Introduction to C++

Himanshu Singh

Github

Recap of C

Built-in/Basic Data types in C

- char
- int
- float
- double

Additionally, C89 define _Bool
void : it indicates no type
Derived data types

- Array
- Structure – struct and union
- Pointer
- Function
- String – C String are not a data type, but can me made to behave as such using <string.h>

Type Modifiers:

- short
- long
- signed
- unsigned

Literals refer to fixed values of built-in type. Literal can be of any of the basic data types

212	//(int) Decimal literal
0173	//(int) Octal literal
0b11001	//(int) Binary literal
0xF23	//(int) Hexadecimal literal
3.14	//(double) Floating-point literal
‘x’	//(char) Character literal
“Hello”	//(char*) String literal

An operator denotes specific operation.

- Arithmetic Operators + - * / % ++ --
- Relational Operators== != > < >= <=
- Logical Operators && || !
- Bitwise Operators | & ~ ^ << >>
- A1ssignment Operators = += -= *= ...
- Miscellaneous Operators . , sizeof & ?:

Precedence of all operator: (), [], ++, --, +(unary), -(unary), !, ~, *(pointer ref), &(address of), sizeof, *, /, %. +, -, <<, >>, ==, !=, *=, =, /=, &, |, &&, ||, ?:, =, +=, -=, *=, /=, <<=, >>=

Structure is collection of data items of different types. Data items are called members.

```
typedef struct _complex {
    double re;
    double im;
} Complex;
Complex c = {2.0, 3.5};
```

Union is special structure that allocates memory only for the largest data member and hold only one member at a time.

```
typdef union _packet {
    int     iData;
    double  dData;
    char    cData;
} Packet;
Packet p = {10};
```

Pointer Array Duality

```
int a[] = {1, 2, 3, 4, 5};
int *p;
p = a;
printf(“a[0] = %d\n”, *p);
printf(“a[1] = %d\n”, *++p);
printf(“a[3] = %d\n”, *(p+2));
```

Pointer to structure

```
typedef struct _complex {
    double re;
    double im;
} Complex;
Complex c = {2.3, 4.7};
complex *p = &c;
(*p).re = 6.3;
p->im = 3.1;
```

C++ Intro

Hello World

```
#include <iostream>
int main() {
    std::cout << “Hello World in C++”;
    std::cout << std::endl;
    return 0;
}
```

IO header is iostream.
Opeartor << to stream to console, is a binary operator

Add two numbers

```
#include<iostream>
int main() {
    int a, b;
    std::cout << “Input two numbers\n”;
    std::cin >> a >> b;
    int sum = a + b;
    std::cout << “Sum of ” << a << “ and ” << b << “ is: ” << sum << std::endl;
}
```

operator >> to stream from console, is a binary operator.

Square root of number

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    cout << “Input number : ” << endl;
    cin >> x;
    double sqrt_x = sqtr(x);
    cout << “Square root of ” << x << “ is: ” << sqrt_x << endl;
}
```

Math header is cmath (C standard Library in C++), sqrt function is from this library

In C++ Standard Library, all names are in std namespace (std::cout, std::cin). We use using namespace std to get rid of std:: for every standard library name.

C/C++ Header Conventions

	C Header	C++ Header
C Program	Use .h Example: #include <stdio.h> Names in global namespace	Not Applicable
C++ Program	Prefix c and no .h Example: #include <cstdio>	No .h Example: #include <iostream>

Dynamically managed array size

```
#include<iostream>
#include<vector>
using namespace std;
int main() {
    vector<int> arr;
    cout << “Enter the  number of elements ”;
    int count, sum = 0;
    cin >> count;
    arr.resize(count);
    for(int i = 0; i < count; i++) {
        arr[i] = i;
        sum += arr[i];
    }
    cout << "Array sum : " << sum << endl;
}
```

resize fixed vector size at run time. Whereas in C for dynamically managing the size of array, we need functions like malloc, calloc and free.

Stings in C and C++

In C, strings are in string.h library. They are array of char terminated by NULL. C-String is supported by functions in string.h in C standard library.

In C++, string is a type. With opeartors (like + for concatenation) it behaves like a built-in type. In addition, for functions in C standard library string.h can be used in C++ as cstring in std namespace.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "Hello ";
    string str2 = "World";
    string str = str1 + str2;
    cout << str;
}
```

Function	Description (Member Function)
Member Function	
Constructor	Construct string object (public)
Destructor	String destructor (public)
Operator =	String Assignment (public)
Iterators	
begin	Return iterator to beginning (public)
end	Return iterator to end (public)
rbegin	Return reverse iterator to reverse beginning (public)
rend	Return reverse iterator to reverse end (public)
cbegin	Return const_iterator to beginning (public)
cend	Return const_iterator to end (public)
crbegin	Return const_reverse_iterator to reverse beginning (public)
crend	Return const_reverse_iterator to reverse beginning (public)

More about std::string

Sorting and Searching

```
#include<iostream>
#include<algorithm>
using namespace std;
bool compare(int i, int j)
{
    return (i>j);
}
int main()
```

```
{
    int data[] = {34,76,12,45,2};
    sort(data, data+5, compare);
}
```

Binary Search

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
    int data[] = {2,5,9,34,65,78};
    int k = 65;
    if(binary_search(data, data+5, k))
        cout <<  “Found!” << endl;
    else
        cout << “Not Found!” << endl;
}
```

const

The value of const variable cannot be changed after definition, and it must be initialized when defined.

```
const int n = 10;
n = 5;              //Compilation Error, n cannot be changed.
int *p  = 0;
p =  &n;            //Compilation error, as n may be changed by *p
```

Manifest constant	Constant value
Is not type safe	Has its type
Replaced textually in cpp	Visible to compiler
Cannot be watched in debugger	Cane be watched in debugger
Evaluated as many times as replaced	Evaluated only on initialization

const-ness can be used with Pointers in 2 ways:

- pointer to constant data, where pointee (pointed data) cannot be changed.
- Constant Pointer, where pointer (address) cannot be changed

```
int m = 4;
const int n = 5;
const int * p = &n;
n = 6;                    //Error, n is constant and cannot be changed
*p = 7;                   //Error, p points to const data, cannot be changed.
p = &m;                   //Okay

const int i = 5;
int * j = &i;                   //Error
```

```
int m = 4, n=4;
int * const p = &n;
n = 6;                    //Okay
*p = 7;                   //Okay
p = &m;                   //Error, p is const pointer and cannot be chnaged.
```

```
const in m = 4;
const int n = 4;
const int * const p = &n;
n = 6;                          //Error n is constatnt
*p = 7;                         // Error, p points to const data
p = &m;                         //Error, p is const pointer
```

```
To decide constness, draw a mental line through *
int n = 5;
int * p = &n;                //non-constant pointer to non-constant pointee
const int * p = &n;          //non-constant pointer to constant pointee
int * const p = &n;          //constant pointer to non-constant pointee
const int * const p = &n;    //constant pointer to constant pointee
```

inline function: just a function like any other. The function prototype is preceded with keyword inline. An inline function is expanded at the site of its call and the overhead of passing parameter of caller and callee function is avoided.

Reference: An alias/synonym for an existing variable. They have same memory location, changing one changes the another.

```
int i = 10;
int &j = i;  //j is reference to i
```

Wrong Declaration	Reason	Correct Declaration
int& i;	No variable (address) to refer to, must be initialized.	int& i = j;
int& j = 5;	No address to refer to as 5 is constant	const int& j = 5;
int& i = j + k;	Only temporary address	const int& i = j +k;

Call by reference

```
void swap(int& x , int& y)
{
    int t = x;
    x = y;
    y = t;
}
void main()
{
    int a =10, b =5;
    swap(a,b);
}
```

A reference parameter may be changed in the function. We can use const to stop reference parameter being changed.

```
int ref_const (const int& x)
{
    return x+1;
}
int main()
{
    int a =10, b;
    b = ref_const(10);
    return 0;
}
```

C++ allows programmer to assign default values to function parameters. Default parameters are specified while prototyping the function.
All parameters to the right of parameter with default argument must have default argument.
Default arguments cannot be redefined. All non-default parameters are needed in a call.

```
void f(int, double = 0.0, char*);          //Error missing default parameter for
                                             parameter 3
void g(int, double = 0.0, char* =  NULL);  //Okay
```

Function Overloading/ Static Polymorphism: functions with same name, similar functionality but different algorithm and identified by different interface data types.

```
typedef struct {int data[10][10];}  Mat;
typedef struct {int data[1][10];}   vecRow;
typedef struct {int data[10][1];}   vecCol;

void Multiply(const Mat& a,    const Mat& b,    Mat& c);
void Multiply(const vecRow& a, const Mat& b,    vecRow& c);
void Multiply(const Mat& a,    const vecCol& b, vecCol& c);
void Multiply(const vecCol& a, const vecRow& b, Mat& c);
void Multiply(const vecRow& a, const vecCol& b, int& c);
```

Two functions with same signature but different return types cannot be overloaded.

```
int    Area (int a, int b) { return a*b; }
double Area (int a, int b) { return a*b; }   //ERROR
```

Overload resolution:

- Identify the set of candidate functions.
- From the set of candidate function, identify set of viable functions.
- Select best viable function
  - Exact
  - Promotion
  - Standard type conversion
  - User defined type conversion

Exact: lvalue to rvalue conversion, Array to Pointer conversion, Function to pointer conversion
Promotion: char to int; float to double; enum to int/short/unsigned int; bool to int
Standard conversion: integral type conversion, less precise floating to more precise, conversion from integral to floating point types, boo conversion.

Operator Functions Implicit for predefined operators of builtin types and cannot be redefined.

Operator Expression	Operator Function
a + b	operator+(a, b)
a = b	operator=(a, b)
c = a + b	operator=(c, operator+(a, b))

Operator overloading (ad-hoc polymorphism): different operators have different implementations depending on their argument.
Intrinsic properties of overloaded operator cannot be changed. (arity, precedence, associativity)
For unary prefix, use MyType& operator++(MyType& s1)
For unary postfix, use MyType& operator++(MyType& s1, int)
operator:: (scope), operator. (member access), operator.* (member access through pointer) operator sizeof and operator?: (ternary conditional) cannot be overloaded.
The overloads of operator&&, operator||, operator, (comma) lose their special proprieties.
The overload-> must return a raw pointer or object reference.

OOP in C++

A class is an implementation of a type, contains data members/ attributes and offer data abstraction/ encapsulation of Object Oriented Programming. Classes also provide access specifier for members to enforce data hiding that separate implementation from interface.
An object of class is an interface created according to its blue print. It comprises data members that specify its state, also support member functions that specify its behaviour.
An implicit this pointer holds the address of an object. Type of this pointer for class X:
X * const this;
this pointer is accessible only in member functions.

```
#include<iostream>
using namespace std;
class X{
public:
    int m1,m2;
    void f(int k1, int k2)
    {
        m1 = k1;                            //Implicit access without this pointer
        this->m2 = k2;                      //Explicit access with this pointer
        cout << "Id = " << this << endl;
    }
};
int main()
{
    X a;
    a.f(2,3);
    cout << "Addr = " << &a << endl;
    return 0;
}
```

this pointer is implicit passed to methods.
Classes provide access specifiers for members (data as well as function) to enforce data hiding that separates implementation from interface.
private: accessible inside definition of class.
public: accessible everywhere.
Information Hiding: The private part of class form its implementation because the class alone should be concerned with it and have the right to change it. The public part of a class constitute its interface which is available for all other for using the class.
Customarily, we put attribute in private part and member functions in public part. This ensures:

- The state of object can be changed only through one of its member functions
- The behaviour of an object is accessible to other through member functions.

Classes wrap data and function acting on the data together as a single data structure. This is Aggregation. Access specifier defines the visibility outside the class, this helps in hiding information about the implementation details of the data members and method. This concept is known as Encapsulation.

Constructor & Destructor

Constructor is implicitly called at instantiation as set by the compiler. Destructor is implicitly called at end of scope. If user has not provided any constructor or destructor, compiler provides free default constructor and destructor, they have no code in the body.

```
#include<iostream>
using namespace std;
class Stack{
private:
    char *data_; int top_;
public:
    Stack() : data (new char[10]), top_(-1) {}
    int empty() { return (top == -1); }
    void push(char x) { data[++top_] = x; }
    void pop() { top_--; }
    char top() { return data[top_]; }
    ~Stack() { delete [] data; }
};
int main()
{
    Stack s;
    for(int i = 0; i < 5; i++)
        s.push('A'+ i);
    while(!s.empty())
    {
        cout << s.top();
        s.pop();
    }
}
```

Constructor	Member Functions
• Is a static member function without this pointer.	• Has implicit this pointer
• Name is same as name of class.	• Any name different from name of class.
• Has no return type, not even void.	• Must have at least one return type.
• Implicit call by instantiation	• Explicit call by the object.

Destructor	Member Functions
• Has implicit this pointer.	• Has implicit this pointer
• Name is ~ followed by name of class.	• Any name different from name of class.
• Has no return type, not even void and does not return anything.	• Must have at least one return type and a return statement.
• Implicit call by instantiation	• Explicit call by the object.

Order of initialization in constructor does not depend on the order in the initialization list. It depends on the order of data members in the definition.

Fail Scenario

```
#include<iostream>
#include<cstring>
#include<cstdlib>
using namespace std;
class String{
    size_t len_; char *str_;
public:
    String(char *s) : str_(strdump(s)), len_(strlen(str_))
    {
        cout << "Ctor";
        print();
    }
    ~String()
    {
        cout << "Dtor";
        print();
        free(str_);
    }
    void print(){
        cout << "(" << str_ << ": " << len_ << ")" << endl;
    }

};
```

```
int main()
{
    String s = "C++";
    s.print();
    return 0;
}
```
Here len_ precedes str_ in list of data member, len(strlen(str_)) is executed before str_(strdump(s)). When strlen(str_) is called str_ is still uninitialized, which is a garbage value, May also cause program to crash.

Copy Constructor
```
#include<iostream>
#include<cmath>
using namespace std;
class Complex{ double re_, double im_;
public:
    Complex(double re, double im) : re_(re)), im_(im)
    {
        cout << "Ctor : "; print();
    }
    Complex(const Complex& c) : re_(c.re_)), im_(c.im_)
    {
        cout << "Copy Ctor : "; print();
    }
    ~Complex()
    {
        cout << "Dtor : "; print();
    }
    double norm() { return sqrt(re_ * re_ + im_ * im_); }
    void print(){
        cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl;
    }

};
int main()
{
    Complex c1(4.2,5.4), c2(c1);
}
```
```
Ctor : |4.2+j5.4| = 6.8410
Copy Ctor : |4.2+j5.4| = 6.8410
Dtor : |4.2+j5.4| = 6.8410
Dtor : |4.2+j5.4| = 6.8410
```

If no copy constructor is provided by the user, the compiler supplies a free one. Free copy constructor cannot initialize the object to proper values. It perform Shallow Copy.
Shallow Copy (bitwise copy): An object is created by simply copying the data of all variables of original object, works well if none of the variables are defined in heap/ free store. For dynamically crated variables, the copied objects refers to the same memory location. This creates ambiguity.
Deep Copy (Lazy Copy) An object is created by copying data of all variables except the ones on heap. Allocates similar memory resources with same value to the object. Need to explicitly define copy constructor and assign dynamic memory as required.

Copy Assignment operator
```
#include<iostream>
#include<cmath>
using namespace std;
class Complex{ double re_, double im_;
public:
    Complex(double re, double im) : re_(re)), im_(im)
    {
        cout << "Ctor : "; print();
    }
    Complex(const Complex& c) : re_(c.re_)), im_(c.im_)
    {
        cout << “Copy Ctor : ”; print();
    }
    Complex& opearator=(const Complex& c) : re_(c.re_)), im_(c.im_)
    {
        cout << “Copy Assignemnt : ”; print();
        return *this;            //return *this for chaining
    }
    ~Complex()
    {
        cout << “Dtor : ”; print();
    }
    double norm() { return sqrt(re_ * re_ + im_ * im_); }
    void print(){
        cout << “|” << re_ << “+j” << im_ << “| = ” << norm() << endl;
    }

};
int main()
{
    Complex c1(4.2,5.4), c2(5.8,3.8), c3(c2);
    c3.print();
    c3 = c1;
    c3.print();
}
```

```
Ctor : |4.2+j5.4| = 6.8410
Ctor : |5.8+j3.8| = 6.9339
Copy Ctor : |5.8+j3.8| = 6.9339
|5.8+j3.8| = 6.9339
|4.2+j5.4| = 6.8410
Dtor : |4.2+j5.4| = 6.8410
Dtor : |5.8+j3.8| = 6.9339
Dtor : |4.2+j5.4| = 6.8410
```

```
#include<iostream>
#include<cstring>
#include<cstdlib>
using namespace std;
class String{ char *str_, size_t len;
public:
    String(char *s) : str_(strdump(s)), len_(strlen(str_)) { }
    String(const String& s) : str_(strdump(s.str_)), len_(s.len_) { }
    String& opeartor=(const String& s)
    {
        if(this != &s){                 //check for self copy
            free(str);
            str_ = strdup(s.str_);
            len_ = s.len_;
        }
        return *this;

    }
    ~String() { free(str_); }
    void print(){
        cout << “(” << str_ << “: ” << len_ << “)” << endl;
    }

};
```

```
int main()
{
    String s1 = “Football”, s2 = “Cricket”;
    s2.print();
    s2 = s1;
    s2.print();
    return 0;
}
```

```
(Cricket : 7)
(Football: 8)
```

“this != &s” comes to check if there is self copy, if there is no check, it will free the variable and behaviour of that variable will be ambiguous.

const-ness

constant objects User defined Data types can also be made constant, if an object is constant none of the data members can be changed. The type of this pointer of constant object of class is:
const MyClass * const this;
A constant objects cannot invoke normal methods of class lest these methods change the object.

```
#include <iostream>
using namespace std;
class MyClass{
    int privmember;
public:
    int pubmember;
    MyClass(int pri, int pub) : privmember (pri) , pubmember(pub) {}
    int getmember() { return privmember; }
    void setmember(int i) {privmember = i; }
    void print() { cout << privmember << “ , ” << pubmember << endl; }
};
int main()
{
    const MyClass myconstobj(5,6);
    cout << myconstobj.getmember() << endl;         //Error1
    myconstobj.setmember(7);                        //Error2
    myconstobj.pubmember  = 8;                      //Error3
    myconstobj.print();                             //Error4
}
```

It is not allowed to invoke methods or make changes in constant object myconstobj.
constant Member Function To declare a constant member function, we use the keyword const between function header and body. It expect this pointer as const MyClass * const this;
Also, no data member can be changed.
Non-constant objects can invoke constant member functions and non-constant member function.
Constant objects can only invoke constant member function.

```
#include <iostream>
using namespace std;
class MyClass{
    int privmember;
public:
    int pubmember;
    MyClass(int pri, int pub) : privmember (pri) , pubmember(pub) {}
    int getmember() const { return privmember; }
    void setmember(int i) {privmember = i; }
    void print() const { cout << privmember << “ , ” << pubmember << endl; }
};
int main()
{
    MyClass myobj(1,2);
    //non-const object can invoke all member functions
    cout << myobj.getmember() << endl;
    myobj.setmember(7);
    myobj.pubmember  = 8;
    myobj.print();
    const MyClass myconstobj(5,6);
    //const object cannot allow any change
    cout << myconstobj.getmember() << endl;
    //myconstobj.setmember(7);                //can't invoke non-const member func
    //myconstobj.pubmember  = 8;              //cant update data member
    myconstobj.print();
}
```

constant Data Members A constant data member cannot be changed even in non-constant object, and must be initialized on the initialization list.

```
#include <iostream>
using namespace std;
class MyClass{
    const int cprimem;
    int ncprimem;
public:
    const int cpubmem;
    int ncpubmem;
    MyClass(int cpri, int ncpri, int cpub, int ncpub): cprimem(cpri),
                      ncprimem(ncpri), cpubmem(cpub), ncpubmem(ncpub) {}
    int getcpri() { return cprimem; }
    void setcpri(int i) { cprimem = i; }      //Error: Assignment to const data
    int getncpri() { return ncprimem; }
    void setncpri(int i) { ncprimem = i; }
};
int main()
{
    MyClass myobj(1,2,3,4);
    cout << myobj.getcpri() << endl;
    cout << myobj.getncpri() << endl;
    myobj.setncpri(7);
    cout << myobj.cpubmem << endl;
    //myobj.cpubmem = 10;              //Error: Assignment to const data
    cout << myobj.ncpubmem() << endl;
    myobj.cpubmem= 20;
}
```

mutable Data Members mutable data member is changeable in a constant object. mutable is provided to model Logical (Semantic) const-ness against the default bit-wise (syntactic) const-ness of C++

- mutable is applicable only yo data members and only to variables
- Reference data members cannot be declared mutable
- static data members cannot be declared mutable
- const data members cannot be declared mutable

```
#include <iostream>
using namespace std;
class MyClass{
    int mem_;
    mutable int mutmem_;
public:
    MyClass(int m, int mm): mem_(m), mutmem_(mm){}
    int getmem() const { return mem_; }
    void setmem(int i) { mem_ = i; }
    int getmutmem() const { return mutmem_; }
    void setmutmem(int i) const { mutmem_ = i; }
};
int main()
{
    const MyClass myconstobj(1,2);
    cout << myconstobj.getmem() << endl;
    //myconstobj.setmem(3);
    cout << myconstobj.getmutmem () << endl;
    myconstobj.setmutmem(10);
}
```

static

static data members is associated with class not object, is shared by all objects of a class.

- Need to be defined outside the class scope to avoid linker error.
- Must be initialized in source file
- is constructed before main() starts and destructed after main() ends.
- Can be accessed
  - with class-name followed by the scope resolution (::)
  - as a member of any object of the class

Order of initialization of static data members does not depend on their order in the definition of the class. It depends on the order their definition and initialization in the source.

static member functions

- does not have this pointer - not associated with an object
- cannot access non-static data members
- cannot invoke non-static member functions
- may initialize static data members even before any object creation
- cannot be declared as const

```
#include <iostream>
using namespace std;
class PrintJobs{
    int nPages_;
    static int nTrayPages_;
    static int nJobs_;
public:
    PrintJobs(int nP): nPages_(nP){
        ++nJobs;
        cout << “Printing ” << nP << “ Pages” << endl;
         nTrayPages_ -= nP;
        }
    ~PrintJobs(){--nJobs;}
    static int getjobs() { return nJobs_; }
    static int checkpages() { return nTrayPages_; }
    static void loadpages(int nP) { nTrayPages_ += nP; }
};
int PrintJobs::nTrayPages_ = 500;
int PrintJobs::nJobs_ = 0;
int main()
{
    cout << “Pages = ” << PrintJobs::checkpages() << endl;
    PrintJobs job1(10);
    cout << “Jobs = ” << PrintJobs::getjobs() << endl;
    cout << “Pages = ” << PrintJobs::checkpages() << endl;
    {
        PrintJobs job1(30), job2(20);
        cout << “Jobs = ” << PrintJobs::getjobs() << endl;
        cout << “Pages = ” << PrintJobs::checkpages() << endl;
        PrintJobs::loadpages(100);
    }
    cout << “Jobs = ” << PrintJobs::getjobs() << endl;
    cout << “Pages = ” << PrintJobs::checkpages() << endl;
}
```

Jobs = 0

```
Pages = 500
Printing 10 pages
Jobs = 1
Pages = 490
Printing 30 pages
Printing 20 pages
Jobs = 3
Pages = 440
Jobs = 1
Pages = 540
```

Static Member function	Non-static Member function
• Cannot access non-static data members or methods • Can be invoked anytime during program execution • cannot be virtual or constant • Constructor is static though not declared static	• Can access non-static data members and methods • Can be invoked only during lifetime of the object • May be virtual or constant • There cannot be non-static constructor.

Singleton Class is a creational design pattern satisfying :

- ensure that only one object of its kind exists
- provides single point of access.

```
class Singleton {
private:
    static Singleton* instance_;
    Singleton() {
        std::cout << "Singleton instance created.\n";
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
public:
    static Singleton* getInstance() {
        if (instance_ == nullptr) {
            instance_ = new Singleton();
        }
        return instance_;
    }
    void doSomething() {
        std::cout << "Doing something with the singleton instance.\n";
    }
};
Singleton* Singleton::instance_ = nullptr;
int main() {
    Singleton* s1 = Singleton::getInstance();
    s1->doSomething();
    Singleton* s2 = Singleton::getInstance();
    s2->doSomething();
    std::cout << "Same instance? " << (s1 == s2 ? "Yes" : "No") << std::endl;
    return 0;
}
```

Meyer's Singleton

```
class Printer{
    bool BlackAndWhite, BothSide;
    Printer(bool bw = false, bool bs = false) : BlackAndWhite(bw),BothSide(bs){
        cout << "Printer Constructed!!" << endl;
    }
    ~Printer(){
        cout << “Printer Destructed!!” << endl;
    }
public:
    static const Printer& printer(bool bw = false, bool bs = false){
        static Printer myPrinter(bw,bs);
        return myPrinter;
        }
    void print(int nP) const{
        cout << “Printing ” << nP << “ Pages” << endl;
    }
};
int main()
{
    Printer::printer().print(10);
    Printer::printer().print(12);
}
```

friend function
A friend function of a class has access to private and protected members of the class (breaks the encapsulation) in addition to public members.

- does not have name qualified with class scope
- is not called with an invoking object of the class
- can be declared friend in more than one class.

A friend function can be a

- global function
- a member function of a class
- a function template

```
#include<iostream>
using namespace std;
class MyClass{
    int data_;
public:
    MyClass(int i) : data_(i) { }
    friend void dispaly (const MyClass& a);
};
void dispaly (const MyClass& a) {
    cout << “data = ” << a.data_;
}
int main()
{
    MyClass obj(10);
    dispaly(obj);
}
```

friend class has access to the private and protected members of the class in addition to public members. It can be declared friend in more than one class.

```
#include<iostream>
using namespace std;
class Node;
class List{
    Node *head;
    Node *tail;
public:
    List(Node *h = nullptr) : head(h), tail(h) { }
    void display();
    void append(Node *p);
};
```

```
class Node{
    int info;
    Node *next;
public:
    Node(int i) : info(i), next(nullptr) { }
    friend class List;
};
void List::dispaly() {
    Node *ptr = head;
    while(ptr) {
        cout << ptr->info << " ";
        ptr = ptr->next;
    }
}
int main()
{
    List l;
    Node n1(1), n2(2), n3(3);
    l.append(&n1);
    l.append(&n2);
    l.append(&n2);
}
```

Overloading of operator for UDTs

operator overloading help us to build complete algebra for UDT's much in the same line as is available for built-in types. Examples:

1) Complex type: Add(+), Subtract(-), Multiply(*), Divide(/), Conjugate(!), Compare(==, !=,..)
2) Fraction type: Add(+), Subtract(-), Multiply(*), Divide(/), Normalize(unary *), Compare(==, !=,..)
3) Matrix: Add(+), Subtract(-), Multiply(*), Divide(/), Invert(!), Compare(==, !=,..)
4) Direct IO: read(<<) and write(>>) for all types

Advanced examples

1) Smart Pointers: De-reference (unary *),Indirection(->), copy(=)
2) Function Pointer or functors (invocation())

Non-member operator function

Binary operator

```
MyType a,b;
MyType operator+ (const MyType&, const MyType&);              //Global
friend MyType operator+(const MyType&, const MyTpe&);         //Friend
```

Unary operator

```
MyType operator++ (const MyType&);                            //Global
friend MyType operator++ (const MyType&);                     //Friend
```

Operator Expression	Operator Function
`a + b`	`operator+(a,b)`
`a = b`	`operator=(a,b)`
`++a`	`operator++(a)`
`a++`	`operator++(a,int)`
`c = a + b`	`operator=(c, operator+(a,b))`

Member operator function

Binary operator

```
MyType a,b;
MyType operator+ (const MyType&);              //Mtype is a class
```

The left operand is invoking object – right taken as parameter

Unary operator

```
MyType operator-();                            //Opearotor fn for unary minus
MyType operator++();                           //Pre Increment
MyType operator++(int);                        //Post Increment
```

Operator Expression	Operator Function
`a + b`	`a.operator+(b)`
`a = b`	`a.operator=(b)`
`++a`	`a.operator++()`
`a++`	`a.operator++(int)`
`c = a + b`	`c.operator=(a.operator+(b))`

Rules:

- No new operator such as **, <>, or & | can be defined
- Preserves arity, precedence, associativity
- ::(scope resolution), .(member access), .*(member access through pointer), sizeof and ?: (Ternary conditional) cannot be overloaded
- The overloads of operator &&, ||, and ,(comma) lose their special properties: short-circuit evaluation and sequencing

```
#include<iostream>
class Complex{ double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0 ) : re_(re), im_(im){}
    void display(){
        cout << "(" << re_ << “ +j ” << im_ << ")";
    }
    Complex operator+(const Complex& c){
        Complex a;
        a.re_ = re_ + c.re_;
        a.im_ = im_ + c.im_;
        return a;
    }
    Complex operator++(){
        ++re_;
        return *this;
    }
    Complex operator++(int){
        Complex c(re_,im_);
        ++re_;
        return c;
    }
};
int main()
{
    Complex c1(4.2,5.4), c2(3.4,9.58), c3;
    c3 = c1 + c2;
    c1.display();
    cout << " + ";
    c2.display();
    cout << " = ";
    c3.display();
    cout << endl;
    ++c1;
    cout << "c1 = ";
    c1.display();
    cout << endl;
    c3 = c2++;
    cout << "c2 = ";
    c2.display();
    cout << endl;
    cout << "c3 = ";
    c3.display();
    cout << endl;
}
```

```
(4.2 +j 5.4) + (3.4 +j 9.58) = (7.6 +j 14.98)
c1 = (5.2 +j 5.4)
c2 = (4.4 +j 9.58)
c3 = (3.4 +j 9.58)
```

Issue: Using global function, accessing private data members inside operator function is difficult. It increasing writing overhead, makes code complicated. We can also use member functions, but if left operand is not an object of the class type, it cannot be overloaded through member function.
To handle such situation, we require friend functions

```
#include<iostream>
class Complex{ double re_, im_;
public:
    explicit Complex(double re = 0.0, double im = 0.0 ) : re_(re), im_(im){}
    void display(){
        cout << "(" << re_ << " +j " << im_ << ")";
    }
    friend Complex operator+(const Complex& a, const Complex& b){
        return Complex(a.re_ + b.re_, a.im_ + b.im_);
    }
    friend Complex operator+(const Complex& a, double b){
        Complex c(b);
        return a+c;
    }
    friend Complex operator+(double a, const Complex& b){
        Complex c(a);
        return c+b;
    }
};
int main()
{
    Complex c1(4.2,5.4), c2(3.4,9.58), c3;
    c3 = c1 + c2;
    c3.display();
    c3 = c1 + 9.9;
    c3.display();
    c3 = 1.25 + c2;
    c3.display();
}
```

Overloading IO operator: operator << and operator >>

```
ostream& operator<<(ostream& os, const Complex& a);        //Global function
ostream& ostream::operator<<(const Complex& a);            //Member function of ostream
ostream& Complex::operator<<(ostream& os);                 //Member function of Complex
```

Return by reference of ostream, so that chaining would work.
For global function, need to break encapsulation, declare everything public.

```
ostream& operator<<(ostream& os, const Complex& a)
{
    os << a.re_ + " + j" + a.im_;
    return os;
}
istream& operator<<(istream& is, const Complex& a)
{
    is << a.re_ + " + j" + a.im_;
    return is;
}
```

ostream member function operator << is not allowed, ostream member function operator << can be done, but the invocation would to a << cout;
So, we need to use friend function

```
#include<iostream>
class Complex{ double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0 ) : re_(re), im_(im){}
    friend ostream& operator <<(ostream& os, const Complex& b);
    friend istream& operator >>(istream& is, Complex& a);
};
friend ostream& operator <<(ostream& os, const Complex& a)
{
    os << a.re_ + " + j" << a.im_ << endl;
    return os;
}
friend istream& operator >>(istream& is, Complex& a)
{
    is >> a.re_ >> a.im_;
    return is;
}
int main()
{
    Complex c;
    cin >> c;
    cout << c;
}
```

Guidelines:

- Use global function when encapsulation is not a concern
- Use member function, when left operand is necessarily an object of a class where the operator function is a member (operator=, operator new, operator new[], operator delete)
- Use friend function otherwise
- While overloading an operator, try to preserve its natural semantics for builtin types
- Decide the return type based on natural semantics for builtin types
- Consider the effect of casting on operands

Namespace

A namespace is a declarative region that provides a scope to the identifiers inside it. It is used to organise code into logical groups and to prevent name collisions that can occur especially when your code includes multiple libraries. namespace provides a class-like modularization without class-like semantics.

```
namespace MyNamespace {
    int myData;
    void myFunction() {cout << "MyNamespace myFunction" << endl;}
    class myClass {
        int data;
    public:
        myClass(int d) : data(d) {}
        void display() {cout << "myClass data = " << data  << endl;}
    };
}
int main() {
    MyNamespace::myData = 10;
    cout << MyNamespace::myData << endl;
    MyNamespace::myFunction();
    MyNamespace::myClass obj(25);
    obj.display();
}
```

Scenario 1: cstlib has a function int abs(int n); that returns the absolute value. You need a special int abs(int n); that returns absolute value of parameter if n is between -128 and 127. Otherwise it return 0.

```
#include<iostream>
#include<cstdlib>
namespace myNS {
    int abs() {
        if(n < -128) return 0;
        if(n > 127) return 0;
        if(n < 0) return -n;
        return n;}
}
int main() {
    std::cout << myNS::abs(-230) << endl;     //0
    std::cout << myNS::abs(-46) << endl;      //46
    std::cout << myNS::abs(52) << endl;       //52
    std::cout << myNS::abs(196) << endl;      //0
    std::cout << abs(-230) << endl;           //230
    std::cout << abs(-46) << endl;            //46
    std::cout << abs(52) << endl;             //52
    std::cout << abs(196) << endl;            //196
}
```

namespace features:

1) Nested namespace

```
#include<iostream>
using namespace std;
int data = 0;
namespace name1 {
    int data = 1;
    namespace name2 {
        int data = 1;
    }
}
int main() {
    cout << data << endl;                     //0
    cout << name1::data << endl;              //1
    cout << name1::name2::data << endl;       //2
}
```

2) ‘using’ namespace

```
#include<iostream>
using namespace std;
int data = 0;
namespace name1 {
    int v11 = 1;
    int v12 = 2;
}
namespace name2 {
    int v21 = 3;
    int v22 = 4;
}
using namespace name1;
using name2::v21;
int main() {
    cout << v11 << endl;
    cout << name1::v12 << endl;
    cout << v21 << endl;
    cout << v22 << endl;  //Treated as undefined
}
```

3) Global namespace

```
#include<iostream>
using namespace std;
int data = 0;
namespace name1 {
    int data = 1;
}
int main() {
    using name1::data;
    cout << data << endl;               //1
    cout << name1::data << endl;        //1
    cout << ::data << endl;            //0
}
```
4) std namespace : Entire C++ stadard Library is put in its own namespace, called std
5) namespaces are open
```
#include<iostream>
using namespace std;
namespace open {
    int x = 10;
}
namespace open {
    int y = 20;
}
int main() {
    using namespace open;
    x = y =  30;
}
```

namespace	class
• Every namespace is not a class	• Every class defines a namespace
• can be reponed and more declaration added to it.	• cannot be reopened
• No instance can be created	• has multiple instance
• using-declaration can be used to shortcut namespace qualification	• No using-like declaration for a class.
• may be unnamed	• Cannot be unnamed

Inheritence

Manager ISA Employee [Single Inheritance]

```
class Employee;
class Manager : public Employee;
```

TwoWheeler ISA Vehicle; ThreeWheeler ISA Vehicle [Hybrid Inheritance]

```
class Vehicle;
class TwoWheeler : public Vehicle;
class ThreeWheeler : public Vehicle;
```

RedRose ISA Rose ISA Flower [Multi-Level Inheritance]

```
class Flower;
class Rose : public Flower;
class RedRose : public Rose;
```

Data Members

- Derived class inherits all data members of Base class
- Derived class may add data members of its own

Member Functions

- Derived class inherits all member functions of Base class
- Derived class may override a member function of Base class by redefining it with same signature
- Derived class may overload a member function of Base class by redefining it with same name but different signature.

Access Specification

- Derived class cannot access private members of Base class
- Derived class can access protected members of Base class

Construction-Destruction

- A constructor of Derived class must first call a constructor of Base class to construct the Base class instance of Derived class
- The destructor of the Derived class must call the destructor of the Base class to destruct the base class instance of the Derived class.

```
class B { //Base class
    int data1_B;
public:
    int data2_B;
    //..
};
class D : public B{ //Derived class
    //Inherits B::data1_B
    //Inherits B::data2_B
    int data1_D;
public:
    //..
};
B b;
D d;
```

d cannot access data1_B even though is a part of d! d can access data2_B.

1) Derived does not inherit Constructor and Destructor of Base class but have access to them.

2) Derived class doe not inherit the static member functions of the Base class

3) Derived class doe not inherit friend functions of Base class

```
class B { //Base class
public:
    void f(int i);
    void g(int i);
    //..
};
class D : public B{ //Derived class
public:
    //Inherits B::f(int)
    void f(int i);        //Overrides B::f(int)
    void f(string&);      //Overloads B::f(int)
    //Inherits B::g(int)
    void h(int i);        //Adds D::h(int)
    //..
};
B b;
D d;
b.f(1);                   //Calls B::f(int)
b.g(2);                   //Calls B::g(int)
d.f(3);                   //Calls D::f(int)
d.g(4);                   //Calls B::g(int)
d.f(red);                 //Calls D::f(string&)
d.h(5);                   //Calls D::h(int)
```

protected Access: Derived class can access protected members of Base class. No other class or global function can access protected members of Base class

Example 1:

```
class B {
protected:
    int data_;
public:
    friend ostream& operator << (ostream& os, const B& b)
    {
        os << “B Object : ”;
        os << b.data_ << endl;
        return os;
    }
};
class D : public B {
    int info_;
public:
    ...
};
B b(0);
cout << b;          //B Object : 0
D d(1,2);
cout << d;          //B Object : 1
```

Example 2:

```
class B {
protected:
    int data_;
public:
    friend ostream& operator << (ostream& os, const B& b)
    {
        os << “B Object : ”;
        os << b.data_ << endl;
        return os;
    }
};
class D : public B {
    int info_;
public:
    friend ostream& operator << (ostream& os, const D& d)
    {
        os << “D Object : ”;
        os << d.data_ << “  ” << d.info_ <<endl;
        return os;
    }
};
B b(0);
cout << b;          //B Object : 0
D d(1,2);
cout << d;          //D Object : 1  2
```

Object Lifetime:

```
class Base {
protected:
    int data_;
public:
    Base(int d = 0) : data_(d) {
        cout << "Base constructor :: " << data_ << endl;
    }
    ~Base() {
        cout << "Base destructuor :: " << data_ << endl;
    }
};
class Derived : public Base {
public:
    int info_;
    Derived(int i = 1) : info_(i) {
        cout << "Derived constructor1 :: " << data_ << "   " << info_ << endl;
    }
    Derived(int i, int d) : info_(i), Base(d) {
        cout << "Derived constructor2 :: " << data_ << "   " << info_ << endl;
    }
    ~Derived() {
        cout << "Derived destructuor :: " << data_ << "   " << info_ << endl;
    }
};
int main() {
    Base b1;
    Base b2(2);
    Derived d1;
    Derived d2(3);
    Derived d3(4,5);
    return 0;
}
Base constructor :: 0
Base constructor :: 2
Base constructor :: 0
Derived constructor1 :: 0   1
Base constructor :: 0
Derived constructor1 :: 0   3
Base constructor :: 5
Derived constructor2 :: 5   4
Derived destructuor :: 5   4
Base destructuor :: 5
Derived destructuor :: 0   3
Base destructuor :: 0
Derived destructuor :: 0   1
Base destructuor :: 0
Base destructuor :: 2
Base destructuor :: 0
```

private Inheritance

```
class Base;
class Derived : private Base;
```

private Inheritance does not mean generalization/specialization.
Compiler converts a derived class object into base class if the inheritance relationship is public.

```
class Person { .... };
class Student : public Person { .... };
void eat(const Person& p);
void study(const Student& s);
Person p;
Student s;
eat(p);             //fine, p is a Person
eat(s);             //fine, s is a Student
study(s);           //fine
study(p);           //error! p isn’t a Student
```

Compiler will not convert a derived class object into Base class object if the inheritance is private.

```
class Person { .... };
class Student : private Person { .... };
void eat(const Person& p);
void study(const Student& s);
Person p;
Student s;
eat(p);             //fine, p is a Person
eat(s);             //error!, Student isn’t a Person
```

		Inheritance		
		public	**protected**	**private**
Visibility	**public**	public	protected	private
	protected	protected	protected	private
	private	private	private	private

Polymorphism

Casting is performed when a variable of one type is used in place of some other type. Casting can be implicit or explicit

```
int i =3;
double d = 2.5;
d = i;              //Implicit: int to double
i = d;              //Implicit: warning ‘=’ : conversion from ‘double’ to ‘int’ :
                      possible loss of data
d = (double)i;      //Explicit: int to double
i = (int)d;         //Explicit: double to int
```

Implicit Casting	Explicit Casting
• Done automatically	• Done programmatically
• No loss of data for promotion, compiler will be silent.	• Data loss may or may not take place, compiler will be silent
• Possible loss of data for demotion compiler will issue warning	
• No throwing of exceptions is type safe	• May throw for wrong type casting
• Requires no special syntax	• Require cast operator for conversion C style operator (<type>) C++ style operator: const_cast, static_cast, dynamic_cast, reinterpret_cast
• Avoid if possible	• Avoid C style cast, use C++ style cast
• Possible only in static time	• Possible in static as well as in dynamic time.
• Maybe disallowed for the user-defined types but cannot be disallowed for built-in types	• Maybe defined for user-defined types

Type Casting Rules: Built-in Types

Casting is safe for promotion (All the data types of the variables are upgraded to the data type of the variable with larger data type)

bool → char → short int → int → unsigned int → long → unsigned → long long → float → double → long double

It does not invoke any conversion function, only re-interprets the binary representation

Casting is unsafe for demotion – may lead to loss of data.

Implicit casting between different pointer types is not allowed. Any pointer can be implicitly cast to void* (with loss of type); but cannot be implicitly cast to any pointer type.

```
int i = 1, *p = &i, a[10];
double d = 1.1, *q = &d;
void *r;
q = p;              //error: cannot convert ‘int*’ to ‘double*’
p = q;              //error: cannot convert ‘double*’ to ‘int*’
q = (double*)p;     //Okay
p = (int*)q;        //Okay
r = p;              //Okay to convert from ‘int*’ to ‘void*’
p = r;              //error: invalid conversion from ‘void*’ to ‘int*’
p = (int*)r;        //Okay
p = a;              //Okay, by array pointer duality
a = p;              //error: incompatible types in assignment of ‘int*’ to ‘int[10]’
```

Implicit casting between pointer and numerical type is not allowed. However, explicit casting between pointer and integral type (int or long) is common practice to support task like serialization(save a file) and de-serialize(open a file). Care should be taken with these explicit cast to ensure the integral type is of the same size as of the pointer.

```
int i = 1, *p = 0;
long j;
i = p;              //error: invalid conversion from ‘int*’ to ‘int’
p = i;              //error: invalid conversion from ‘int’ to ‘int*’
i = (int)p;         //error: cast from ‘int*’ to ‘int’ losses precision
p = (int*)i;        //Warning: cast to pointer from integer to different size
j = (long)p;        //Okay
p = (int*)j;        //OKay,
```

Type Casting Rules: Unrelated Classes

Implicit Casting between unrelated classes is not permitted

```
class A { int i; };
class B {double d; };
A a;
B b;
A *p = &a;
B *q = &b;
a = b;              //error: binary ‘=’ : no operator which takes a right-hand operand
                      of type ‘B’
a = (A) b;          //error: ‘type cast’ : cannot convert from ‘B’ to ‘A’
b = a;              //error: binary ‘=’ : no operator which takes a right-hand operand
                      of type ‘B’
b = (B)a;           //error: ‘type cast’ : cannot convert from ‘A’ to ‘B’
p = q;              //error: ‘=’ : cannot convert from ‘B*’ to ‘A*’
q = p;              //error: ‘=’ : cannot convert from ‘A*’ to ‘B*’
p = (A*)&b;         //explicit on pointer: type cast is okay for the compiler
q = (B*)&a;         //explicit on pointer: type cast is okay for the compiler
```

Forced Casting between unrelated classes is dangerous. From above example, `p->i` and `q->d` will point to garbage values

Type Casting Rules: Inherence Hierarchy

Casting in hierarchy is permitted in a limited sense. Up-Casting is safe. Forced Down-Casting is risky.

```
class A { ... };
class B : public A { ... };
A *pa = 0;
B *pb = 0;
void *pv = 0;
pa = pb;            //UPCAST: Okay
pb = pa;            //DOWNCAST: error: ‘=’: cannot convert from ‘A*’ to ‘B*’
pv = pa;            //Okay, but lose the type for A* to void*
pv = pb;            //Okay, but lose the type for B* to void*
pa = pv;
pb = pv;            //error: ‘=’: cannot convert from ‘void*’ to ‘B*’
```

Type Binding

Static type of the object is the type declared for the object while writing the code, compiler sees the static type. The dynamic type of the object is determined by the type of object to which it refers at run-time, compiler does not see the dynamic type.

```
class A { ... };
class B : public A { ... };
int main() {
    A *p;           //Static type of p: A*
    p = new B;      //Dynamic type of p: B*
}
```

Static Binding (early binding): When the function invocation binds to the function definition based on the static type of the objects.

Dynamic Binding (late binding): When function invocation binds to the function definition based on the dynamic type of objects.

Basis	Static Binding	Dynamic Binding
Event Occurrence	Event occur at compile time – Static Binding	Event occur at run time – Dynamic Binding
Information	All information needed to call a function at compile time	All information needed to call a function is known only at run-time
Advantage	Efficiency	Flexibility
Time	Fast Execution	Slow Execution
Actual Object	Actual object is not used for binding	Actual object is used for binding
Alternate name	Early Binding	Late Binding
Example	Method overloading Normal function call Overloaded function call Overloaded operators	Method Overriding Virtual functions

```
class A {
public:
    void f() {
        cout << "A::f()" << endl;
    }
};
class B : public A{
public:
    using A::f;
    void f(int a) {
        cout << "B::f(int)" << endl;
    }
    void f(double a) {
        cout << "B::f(double)" << endl;
    }
};
```

When derived class declares a function with same name as a function in the base class, all overloaded version of that function in the base class are hidden – even if there signatures are different. We can explicitly bring A::f() into scope of B using a 'using' declaration.

Non-Virtual Method	Virtual Method

```
class B {
public:
    void f() { }
};
class D : public B{
public:
    void f() {}
};
int main() {
    B b;
    D d;

    B *p;
    p = &b; p->f();  //B::f()
    p = &d; p->f();  //B::f()
}
```
	```
	class B {
	public:
	    virtual void f() { }
	};
	class D : public B{
	public:
	    virtual void f() {}
	};
	int main() {
	    B b;
	    D d;

	    B *p;
	    p = &b; p->f();  //B::f()
	    p = &d; p->f();  //D::f()
	}
	```

Static Binding, binding is decided by type of pointer.	Dynamic Binding, binding is decided by type of object.

Virtual Destructor

In below code destructor of D is not called.

```
class B {
    int data_;
public:
    B(int a) : data_(a) { cout << "B()" << endl; }
    ~B(){ cout << "~B()" << endl; }
    virtual void f() { cout << a << endl; }
};
class D : public B{
    int *ptr;
public:
    D(int a, int b) : B(a), ptr(new int(b)) { cout << "D()" << endl; }
    ~D(){ cout << "~D()" << endl; delete ptr;}
    void f() { B::f(); cout << " " << b << endl; }
};
int main() {
    B *pb = new B(1);
    B *pd = new D(2,3);
    pb->f();
    pd->f();
    delete p;
    delete q;
}
```

Output:

```
B()
B()
D()
1
2 3
~B
~B
```

To overcome this problem, make destructor of base class as virtual.

```
class B {
    int data_;
public:
    B(int a) : data_(a) { cout << "B()" << endl; }
    virtual ~B(){ cout << "~B()" << endl; }
    virtual void f() { cout << a << endl; }
};
class D : public B{
    int *ptr;
public:
    D(int a, int b) : B(a), ptr(new int(b)) { cout << "D()" << endl; }
    ~D(){ cout << "~D()" << endl; delete ptr;}
    void f() { B::f(); cout << " " << b << endl; }
};
int main() {
    B *pb = new B(1);
    B *pd = new D(2,3);
    pb->f();
    pd->f();
    delete p;
    delete q;
}
```

Pure Virtual function: Virtual function that has no implementation in the base class and must be overridden by ant concrete (non-abstract) derived class.

```
class root { public:
    void f();                   //Non-Virtual function
    virtual void g();           //Virtual function
    virtual void h() = 0;       //Pure Virtual function
};
```

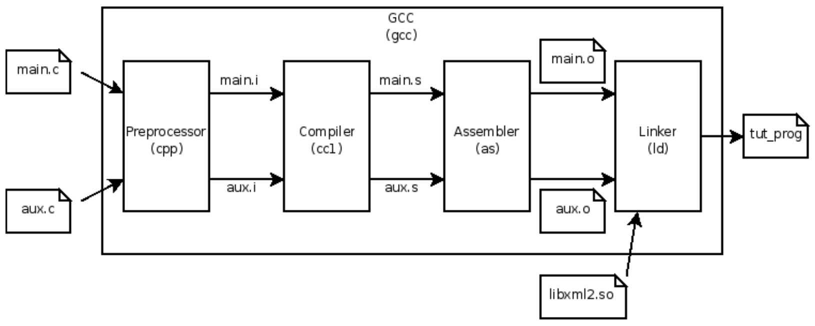
A class containing at least one Pure Virtual Function is called an Abstract Base Class. Pure Virtual function can be inherited or derived in the class. No instance can be created for an Abstract Base class. It can however have other virtual (non pure) and non-virtual member functions as well as data member, data member should be protected and member functions should be public. A concrete class must override and implement all Pure Virtual Functions, so that it can be instantiated.

<u>Exceptions</u> <u>and</u> <u>Telmplates</u>

STL

C++11 and beyond

Extra



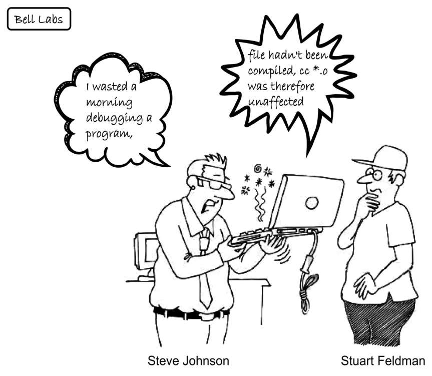
Build Pipeline

The C preprocessor (CPP) has the ability for the inclusion of header files, macro expansion, conditional compilation and line control/ It works on .c .cpp .h files and produces .i files.
The compiler translates the preprocessed C/C++ code into assembly language which is a machine level code in text that contains instructions that manipulates the memory and processor directly. It works on .i files and produces .s files.
The Assembler translates the assembly file into binary machine language or object code it works on .s files and produces .o files.
The linker links are program with the pre-compiled libraries for using their functions and generates the executable binary. It works on .o (static library) .so (shared library or dynamically linked library) and .a (library archive) files and produce a.out file.

make

make is tool which controls the generation of executables and other non-source files of a program from the programs's source files.
The manual process iof build would be difficult in any practical software project due to:

- Volume: project has hundreds of folder, source and header files, need hundreds of command
- Workload: Build needs to be repeated several times a day with code changes in some file
- Dependency: Often change in one file, all translation units do not need to be recompiled
- Diversity: We may need to use different tools for different files, different build flags, different folder structures.


Bell Labs
I wasted a morning debugging a program.
file hadn't been compiled, cc *.o was therefore unaffected
Steve Johnson
Stuart Feldman

```
hello: hello.c main.c
	gcc -o hello hello.c main.c -I
```

Write these lines in a text file named makefile or Makefile and run make command and it will execute the command. Make file comprises a number of rules. Every rule has a target to build, a colon separator(:). Zero or more files on which the target depends on and commands to build on the next line. Hash(#) starts a comment that continue till the end of the line.

<u>Simple and recursive variable</u>

Simply expanded variables (defined by :=) are evaluated as soon as encountered
Recursively expanded variables (defined by =) are lazily evaluated, may be defined after use.

Simply Expanded	**Recursively Expanded**

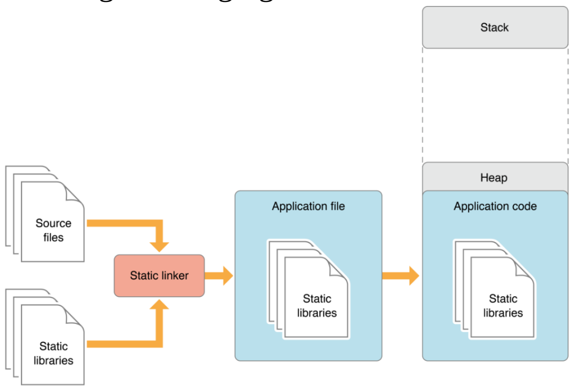
MAKE_DEPEND := $(CC) -M
...
#some time later
CC = gcc
	MAKE_DEPEND = $(CC) -M
	...
	#some time later
	CC = gcc

$(MAKE_DEPEND) expands to:	
<space>-M	ccc -M

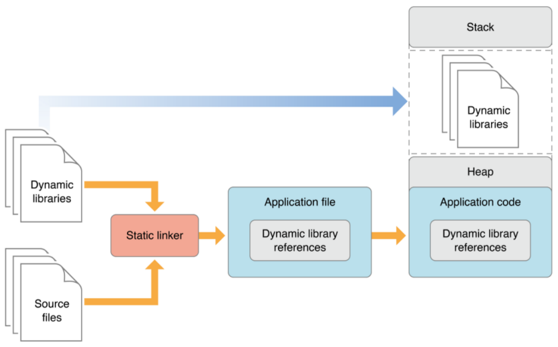
Library: Package of code that is meant to be reused by many programs. Typically has to pieces:

- A header file that defines the functionality the library is exposing to programs using it.
- A pre-compiled binary that contains the implementation of the functionality pre-compiled into machine language, it prevents people from accessing or changing the source code protecting IP.

Static Library: routines that are compiled and linked to the program. A program compiled with a static library would have the functionality of the library as part of the executable. Extension : .a (Unix), .lib (Windows)



Dynamic/ Shared Library: routines that are loaded into the application at the runtime. Extension: .so(Unix) .dll(Windows)



Property	Static Library	Shared Library
Compilation	Recompilation is required for changes in external files	No need to recompile the executable.
Linking Time	Happens as the last step of compilation process	Are added during linking when executable file and libraries are added to the memory
Import/ Mechanism	Are resolved in a caller at compile-time and copied into target application by the linker	Get imported at the time of execution of target program by the OS
Size	Are bigger in size, because external programs are built in the executable file	Are smaller, because there is only one copy of the shared library that is kept in memory
External file changes	Executable file will have to be recompiled if any changes were applied to external files	No need to recompile executable – only the shared library is replaced
Time/ Performance	Takes longer to execute, as loading into memory happens every time while executing	Faster because shared library is already in the memory
Compatibility	Never has compatibility issue, since all code is in one executable module	Programs are dependent on having a compatible library.