     - Post-order: left subtree, right subtree, node.
       1, 4, 7, 6, 3, 13, 14, 10, 8
     - In-order: left subtree, node, right subtree.
       1, 3, 4, 6, 7, 8, 10, 13, 14



4. Build a binary search tree - build the skeleton

```
class Tree {
    private:
        TreeNode *root;
    public:
        Tree();
        void insert (int v);
        void print_inorder();
        bool valueExists(int v);
        void del(int v);
};
```

```
class TreeNode {
    private:
        int value;
        TreeNode *left;
        TreeNode *right;
    public:
        TreeNode();
        TreeNode(int v);
        void insert(int v);
        void print_inorder();
        int minimum();
        bool valueExists(int);
        void del(int v, TreeNode *& pp);
};
```

5. Build a binary search tree - member funcs
   In Tree class

```
void Tree::del(int v){
    if(root == NULL){
        return;
    }
    root -> del(v, root);
    return;
}
```

```
void Tree::print_inorder(){
    if(root != NULL){
        root -> print_inorder();
    }
    cout << endl;
}

bool Tree::valueExists(int v){
    return root -> valueExists(v);
}
```

```
Tree::Tree() {
        root = NULL;
}

void Tree::insert(int v) {
    if(root == NULL){
        root = new TreeNode(v);
    }else{
        root -> insert(v);
    }
    return;
}
```

6. Build a binary search tree - constructors
   - 三个变量：value, left, right; value为-1, right=left=NULL

```
TreeNode::TreeNode(){ //default construc
    value = -1;
    left = right = NULL;
}

TreeNode::TreeNode(int v){
    value = v;
    left = right = NULL;
}
```

7. Build a binary search tree - Insert
   As shown in the right pic.
   - If exist, return.
   - 小于当前位置，向左移，
     左侧为空，插入；
     不为空，从左侧位置再循环一次。
   - 大于当前位置，向右移，右侧为空，插入；不为空，右侧位置再循环一次。

```
void TreeNode::insert (int v){
    if(value == v)
        return;
    if(v < value){
        if(left == NULL) //insert to the left
            left = new TreeNode(v);
        else
            left -> insert(v);
    }else{
        if(right == NULL) //insert to the right
            right = new TreeNode(v);
        else
            right -> insert(v); //recursion
    }
    return;
}
```

8. Build a binary search tree - Traversal (in-order)

```
void TreeNode::print_inorder(){
    if(left != NULL)
        left -> print_inorder();
    cout << value <<" ";
    if(right != NULL)
        right -> print_inorder();
}
```

   - 左侧有数值，print左侧所有；
   - 右侧有数值，print右侧所有。

9. Build a binary search tree - find a minimum

```
int TreeNode::minimum(){
    if(left == NULL)
        return value;
    return left -> minimum();
}
```

   - 左侧无数，则返回当前数值为最小值；
   - 左侧有数，则左移一位，再次call本func，直到traverse到了最左侧。

10. Build a binary search tree - delete node
    Find the special feature of that node.

```
void TreeNode::del(int v, TreeNode*& pp) {
    if(v < value){
        if(left != NULL)
            left -> del(v, left);
        return;
    }
    if(v > value){
        if(right != NULL)
            right -> del(v, right);
        return;
    }
    if((left == NULL) && (right == NULL)){
        pp = NULL;
        delete this;
    }else if((left == NULL) && (right != NULL)){
        pp = right;
        right = NULL;
        delete this;
    }else if((left != NULL) && (right == NULL)){
        pp = left;
        left = NULL;
        delete this;
    }else{
        int m = right -> minimum();
        value = m;
        right -> del(m, right);
    }
    return;
}
```

   - 小于向左移动，大于向右移动。
   - 等于时，左右都无数值，store存null，删除左右。
   - 等于时，单左为空，store右数值，赋值右为空，删除右。
   - 右侧同理。
   - 当前位置左侧右侧都有数值，m存储右位的最小值，value存m，删除m；value放在删除的位置。

INHERITANCE & POLYMORPHISM

Lec20-21: Inheritance basic

1. **Usage:** inherit attributes and methods from one class to another.
   - After being declared, the derived class can be used like any other class.
2. **Definition**: mechanism => allow a class B (superclass) to inherit the content of class A (subclass)
3. **Types**: base class and derived class <= share properties, while the derived class has its own properties.
   - Derived class: inherits from other class <= larger than the base class.
     - Receives all of the member variables and functions of the base.
     - Can directly use the members of the base class
     - Can be extended/customized/overridden as needed
     - Properties
       ~inherits all member variables from base class
       ~can access protected members of base class, cannot access private members of base class
       ~can add member vars and funcs
       ~cannot remove member vars and funcs
4. **Benefit**: can reuse code, or features, from other classes.
5. **Important properties**
   - Data protection: derived classes cannot access private members of base class, although they are also inherited (will have memory allocated for it in derived class, though cannot be accessed).
   - Protected members: a member variable/function accessible to derived class but inaccessible to outside.
     - *Access specifier*: protected

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

   - Overriding (redefining) vs overloading
     - *Overloading*: same function name, different parameter list
     - *Overriding*: redefine a function in derived class
       ~member var redefine with the same name in derived class <= two copies of it, one in base obj, and one in derived class obj.
6. **Syntax**

```
class DerivedClass : public BaseClass
{
  /* Only list the inherited members if you wish
  to change them, plus new members you're adding. */
}

Example:
class OrderedList : public List {
  void insert (int v); // redefine insert()
  // value, next, delete(), print()… are all inherited,
  // and you can use them as if they were defined in
  // OrderedList
}
```

7. Mixing types by hierarchy: base obj can be derived obj, derived obj cannot be base obj.

```
void g (Manager mm, Employee ee)
{
  Employee *pe = &mm; // OK
  Manager *pm = &ee; // Not OK!
  ..
}
```

8. Inheritance vs membership: membership => only belongs to base class(独属), inheritance => belongs to everyone(众属)
9. Base/derived class in containers
   - Stored base obj into container cannot be converted back to derived class obj, but can convert ptrs

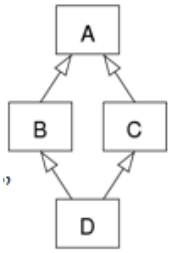
```
vector<Employee> vec;
vec.push_back(manager);
vec[0].print_manager_level(); // ERROR
```

```
vector<Employee*> vec;
vec.push_back(&manager);
((Manager*)vec[0])->print_manager_level(); // OK
```

10. Notice: avoid multiple inheritance (e.g. diamond of dread)
    - Code won't compile
      - Two copies of grandpa's members are created when creating a child obj
      - Compiler won't know which "getName()" should be invoked

```
class Grandpa {
private:
  char* name;
public:
  char* getName () { return name; }
};
class Parent1 : public Grandpa { .. };
class Parent2 : public Grandpa { .. };
class Child : public Parent1, public Parent2 { .. };
```

```
int main () {
  Child c;
  c.getName();
  .. ..
}
```



11. Accessing parent members (by member func)

```
class Base {
    private:
        int a;
    public:
    void set(int _a) {
        a = _a;
    }
    void print() {
        cout << "Base.a = " << a << endl;
    }
};
```

```
class Derived : public Base {
    private:
    int b;
    public:
    void set(int _a, int _b) {
        Base::set(_a);
        b = _b;
    }
    void print() {
        Base::print();
        cout << "Derived.b = " << b << endl;
    }
};
```

```
int main(){
    Derived b;
    b.set(1, 2);
    b.print();
    return 0;
}
```

    - In member func of Derived class, called member func in Base class, in order to access parameter a and set its val to _a.
    - If not use inheritance, the Derived class cannot set a since it doesn't include such a private var.
12. Accessing parent members (by protected method)

```cpp
class Base {
    protected:
        int a;
};
```

```cpp
class Derived : public Base {
    private:
        int a;
    public: void set(int _base, int _derived) {
        Base::a = _base;
        a = _derived;
    }
    void print() {
        cout << "Base.a = " << Base::a << endl;
    }
};
```

```cpp
int main(){
    Derived b;
    b.set(1, 2);
    b.print();
    return 0;
}
```

- Output: Derived a is 2, Base a is 1
- protected: keyword that means members cannot access outside the class but can be accessed by inherited class. (private cannot be accessed by inheritance and outside)

13. Access predecessor's members

```cpp
class Grandpa {
    protected:
        int a;
    public:
        void print(){
            cout << "Grandpa.a = " << a << endl; }
};
```

```cpp
class Dad : public Grandpa {
    protected:
        int a;
    public:
        void print(){
            cout << "Dad.a = " << endl;
        }
};
```

```cpp
class Son : public Dad {
    private:
        int a;
    public:
        void set(int _grandpa, int _dad, int _son){
            Grandpa::a = _grandpa;
            Dad::a = _dad;
            Son::a = _son;
        }
        void print(){
            Grandpa::print();
            Dad::print();
            cout << "Son.a = " << a;
        }
};
```

The derived classes can use the protected members in parent classes.
Since dad is inherited from grandpa, the son is inherited from dad, so the son is also inherited from grandpa.

Lec22-23: Inheritance constructors & polymorphism

1. Constructors in Inheritance:
   - Constructors are not inherited
   - Initializing a derived class object => default constructor is called if not called other constructors.
   - Explicitly call non-default base constructor (notice the parameters):

     ```
     Derived::Derived(..) : Base(..) { .. }
     ```

   - Base constructor always called first.

```
class A {
    private:
        int a;
        int b;
    public:
        A(int _a, int _b) { //constructor for A
            a = _a;
            b = _b;
        }
        void print() {
            cout << "A.a = " << a << ", A.b = " << b << endl;
        }
};
```

```
class B : public A {
    private:
        int a;
        int b;
    public:
        B() : A(1, 1) { //set val for class A
            a = b = 2; //set val of B itself
        }
        void print() {
            A::print();
            cout << "B.a = " << a << ", B.b = " << b << endl;
        }
};
```

2. Initialize member vars in constructor with similar syntax

```
class Employee {
private:
  string name;
  int sin;
public:
  //Employee() : name (“”), sin(0) { }; //default constructor
  Employee(string n, int s) : name(n), sin(s) { /* empty */ }
  //the way to set name = n, sin = s
};
...
Manager::Manager(string n, int sin, int _level)
  : Employee(n, sin), level(_level) { /* empty */ }
  //the way to set level = _level
```

A new way to initialize the parameter by inheritance

3. Order of constructor invocations: bottom-up; child(manager) > parent(employee) > grandparent(staff)[base]
   *This is actually the order of destructor
   *order of constructor should be the inversed order!
4. Types of constructor
   Constructors and destructor are not inherited, but they can be invoked.
   E.g. if the derived class has an obj of itself, then it will invoke the default constructor in Base class; cannot call Base's user-defined constructors in derived class.
   - Default constructor: occurs when no user-declared constructor; no need to define.
   - User declared constructor: func that no return type, with parameters.
   - Destructor: called automatically; needs to define.
   - A class contains member vars that point to dynamic memory: should manage memory and write,
     - operator =

```
Base & Base::operator= (const Base & rhs) {
 // Deep copy
}

Derived & Derived::operator= (const Derived & rhs)
{
  Base::operator=(rhs);
  // Deep copy any extra Derived variables
  return (*this);
}
```

     ~use operator overflow to realize deep copy
     - copy constructor: for derived class, call base class copy func; deep copy extra member vars contained in the derived class.

```
Base::Base (const Base & original) {
 // Deep copy original into *this
}

Derived::Derived (const Derived & original) : Base(original)
{
  // Deep copy extra Derived vars
}
```

     ~传入一个Base类型的obj，copy到ptr this，而后在derived func中inheritance在Base里的constructor。
     ~如若不写，则会自动call在Base里的default constructor，报错
5. Virtual functions
   - Situation: Func declared as virtual in Base, redefined in derived class.
   - A call made via a pointer, which is declared as Base* type points to an obj of derived class, cause the func defined in Base invoked.

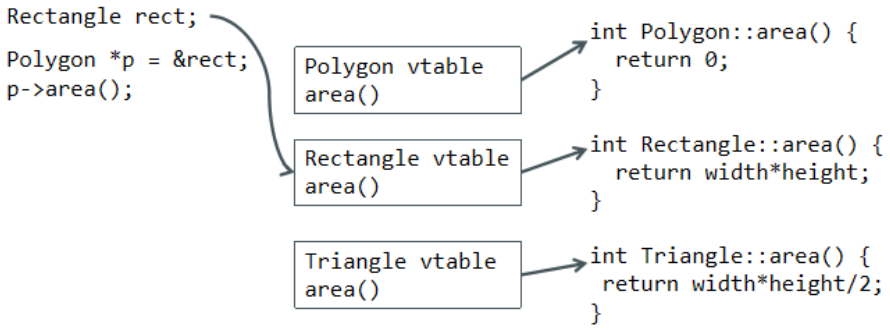
```
class Polygon {
 protected:
   int width, height;
 public:
   void set_values (int a, int b) { width=a; height=b; }
   virtual int area() { return 0; }
 };

class Rectangle: public Polygon {
 public:
   int area() { return width*height; }
};

class Triangle: public Polygon {
 public:
   int area() { return width*height/2; }
 };
```

```
Rectangle rect;
Triangle trgl;
Polygon * ppoly1 = &rect;
Polygon * ppoly2 = &trgl;
ppoly1->set_values(4,5);
ppoly2->set_values(4,5);
cout<<ppoly1->area(); // 20
cout<<ppoly2->area(); // 10
```

6. Polymorphism: the ability of a C++ function or object to perform in different ways, depending on how the function or object is used.
    - Polymorphic class: A class that declares or inherits a virtual func.
7. Virtual functions: in Base class, it is a placeholder declaration of the func; it is a run-time polymorphism.
    - Cannot declare a member var as virtual
    - *Late binding:* feature of virtual func => the decision of which inherited func to call is made at runtime when func is invoked, depending on the type of the target obj.
    - *A call to a virtual func is not a simple jump at run-time* => figure out the call target.
        - Performance penalty occurs, so more expensive than a normal call.
        - Virtual method table (vtable) to implement virtual function: a table that maps each virtual func to the most-derived func target; each obj has a ptr to the vtable of its type.



8. Pure virtual func: A virtual function just provides the template, and the derived class implements the function.
9. Abstract class (interface): A class having pure virtual function.
    - cannot be used to create direct objects of its own.

`virtual void display() = 0;` `virtual void display() {}`

Virtual function	Pure virtual function
A virtual function is a member function in a base class that can be redefined in a derived class.	A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class.
The classes which are containing virtual functions are not abstract classes.	The classes which are containing pure virtual function are the abstract classes.
In case of a virtual function, definition of a function is provided in the base class.	In case of a pure virtual function, definition of a function is not provided in the base class.
The base class that contains a virtual function can be instantiated.	The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated.
If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation.	If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class.
All the derived classes may or may not redefine the virtual function.	All the derived classes must define the pure virtual function.

10. Overhead: the cost, usually counted in time or memory usage.
    - Virtual func call has overhead 6-13% than non-virtual func calls.
    - Class with virtual func prevent compilers from other optimizations, e.g.:
        - Function inlining: an enhancement feature that improves the execution time and speed of the program.
    - Just-In-Time compiler can use inline caching to reduce this overhead.
        - Inline caching: technique for runtime optimization.

COMPLEXITY & RECURSION

Lec26-27: complexity

1. Big-O notation:
   - Only consider highest order term when comparing algorithms since that dominates.
   - T(n) is on the order of O(f(n)) if T(n)'s highest order term is f(n), disregarding constants.
   - T(n) is in O(f(n)) if constant C, N, so that $T(n) \leq C \times f(n)$ for every $n > N$.
   - O(n^3) means three loops together (outer - outer - inner).
   - examples:
     - O(log(N)) => binary search (on sorted array)
     - O(1) => constant time
     - O(n) => linear time, n is the number of elements in array
     - O(n^x) => x is any const; means x loops grouped together
     - O(2^n) => recursive calculation of Fibonacci numbers

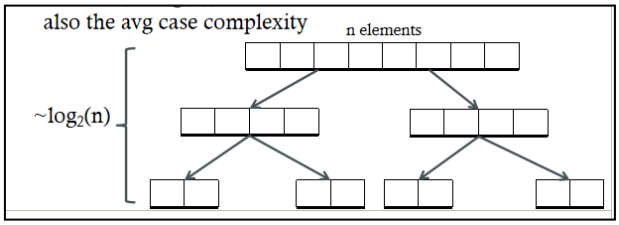
```
int fibonacci(int num)
{
    if (num <= 1) return num;
    return fibonacci(num - 2) + fibonacci(num - 1);
}
```

2. Analyze the complexity of recursive programs

```
int fact(int n) {
  if (n == 1) return 1;
  return (n * fact (n - 1));
}

 T(n) = 2 + T(n-1)
      = 2 + 2 + T(n-2)
      = 2 x 3 + T(n-3)
      = 2 x (n-1) + T(n - (n-1)) = 2 x (n-1) + T(1) = 2n - 2


 Complexity: O(n)
```



- QuickSort => T(n) = TselectAndShuffle(n) + T(k) + T(n - k - 1)

```
void qsort (int *a, int begin, int end){
    int pivotIndex = selectAndShuffle(a, begin, end);
    if(pivotIndex - 1 > begin)
        qsort(a, begin, pivotIndex - 1);
    if(pivotIndex + 1 < end)
        qsort(a, pivotIndex + 1, end);
    return;
}
```

QuickSort: best/avg. case

k = n/2, i.e., evenly split the array

$T(n) \approx C \times n + 2 \times T(n/2)$

$\approx C \times (n + n) + 4T(n/4))$

$\approx C \times (n + n + n) + 8T(n/8)$

$\approx O(n \times \log(n))$ } log(n)

   - Worse case:

$T(n) \approx T_{selectAndShuffle}(n) + T(k) + T(n - k - 1)$

$\approx C \times n + T(k) + T(n-k-1)$

Worst case: k = 0, i.e., pivotIndex == begin

$T(n) \approx C \times n + T(n-1)$

$\approx C \times (n + n-1) + T(n-2)$

..

$\approx C \times (n + n-1 + n-2 + \dots 1) = C \times n \times (n+1)/2 = O(n^2)$

- selectAndShuffle =>

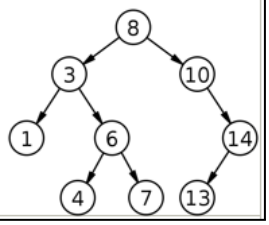
```
int selectAndShuffle (int*a, int begin, int end){
    int pivotValue = a[begin];
    int current = begin;
    for (int i = begin+1; i <= end; i++)
        if(a[i] < pivotValue){
            current++;
            swap(a[current], a[i]);
        }
    swap(a[begin], a[current]);
    return current;
}
```

- Search on binary search tree

- If the tree is balanced, e.g., every non-leaf node has two children
  - $T(n) = C + T(n/2) = 2C + T(n/4) = .. = C * \log_2(n)$

```
bool TreeNode::valueExists(int v) {
  if (v == value)
    return true;
  if (v < value) {
    if (left != NULL)
      return left->valueExists(v);
    return false;
  }
  if (right != NULL)
    return right->valueExists(v);
  return false;
}
```

O(log(N))



HASH TABLE

Lec26

1. Hash table: a data structure that maps keys to values; uses a hash func to compute indexes for a key.
   - Idea:
     - Stores data in a very large array; typically 两倍 of stored data#
     - Each key maps to a unique index
     - Has func h(k), a func to map each element into an array index.
   - Time complexity: O(1) constant access time.
2. Collision: a problem that two keys may map to the same array entry.
   - Hashing with chaining: each hash table entry contains a ptr to a linked list of keys that map to the same entry.
   - The worst case: O(n) complexity
3. Good hash function: avoid collision; the average length of each list is 1.
   - Use more spaces
   - Use smarter algorithms
     - Real-world hashing algorithms usually involve multiply a large prime number; e.g., h(k) = k * 31 % m
4. Algorithms before

	Unsorted Array	Sorted Array	Unsorted List	Sorted List	BST (min level tree)
Insert	O(1)	O(n)	O(1)	O(n)	O(log n)
Search	O(n)	O(log n)	O(n)	O(n)	O(log n)

5. Implementation

- components:
    - Good hash func to map keys to values
    - Insert, search, and delete operations
    - Data structure to account for collision of keys