STL

- **Containers** — hold objects
- **Iterators** — manipulate STL elements
- **Algorithms** — Perform common manipulations.

- Sequence Containers — Linear Data Structures — vector, dequeue, list.
- Associative — Non Linear — set, multiset, map, multimap.
- Adaptors — stacks, queue, priority_queue.

Iterator Support:

- vector, Dequeue ← Random access
- list, set, multiset, map, multimap ← Bi Directional
- stack (LIFO), queue (FIFO), priority_queue ← None.

Algorithms — Operate on ~~contain~~ container elements only indirectly through iterators.
eg. find(), Sort()

Mutating algos → swap, fill, copy, remove, replace_if, reverse, transform, remove_if, unique, generate, replace

Non-mutating / modifying → equal, find, count, find_first_of, find_if, count_if, find_each, search, find_end.

```
pair<int, string>
make_pair(2, "abc");
```

Discovering Modern C++

C++ is a strongly typed language as every variable has a type that never changes.

Basic / aka Intrinsic types are:
char, short, int, long, long long (very long integers)
unsigned ← unsigned versions of above
signed ← signed " —— "
float, double, long double, bool

signed is default except for "char".
With unsigned, we can store 2x of positive value plus zero.

Char can have −128 to 127 (signed)
or 0 to 255 (unsigned)

Typically float is 32 bit, double is 64, and long double is 80 bits.

Literals can be qualified with a suffix explicity such as
2, 2u, 2l (long), 2ul (unsigned long), 2.0 (double)
2.0f (float), 2.0l (long double)

long is 32 bits on a 32-bit platform
& 64 " —— 64-bit "

In C++ 11 one can enclose value in {} and they won't be lost in assignment.

Narrowing

```
long l = {1234.....123};  // either Error or ok if supported
int it = {3.14};  // error; truncation not allowed
Unsigned u2 = -3;  // ok
"  u3 = {-3};  // error
```

C++

Type modifiers : signed, unsigned
Type Qualifiers : const, volatile, static
Types : char, bool ← 1 byte
wchar_t, short ← 2 bytes
int, long, float ← 4 "
long long, double ← 8 "

<climits> or <limit.h> contain limits of integers such as INT_MIN

<cfloat> or <float.h> has floating type limits.

Copy Initialization

```
int a1 = 0;
```

uniform initialization

```
char e1[8] {"hello"};
char *p1 = new char[8] {"hello"};
```

uniform initialization should be used for user-defined types and copy for primitive type for code clarity

Hiding

when a variable with same name exists in nested scopes, then only one variable is visible. Inner one hides the outer one.

```
int main()
{   int a = 5;      // a#1
    {
        a = 3;      // a#1 is assigned.
        int a;      // a#2
        a = 8;      // a#2 is assigned; a#1 is hidden
    }
}
```

C++ operators

- Arithmetic ++, +, *, % ... logical
- Boolean <=, != , ... &&, ||
- Assignment =, += ...
- Program flow function call, ?:
- memory new, delete
- Access ., →, [], *, ...
- Type handling dynamic_cast, typeid, sizeof, alignof, ...
- Error " throw
- unary +, −, !

LValue ← Tech term for an addressable data item.

```
int i=3, j=3;
int k = ++i + 4;   // i= 4, k=8
int l = j++ + 4;   // j=4, l=7

int j = -i;   // j is -3  ← unary negative operator
unary plus has no effect
```

Range based for loop:

```
using namespace std;
int arr[] = {1, 2, 3, 4, 5};

for (const auto& x : arr)
    cout << x << endl;          // because of & a local copy is not made.

for (const auto& x : {1, 2, 3, 4, 5})
    cout << x << endl;

// int *beg = &arr[0];
// int *end = &arr[4];
// In C++ 11 above 2 can be replaced by
int *beg = std::begin(arr);
int *end = std::end(arr);

while (beg != end) {
    cout << *beg << endl;
}
```

```
int i=3, j=7, k;
k = f(++i) + g(++i) + j;
```

can be interpreted as

$k = f(4) + g(5) + 7$ OR $f(5) + g(4) + 7$

depending on compiler.

So, it is dangerous to modify values in expressions.

Range based for loop C++11

```
int primes[] = {2, 3, 5, ..., 19};
for (int i : primes)
    std::cout << i << " ";
```

main()

```
int main(int argc, char* argV[])
        Same as
int main(int argc, char** argv)
```

Assertions

assert() macro in header <cassert> is used to evaluate an expression and when result is false, throws exception

```
assert(result >= 0);
```

They work only with DEBUG.

Exceptions

With return codes, caller can ignore the return value and program will continue. Not the case with exceptions. With exceptions you can throw the error description too as a string not just the error code. One can throw anything: string, number, user types etc.

malloc / calloc / realloc and free are used by C.

calloc is same as malloc except that it initializes the memory by 0. These methods return a void pointer.

noexcept C++ 11

double sqroot(double x) noexcept { ... }

if an exception is thrown program will terminate.
calling code do not need to check for exceptions from sqroot().

static_assert (11) ← C++ 11 support

if a program error is detected in compile time, error message is emitted & compilation stops.

I/O Streams

streams are not limited to ~~istream~~ screen, keyboards and files. Every class can be used as a stream when derived from istream, ostream OR iostream.

I/O streams are formatted by so called I/O manipulators found in header <iomanip>

stream ops can return an error code or can also be enabled to throw an exception.

new / delete

```
{ int *p = new int;
  delete p; }

{ int *p = new int[10];
  delete [] p; }
```

~~null~~ nullptr C++ 11 new Keyword

```
int * p = nullptr
int * p2{};
```

same as `int *p = 0;` in old C++

new is an operator. It can initialize memory and call constructors.

Make Functions

```
std::unique_ptr<int> p{new int{53}};
```

↑ this can be written with make function as

```
auto p = std::make_unique<int>(5);
auto point = std::make_unique<Point>(3,5);
auto pArr = std::make_unique<int[]>(10);
```

↑ Size of Array

Same applies with shared_ptr.

make_shared

3 strategies to minimize pointer related errors :—

1. Use containers such as vectors / maps
2. Encapsulate dynamic memory management in classes. Allocate / Deallocate in constructor / destructors. This principle is called RAII → Resource Allocation Is Initialization
   ↑ Use of objects to allocate resources so that constructor / destructor handles it.
3. Smart pointers.

Smart Pointers <memory>

c++11 introduced 3 new types → unique_ptr, shared_ptr and weak_ptr. auto_ptr should not be used anymore.

```
unique_ptr<double> dp{new double};
*dp = 7;

double* raw_dp = dp.get();
```

The copy constructor of unique_ptr is deleted

unique_ptr cannot be assigned or copied. They can only be moved.

```
{unique_ptr<double> dp2{move(dp)}, dp3;
dp3 = move(dp2);
```

After move, dp2 shouldn't be used.

ownership moved and dp, dp2 are reset to nullptr. dp3 will release the memory in destructor.

unique_ptr can be passed / moved by reference

dp3 reset if new object needs to be assigned

array version

```
unique_ptr<double[]> da{new double[3]};
```

shared_ptr ← same as unique_ptr but can be copied or assigned. So multiple parties point to same data. Deallocated through reference counter. So more overhead & space used than unique_ptr.

Custom deleter

```
struct free_{
    void operator()(int *p){
        free(p);
        cout << "pointer freed";
    }
};

int main()
{
    std::unique_ptr<int, Free> p{(int*)malloc(4), Free{}};

    *p = 100;
    std::cout << *p << endl;
}
```

When out of scope, this will invoke custom deleter

weak_ptr

shared_ptr has issues with cyclical references. Dangling pointer problem. That is resolved using weak_ptr.
weak_ptr is used along with shared_ptr.

```
shared_ptr<int> sptr;
*sptr= 10;
weak_ptr<int> weak =sptr;
```
↑ get ptr to data without ~~ta~~ taking ownership.

To check if ptr is still valid use lock()

```
if(auto tmp = weak.lock())
 {
   // use weak
 }
else
  {
   //expired   //nullptr.
  }
```

reference_wrapper class C++11

A compromise between pointers & references.

```
double & sqroot (double d)
{  double s= d*d;
   return s;
}
```
↑ stale Reference. Causes runtime errors.

Macros ← Risky. Have no regard to namespace, scopes etc.
C++ provides alternatives like constants, inline and constexpr, templates.

Include guards

header file

```
#ifndef NEWFILE_HEADER_H_
#define "
#include <cmath>
:
#endif
```

Avoids double inclusions.

can be changed to:

```
#pragma once
#include <cmath>
```

DDD Domain Driven Design

Core idea is that developers regularly talk to domain experts about how software components should be named and behave so that resulting s/w is as intuitive as possible.

C++ classes The most important benefit of classes in C++ for us is not the inheritance mechanism but the ability to establish new abstractions and to provide alternative realizations for them.

Defining the behavior of a class in its external interface but not how it is implemented, we establish Abstract Data Type ADT

default and delete keywords

Compiler can be instructed to create a default constructor, copy constructor and assignment operator.

```
class Complex {
public:
    Complex() = default;
    Complex(const Complex&) = default;
};

Complex() = delete;
Complex(const Complex&) = delete
```

↑ The copy constructor won't be available here and therefore objects cannot be copied.

```
void setValue(int value) {
    m_int = value;
}
void setValue(float value) = delete;
```

↑ Now setValue(8.5) will fail. Otherwise implicit conversion to int happens and setValue(int) gets invoked.

constructors

1. default 2. copy 3. Move constructor (C++ 11)

C++ 11 offers Delegating Constructors; these are constructors that call other constructors.

```
class Complex
{
    public:
        Complex(int x, int y) : x{x}, y{y} {}
        Complex(int x) : Complex{x, 0} {}
        Complex() : Complex{0} {}
        ...
};
```

C++11 also lets you set default values for member variables

```
class Complex
{
    ...
    private:
        int x=0, y=0;
};
```

a constructor can be made explicit so it can be used by compiler implicitly to do type conversions

explicit Complex(...)

copy constructor

```
Complex(const Complex& c) { ... }
```

assignment operator

```
Complex& operator=(const Complex& src)
{
    if (this == &src)
        return *this;
    // do assignments etc.
    return *this;
}
```

[Complete Modern C++ / Udemy]

&& ← RValue Reference.
temporaries and Constants binds to && reference.

```
void Print(int &x){
    cout <<x;
}

void Print(int &&x){
    cout <<x;
}

int main(){
    int x=10;
    Print(x);

    Print(3);
}
```

invokes

invokes

Move Constructor

It copies data from source including pointers without first having to allocate new memory. The new object then releases the memory.

```
class MB
{
  private:
    size_t len;
    int *data;
  Public:
    explict MB(size_t length) : len(length),
                                data(new int[length]) {}
    ~MB()
    {
      if(data != nullptr) { delete[] data; }
    }
    MB(const MB& other) : len(other.len),
                          data(new int[other.length])
    {
      std::copy(other.data, other.data+len, data);
    }
    MB& operator=(const MB& other)
    {
      if(this != &other)
      {
        delete[]data;
        // assign & copy other
      }
      return *this;
    }
MB(MB&& other) : data(nullptr), len(0)
{
  data = other.data;   //no new
  len = other.len;
  other.data = nullptr; } source can no longer
  other.len = 0;        } free the object.
}
```

move constructor is invoked when a copy is being returned from a function.

```
void foo()
{ MB mb1;
  // do some ops
  return mb1;
}
```

```
MB& operator = (MB && other)
{
    if (this != &other)
    {
        delete[] data;
        data = other.data;
        len = other.len;
        other.data = nullptr;
        other.len = 0;
    }
    return *this;
}
};
```

} source no longer can be used.

Move constructor only works on Rvalue. You can also call move assignment operator from move constructor and avoid redundant code.

Revised version using move assignment operator

```
MB(MB&& other) : data(nullptr), len(0)
{
    *this = std::move(other);
}
```

↑ preserves the R-value property of the other parameter.

objects are considered expired after std::move()

Destructors

- Never throw exception from a destructor as that would cause runtime error. They are implicitly declared "noexcept".

C++ has 6 methods with default behavior (4 in C++ 03)
- Default, copy & Move constructor
- Copy & Move assignment operator
- Destructor

subscript operator

as in vector object V[5] or v[i] accessing an element directly instead of using .at(i).

```
class vector
{public:
    double& operator[](int i)
    {
        return data[i];
    }
};
```

```
double at(int i)
{
 return data[i];
}
```

const ← makes a method const, ie. no class members can be changed. To override constness for some members, declare them as mutable.

RAII - Resource Acquisition is Initialization - paradigm & idea about tying resources to objects and using the mechanism of object construction & destruction to handle resources automatically in programs. When object goes out of scope, the resource is released automatically.

① why would u want to create a class that should not allow copying?

Ans: For classes that has resources that cannot be copied such as file stream, thread, or handler etc. Such class can have more constructor though.

std::move is useful on non-copiable objects. Its usage on ~~redund~~ primitive types is redundant.

Generic Programming

- A paradigm for maximum applicability while providing correctness. Its main tools are templates.

Templates – create functions & classes that operates on a parametric (generic) types.

Function Template aka Generic function – blueprint to generate a potentially infinite number of functions overloads.

```
template <typename T> T max (T a, T b)
{
    if (a > b)
        return a;
    return b;
}
```

implicit instantiation

Can be called as `max(5, 6)`, OR `max(5.0, 6.0)`

OR `max<float>(5.0, 6.0);`

↑ explicit instantiation

move and forward keywords

move constructor is invoked.

std::move ← transfer resources from one to another object

```
vector x{std::move(w)};
```

↑ x steals data from w and leaves it empty.

Forward References OR aka Universal Reference = T&&

Like move, forward is a pure cast and does not generate a single machine op. It is said: move does not move and forward does not forward. They rather casts their arguments to be moved or forwarded.

In c++ 14, compiler can deduce the return type of a function.

ex.

```
template < typename T , typename U >
  inline auto max (T a, U b)
  {
    return a > b ? a : b;
  }
```

namespaces

```
namespace c1 {
  ...
  namespace c2 {
    ...
    struct cc {
      ...
    };
  }
}

void fun()
{
  using namespace c1::c2::cc
  cc x;
  ...
}
```

(namespace: c1::c2, struct: cc)

This works in a function and preferable. Importing a name into a namespace within header files increases the danger of name conflicts because name remains visible in subsequent files.

when namespaces are too long or nested, we can rename them with a namespace alias :-

```
namespace lname = long_namespace_name;
namespace nested = long_name :: another_name ; // one more.
```

Argument Dependent Lookup (ADL)

Expands the search of function names to the namespaces of their arguments.

ex.

```
void main()
{
    mymath :: matrix A, B, C, D.
    mymath :: Initialize(B); // Qualified.
    Initialize(C); // ADL - via 'C'

    yourmath :: matrix E, F, G;
    Initialize(E); // Error if yourmath ::
                   namespace dont have an
                   Initialize() method
```

operators are also subject to ADL.

Automatic variable type =

```
auto a = 2 * 7.5;
auto j = sqrt(3.7);      } double
auto v = g(x,y,2);       result of function g
```

we can qualify auto with const and reference attributes.

```
auto &       ri = i; // ref on i
const auto & cri = i;
auto &&      ur = g(x,y,2); // fwd reference to
                            // result of g();
```

C++11 = decltype

Uses a function as declaration type.

```
decltype(g(x,y,2)) a = g(x,y,2);
```

↑ usually used in template classes

```
decltype(v1[0] + v2[0])
```

we deduce what type we get when we add elements of both vectors v1 & v2. The two vectors may have different types and the resulting vector yet another one.

The 2 features auto & decltype not only differ in their applications; the type declaration is also different.

auto drops references and const qualifiers and a corresponding auto variable would be a value.

decltype takes the expression type as is.

With advanced generic and metaprogramming, we can greatly benefit from these extremely powerful features.

```
decltype(auto) v= expr;
v is now same type as expr; // ref & const are maintained.
```

↑ same as

```
decltype(expr) v= expr;
```

but with auto it is more readable specially if "expr" is a long expression

Defining types

C++11 has using for that.

Earlier: typedef double da1[10];

now: using da2 = double[10];

better with fn pointers:

⇒ typedef float float_fun1(float, int);

⇒ using float_fun2 = float(float, int)

For function pointers one should use "function" class template in <functional>.

Fn ⇒ double add(double x, double y);

Fn ptr for above can be declared as:-

```
using bin_fn = function<double(double, double)>;
bin_fn f = &add;
f(6.0, 3.0);

vector<bin_fn> fns;
fns.push_back(add);  // address taken implicitly.
```

You can add functions to containers that have diff ret types. Only the args should be same. Even lambda fns such as:

```
fns.push_back([](double x, double y){return x/y;});
for(auto &f : fns)
    cout << f(6,3);
```

This will call each entry in the fns vector.

reference_wrapper

You can create a container of references (avoiding use of pointers) using reference_wrapper keyword.

eg

```
vector < reference_wrapper <vector<int>> > VV;
map <int, reference_wrapper<vector<int>> > mv;
vector<int> V1 = {2,3,4};
```

this can also be declared as =

```
map <int, decltype(ref(V1))> mv;
```

```
mv[0] = ref(V1);
```

⇐ Error as no default constructor exists that can handle this.

Instead use

```
mv.insert(make_pair(8, ref(V1)));
```

or

```
mv.emplace(                    )
```

emplace does in-place insertion and is faster. Whereas insert inserts an element and then reorders the map.

Time library | Chrono

<chrono>

```
time_point<system_clock> now = system_clock::
                                    now();
~~cout << ctime(&~~
time_t now_time = system_clock::to_time_t(now);
cout << ctime(&(now_time));
```

⇒ Wed Feb 11 22:31:31 2018

Determine time spent

```
time_point<steady_clock> start = steady_clock::now();
    funx();
auto end = steady_clock::now();
cout << "took " << (end-start).count() << "ticks"
```

How to convert ticks to micro/nanoseconds

```
duration_cast<microseconds>(end-start).count();
duration_cast<nanoseconds>(end-start).count();
```

functors aka Function Objects

a C++ class that acts as a function via overloading of operator().

ex.

```
class psc_f
{
public:
    psc_f(double alpha) : alpha(alpha) {}
    double operator()(double x) const
    {
        return sin(alpha * x) + cos(x);
    }
private:
    double alpha;
};

template <typename F, typename T>
T inline fin_diff(F f, const T& x,
                  const T& h)
{ return (f(x+h) - f(x))/h;
}

int main()
{
    psc_f psc_0(0.01);
    cout << fin_diff(psc_0, 1, 0.001);
    cout << fin_diff(psc_f(2.0), 1, 0.001);
}
```

Pmath_gmm<int, vector<int>, std::greater<int>> g;

LAMBDA λ

C++11 introduced λ expr., that are simply shorthand for a functor.

The λ expr doesn't only define a functor but immediately creates an object thereof.

```
fin_diff ( [] (double x) { return sin(x) + cos(x); },
                                    1, 0.001);
```

functor on the fly.

return types are usually deduced. In case it can not or we want to be explicit, we can provide it.

```
[] (double x) -> double { return sin(x) +
                                cos(x); }
```

which local variables can be accessed inside λ — arg type — return type.

CAPTURE

If we have to use an external or out of scope variable in λ expr, we have to capture it first

ex.
```
double phi=2.5, xi=0.2;
auto sin2 = [phi, xi] (double x)
    { return sin(phi*x)
        + cos(x) * xi; };
```

capture

can access them directly otherwise.

Salar Afhar
Sean Sallee

One cannot modify the captured values inside a λ.
To do that use mutable.

```
auto l_mut = [phi](double x) mutable {phi+=0.6;
                                      return phi;};
```

functor class equivalent will be

```
struct l_mut_f
{
  double operator()(double x);
  // ...
  double phi, xi;
};
```

Capture by Reference

```
double phi = 2.5, xi = 0.2;
auto pxr = [&phi, &xi](double x) {
                return sin(phi*x) + cos(x)*xi;
              };
phi = 3.5, xi = 1.2
a = fin-diff(pxr, 1, 0.001);
      ↑ uses phi = 3.5, and xi = 1.2
```

values at the time of fn call are used — not those when λ was created.

(04:00:47)

Generic Lambdas

```
template <typename C>
void reverse_sort(C& c)
{
  sort(begin(c), end(c),
       [](auto x, auto y) { return x>y; });
```

Range based for

```
vector<int> v = {3, 4, 7, 9};
for(int i=0; i<v.size(); i++)
    v[i] *= 2;
```

or

```
for(auto& x : v)
    x *= 2;
```

+4

12:34:54

```
template<typename T, int size, typename Callback>
void forEach(T(&arr)[size], Callback operation) {
    for(int i=0; i<size-1; ++i) {
        operation(arr[i]);
    }
```

```
forEach(arr, [&, offset](auto &x) {
        sum += x;
    });
```

or

```
forEach(arr, [](auto x) { std::cout<<x<<" "; });
```

Callback ops.

STL

unordered_map
unordered containers can be used in same manner as their ordered counterparts.

Algorithms

General purpose algos are defined in the header <algorithm> and for numbers (searching) in <numeric>

some algos are :-

find(), find_if(), copy(), unique()

unique ← removes duplicate entries in a sequence.

sort()

numeric algos

accumulate(), partial_sum, adjacent_difference,

tuple from <tuple>

a tuple is usually used to return more than one result type. Instead of returning via fn args as references, we can return a tuple.

```
tuple<matrix, vector> fn (const matrix & A)
{
    matrix LU(A);
    vector P(n);
    // do something
    return tuple<matrix, vector>(LU, P);
                    OR
    return make_tuple(LU, P);
}
```

auto t= make_tuple (LU, P, 7.3, 9, LU*P);

caller of the function will extract the values from the tuple using get() :-

~~tuple<matrix~~

```
auto t = fn(A);
auto LU = get<0>(t);
auto p = get<1>(t);
```

// types are deduced

Same as

```
tuple<matrix, vector> t = fn(A);
matrix LU = get<0>(t);
vector P = get<1>(t);
```

if unambiguous, they can also be accessed by type such as :-

```
auto t = fn(A);
auto LU = get<matrix>(t);
auto p = get<vector>(t);
```

or use **tie**

```
matrix LU;
vector P;
tie(LU, P) = fn(A);
```

Meta Programs – programs on programs. In C++ we can write programs that compute during compilation or transform themselves. Can be achieved using template meta functions or entity with "constexpr".

constexpr was introduced in C++ 11.

ex.

```
constexpr long fibonacci(long n) {
    return n<2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}

template <typename T>
constexpr T square(T x)
{
  return x*x;
}
```

constexpr : one feature is their usability at compile time as well as run time.

```
cout << R"(This is not a \tab)";
```

ignores special characters.

OOPS

Basic principles of OOP related to C++ are :-

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Late Binding.

⇒ C++ has 6 methods with default behaviour
(4 in C++03)

1. Default constructor
2. Copy "
3. Move " C++11 or higher
4. Copy assignment
5. Move assignment C++11 or higher.
6. Destructor.

If a class has ownership (pointers, resources etc) you must provide these 5. This is called = Rule of 5

Rule of 0 ⇒ if a class doesn't have any pointers etc. do not create any of these 5. Compiler will generate these for us.

```
class C
{
public:
private:
    type1 var1;
    type2 var2;
    typen varn;
public:
default → C() : var1(), var2(), ... varn()
    {}

copy constructor → C(const C& that) :
        var1(that.var1),
        var2(that.var2),
        varn(that.varn)
    {}
```

```
C ( C&& that )
  : var1 ( std::move ( that.var1 ) ),
    var2 ( std::move ( that.var2 ) ),
    //...
    varn ( std::move ( that.varn ) )
{}
```

move constructor

```
C& operator=( const C& that )
{
  var1 = that.var1;
  var2 = that.var2;
  //...
  varn = that.varn;
  return *this;
}
```

Copy assignment

```
C& operator=( C&& that )
{
  var1 = std::move ( that.var1 );
  var2 = std::move ( that.var2 );
  //...
  varn = std::move ( that.varn );
  return *this;
}
```

move assignment

```
~C()
{
  varn.~typen();
  var2.~type2();
  var1.~type1();
}
};
```

Destructor

Controlling the generation of default methods:
C++11 provides 2 declarators :- default and delete :

ex

```
class move_only()
{
  public:
    move_only() = default;
    move_only(const move_only&) = delete;
    move_only(move_only&&) = default;
    move_only& operator=(const move_only&)
                                  =delete;
    move_only& operator=(move_only&&)
                                  =default;
    ~move_only() = default;
};
```

Op overloading

Integer Sum1 = a+5; → resolves as a.operator+(5);

Integer Sum2 = 5+a; ← fails as 5 is a value

to make it work one can write a global operator+ as:

```
Integer operator+(int x, const Integer& y) {
        Integer temp;
        temp.x = x + y.getValue();
        return temp;
}
```

~~Sum1~~ cout << sum1 << endl;

How to make it work?
use global overloading op.

```
std::ostream& operator<<(std::ostream &out,
                         const Integer &a) {
        out << a.getValue();
        return out;
}
```

```
std::istream& operator>>(std::istream &input,
                         Integer &a) {
        int x;
        input >> x;
        a.setValue(x);
        return input;
}
Integer a(1);
std::cin >> a;
```

friend

using friend we can make a function as friend of a class and it can access its private & public members.

we can also make a class as friend of another class.

overloading → operator

```
class IntPtr {
        Integer *m_p;
public:
    IntPtr(Integer *p): m_p(p) {}
    ~IntPtr() {
        delete m_p;
    }
    Integer *operator->() {
        return m_p;
    }
};
```

asterisk op.

```
Integer &operator*() {
    return *m_p;
}
```

```
void main() {
    IntPtr p = new Integer;
    p->setValue(3);
```

⇒ m_p->setValue

```
    (*p)->setValue(3);
}
```

↖ asterisk op is invoked.

- Operator functions should be non-static, except for new and delete.
- one argument should be user-defined type.
- Global overload if 1st arg is primitive type.
- These cannot be overloaded ⇒ . , ?: , .* , sizeof

casts

① static_cast = checks if cast is valid or not

```
float f = static_cast<float>(a) ; //a is integer
```

②
```
char *p = (char*) &a; // int a;
```
↖ works

```
char *p = static_cast<char*>(&a);
```
↑ this fails.

reinterpret_cast will work. it is same as C style cast except that C-style cast discard the qualifiers such as const too.

```
char *p = reinterpret_cast<char*>(&a);
```

③
```
const int x=1;
int *p = const_cast<int*>(&x);
```
↑ cast away constness

④ dynamic_cast

Type conversion operator ⇒ To convert user defined object to primitive type

Operator <type>()

operator int()

```
Integer a1(5);
int x = a1;
        ↑ Type conversion op.
```

```
Integer::operator int()
{
    return *mptInt;
}
```

OR

```
int x = static_cast<int>(a1);
```

C++11 allows explicit keyword with type conversion op. So users have to use <static_cast> for conversions.

45-50

```
#include <sstream>
std::stringstream ss;          int a {5}, b{6};
                               int sum = a+b;
ss << "sum of" << a << " & " << b << " is: "
                                  << sum << std::endl;
std::string s = ss.str();
```

So stringstream can be used to conveniently concatenate primitive types / or convert/parse numbers:

```
int sum = 10;
auto ssum = std::to_string(sum);

std::string data = "12  89  21";
std::stringstream ss;
ss.str(data);

while (ss >> a) {
        std::cout << a << std::endl;
}
```

This will extract numbers from the stringstream.

```
int x = std::stoi("54");   ← convert string to integer.
```

Threads

```
# include <thread>
  void Process()
  {
      std::this_thread::sleep_for(1s);
                          ↑
              std::chrono::seconds(1);
  }

  int main() {
      std::thread t1(Process);
      HANDLE handle = t1.native_handle();
      SetThreadDescription(handle, L"My Thread");
      int cores = std::thread::hardware_concurrency();

      t1.join();
      return 0;
  }
```

Task

```
std::future<int> result = std::async(std::launch::async,
                              Operation, 10);
if(result.valid()) {
    auto status = result.wait_for(60s);
    switch (status)                 wait()
    {
        std::future_status::ready:
                  break;
        std::  ________  ::timeout:
    }
auto sum = result.get(); ← control will wait
                           till Operation is finished
```

.NET C++

Boxing / Pinning ⇔ Mixing C++, .NET gets easier.

Boxing : passing unmanaged data to a method expecting managed data cause compile time errors.

Boxing is used to pass this unmanaged data. It creates a reference out of data & then passes that.

Eg.

```
int i;
System::Console::write(__box(i));
```

Pinning :- Enables you to pass unmanaged data to C++ methods that expects pointers.

Managed refernces are internal pointers that can be moved by GC, so pinning is required.

```
__gc class sample {
public:
    int a;
    int b;
};
```

C++

```
Equalize(int *pa,
         int *pb);
```

```
sample *s = new Sample();
int __pin * pa = &(s->a);
int __pin* pb = &(s->b);
Equalize(pa, pb);
pa=pb=0; ← unpin the s->a, s->b.
```

Properties in .NET

```
__gc class Employee {
    private:
        int salary;
    public:
        __property int get_salary() { return salary; }
        __property void set_salary(int s)
            { salary = s; }
};
```

use them like this:

```
Employee e;
e->salary = 1000;
e->salary = e->salary + 500;
```

Language Extensions = keywords that starts with __ (double underscore)

Note:

code written in managed C++ can be used in VB.NET / C#.

Managed C++ can use C# code with ease.

Unmanaged C++ from C#

- Write a wrapper class in C++.NET.
  or
- Convert unmanaged code in a DLL and access it through PInvoke.
  or
- Convert it into COM and access it through Interop.

Mixed DLLs

- Do not write managed code in DllMain or the program can hang because of Loader Lock.
- from 1.1 .NET onwards the DLLs are ~~graduated~~ generated with /NOENTRY that suppresses the DllMain() function.
  Linker → Advanced → NoEntry Point.
- In the absence of DllMain(), you need to add a initializer method in DLL that should call _crt_dll_initialize().
  To clean you should call _crt_dll_terminate().
  The consumer exe of your DLL should call the initializer method.

{ Not needed in new version of MSVS }

ILDASM – Intermediate Language DisAssembler,
use it to examine the contents of an assembly.

/clr option – Remove from a cpp file to force it to get compiled as non-managed.

Using unmanaged DLL from Mangaed c++

– Use same way as in a c++ project

char* and String

They dont replace each other directly.
This is how to convert.

```
String *s = new String("Hello"); //C#

char *pchar = (char*) Marshal::
        StringToHGlobalAnsi(s).ToPointer();

Log(pChar);
↑ C++ function
```

Unmanaged DLL from C#

- Imported methods must be ~~export ss~~ "static extern".
- Use DllImport to import the DLL and declare the members in C# file before main()
- Call the functions.

Note:- Better plan is to create a manged library in C++ and write wrappers.

COM components from Managed C++

- Through RCW (Runtime callable wrapper) aka Interop Assembly
- The wrapper internally calls the COM methods, marshals params & translates HRESULT error codes to ~~ex~~ .NET exceptions
- Visual Studio can generate RCW if vendor didn't supply one.

Managed Code as a COM

- Unmanaged code can access .NET component using CCW (Com Callable Wrapper)
- There are some conditions that .NET code must meet to be accessed via CCW.
  - → must implement atleast one interface
  - → Must generate typelib.
  - → Should be put in GAC.
  - → Information to enable COM to find the component must be added to registry.

tlbexp ManagedAss.dll.

All assemblies in GAC must have a strong name (ie version number, localization info, public key, digital sign)

sn. -k mylibrary.snk

Add to GAC using "gacutil" command.

Final step is making registry entries that points to CCW, mscoree.dll.
use regasm for that.—

regasm mylibrary.dll.

mscoree.dll ← The execution engine of the .NET runtime (Common Object Runtime ~~Code~~) COR

#import "mylib.tlb" exclude ("allocator<void>", "type2", "type3" ...)

To exclude fatal error code C1196.

128	64	32	16	8	4	2	1

5 = 101, 7 = 111

10/2 11/3

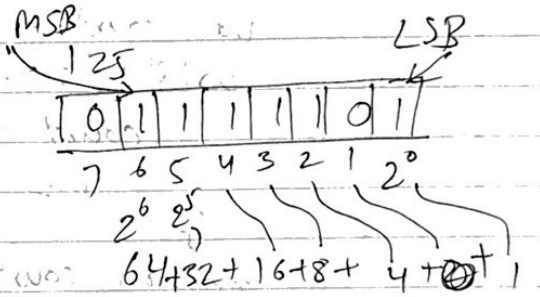
Bitwise Operations

& = AND | = OR ~ = One's complement / NOT (turn a bit on/off)
^ = XOR << = Left shift >> = Right Shift.
mask particular part of byte.

Decimal Number system =
Repeatedly do the modulo operation until n becomes 0
int n = 1256 mods are 6, 5, 2, 1
n/10 becomes 125, 12, 1

Remainder

10 |	1256
10 |	125 → 6
10 |	12 → 5
10 |	1 → 2
	0 → 1



An odd number always has 1 as LSB = 125

$100101 = 2^5 + 0\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 2^0$
(5 4 3 2 1 0)
$= 32 + 4 + 1 = 37$

Advantages of Bitwise ops

→ Faster, directly supported by processor.

2 |	125
2 |	64 → 1
2 |	32 → 0
2 |	16 → 0
2 |	8 →

Remainder

2 |	125
2 |	62 → 1
2 |	31 → 0
2 |	15 → 1
2 |	7 → 1
2 |	3 → 1
2 |	1 → 1
	0 → 1

1111101

Problem: count number of bits

n = 125 ; output = 7 bits. (1 1 1 1 1 0 1)

n = n >> 1; is same as n = n/2

shift by 1 bit to right

```
int counter (int n)
{
    int count = 0;
    while (n > 0) {
        count ++;
        n = n >> 1;
    }
    return count;
}
```

stack method

```
convertDecimal (int n)
{
stack<int> st;
while (n > 0) {
    int rem = n % 2;
    st.push(rem);
    n = n >> 1;
}
while (!st.empty())
{
    cout << st.pop();
}
```

n = n << 1; ← multiplies by 2

← same as n = n * 2;

```
int bitsCounter (int n)
{
    int count = 0;
    while ((1 << count) <= n) {
        count ++;
    }
    return count;
}
```

Shift left till we find a number bigger than n.

a	b	AND &	OR |	XOR ^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

~ Inverse operator it inverses all the bits. Also known as 1's complement.

5 = 101

~5 = 010

~000101

⇒111010

>> ← Right shift op

<< ← Left shift op.

a = 12 0 1 1 0 0

a = 10 0 1 0 1 0

a & b 0 1 0 0 0 = 8

2^3

Count set(1) bits of a number.

n = 125 = 1 1 1 1 1 0 1 = 6 set bits

one way is to keep diving n by 2 and check the remainder. if it is 1 increase the counter.

Using & and >>

```
int helper(int n){
    int count = 0;
    while(n > 0){
        if((n & 1) == 1)
            count++;
        n = n >> 1;
    }
```

count += (n & 1);

n & 000001 → number 1

Binary to Decimal

$$101 \Rightarrow 1\times2^2 + 0\times2^1 + 1\times2^0$$

Power $\Rightarrow 2\ 1\ 0$

$$\Rightarrow 4 + 0 + 1$$

$$\Rightarrow 5$$

Decimal to Binary

2	14	
2	7	0
2	3	1
2	1	1
	0	1 ↑

$\Rightarrow 1110$

Addition

```
  1 1
  1 0 1   (5)
+ 1 1 1   (7)
-------
1 1 0 0  (12)
```

$2 \Rightarrow 10$ in binary

$3 \Rightarrow 11$

Minus

$9 - 7 \Rightarrow 9 + (-7)$

There is no minus ops in bitwise ops
One has to find -ve inverse and add it to a number.

$a - b \Rightarrow a + (-b)$

This is 2's complement

$\Rightarrow a + x$

$$\boxed{x = \sim b + 1}$$

inverse the bits and add 1

Negative numbers are stored as 2's complement
if MSB is 0, it is +ve Number
if MSB is 1, it is −ve.

$\sim x = 2^{32} - x;$

$12 - 5 = 12 + (-5)$
$= 12 + \underbrace{\text{Negative}}_{\text{Inverse}}$ } 2's complement
① Invert all bits
② Add ONE

$5 = 101$ $\quad \sim x + 1$

$\sim x$ Inverse 010
$+1$ Add 1 001
$\sim x + 1 = 111011$

000000101
111111011

$12 = 1100$
$+5 = 0101$
$-5 = 1011$

$\begin{array}{r} 1111 \\ 00000\,1100 \\ +\,11111\,1011 \\ \hline 1\,0\cdots 00\,0111 \end{array}$

7 ← Discarded carry.

$a = a >> 1$ ← Divide by 2
$a = a << 1$ ← Multiply by 2

ODD or EVEN

```
if(a%2==0)
    cout<<"even";
else
    cout<<"odd";
```

The LSB is 1 for odd and 0 for even

```
if(a&1==0)
    cout<<"even";
else
    cout<<"odd";
```

$\begin{array}{rr} 111 & 110 \\ \&\,001 & \&\,001 \\ \hline 001 & 000 \end{array}$

```
int switchSign(int n)
{
  return ~n+1;
}
```

Swap 2 Numbers

```
int a=5;
int b=7;

int temp=a;
a=b;
b=a;
```

Swap using XOR

```
a=a^b;
b=a^b;   // a^b^b
a=a^b;   // a^b^a
```

0

XOR

$a \wedge a = 0$

$0 \wedge a = a$

$a \wedge b \wedge b = a$

Check I^{th} bit

`1<<i`

	8	7	6	5	4	3	2	1	0
n =	1	0	0	1	1	0	1	0	1
mask =	0	0	0	1	0	0	0	0	← shift 1 by 5
n & mask =	0	0	0	1	0	0	0	0	`1<<5`

if non zero then I^{th} (5^{th}) bit is set
or else not set (0)

Toggle the bit

n ^ mask =	1	0	0	0	1	0	1	0	1

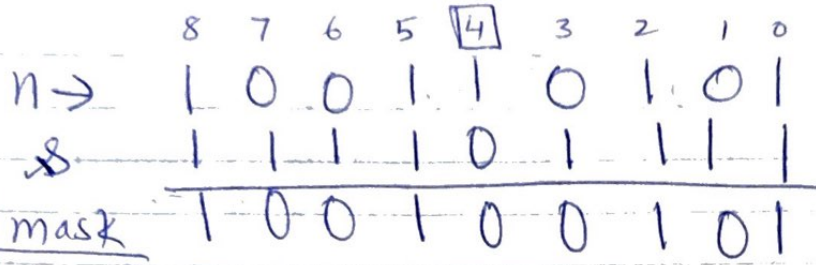
XOR toggles the I^{th} bit

Flips a bit

Set I^{th} bit

	1	0	0	1	0
n | mask =	0	0	1	0	1
OR →	1	0	1	1	1

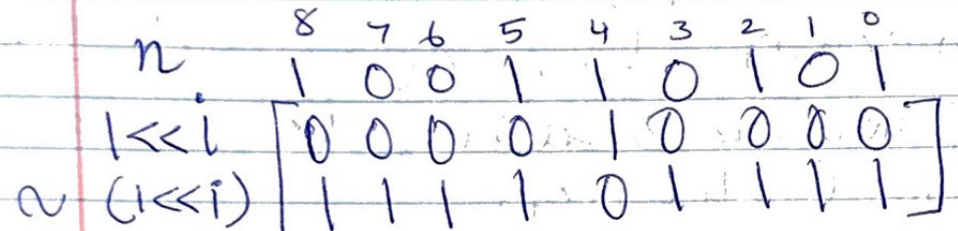
Clears i^{th} bit — Toggle. — Unset a flag




```
unsetFlag(int flag)
{
    newValue &= ~flag;
}
```

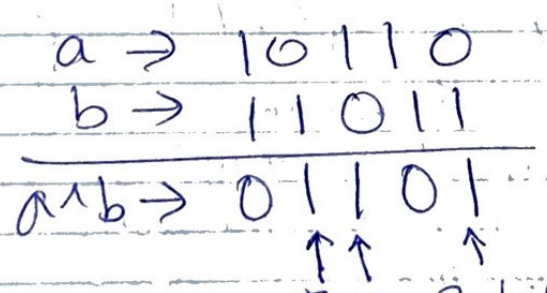
Steps

`1<<i`




```
mask = ~(1<<i)
int result = n & mask;   ← clears the i^th bit
```

Find number of bits to change to convert a to b.



$n \& (n-1)$ changes LSB to 0

LSB — Least Significant Set Bit

1101

$n = n \& (n-1)$ → 1100

$n = n \& (n-1)$ → 1000

$n = n \& (n-1)$ → 0000

So if there are 3 set bits, the number will become 0 after 3 ops.

If we have to check if a given number is in power of 2, simply $n \& (n-1) == 0$ will tell

XOR

$5 \wedge 5 = 0$

$0 \wedge 5 = 5$

101
101
000

Find the only non-repeating element.

$a = [5, 4, 1, 4, \boxed{3}, 5, 1]$

If we XOR all the numbers from this set, the value will come out as 3. The non repeating number!

	XOR
0 0	0
0 1	1
1 0	1
0 1	0

Interview Bit.com
Geeksforgeeks.

Find 2 non repeating element.

$a = [5, 4, 1, 4, 3, 5, 1, 2]$

XOR all numbers
You will get result = $3 \wedge 2$

$3 \wedge 2 = 0\ 1$

1) Divide the set into 2 where LSB of one bucket is 1 & another is 0.

(5, 1, 3, 5, 1) (4, 4, 2)

$3 \wedge 2 \wedge 5 \wedge 1 \wedge 3 \wedge 5 \wedge 1$

$a = 2$

$3 \wedge 2 \wedge 4 \wedge 4 \wedge 2$

$b = 3$

Find only non-repeating element where others repeat THRICE

$a = [2, 2, 1, 5, 1, 1, 2]$

Count of bits

...		1	3	4
32 ...	3	2	1	0

Same approach will work for K where count % K

Answer ⇒ 1 0 1 ⇒ 5

Look at each bit count and we count % 3 to get bits for non-repeating

Two number x & y have opposite signs if

$$x \wedge y < 0$$

MSB of x or y will be 1 and 0 $\quad 1 \wedge 0 = 1$

For -ve numbers MSB is 1

Sum of n natural numbers is $= \frac{n(n+1)}{2}$

→ logical Right shift = >>>

→ Arithmetic Right shift = >>

available in Java, Javascript & F# only

Preserves the number sign if a number is written in 2's complement.

assume 1011 >>1 → 1101

MSB ↑ Sign bit

↑ Signed bit is preserved.

10000000 01100000 >> 4

will give

11111000 00000110

Find bit length of a given no.

```
int bitslength (int n)
{
    int ctr=0;
    while((1<<ctr)<=n)
        ctr++;
    return ctr;
}
```

RSA Example

Primes P,Q	7, 19
Product = N	133
Totient = T	108
Public key = E	29
Pvt key = D	41

Generate Keys:
- select 2 Primes P, Q
- Calc Product P*Q
- Calc Totient (P-1)*(Q-1)
- Select Public key E such as
  - Must Be prime
  - Less than Totient
  - Must NOT be a factor of Totient
- select a Private key D so that
  - $DE \% T = 1$
    (% = mod)

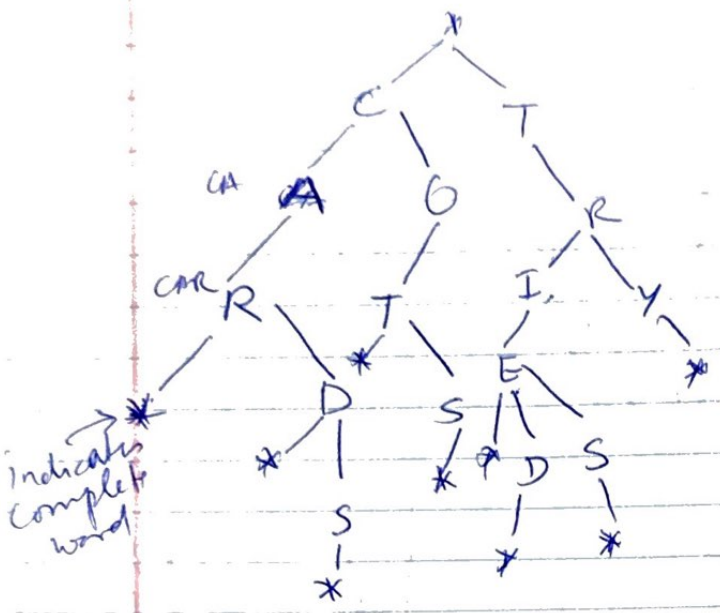
Encryption:

Message ^ E * MOD N = Ciphered Text

Decryption:

CipheredText ^ D * MOD N = Message

TRIE

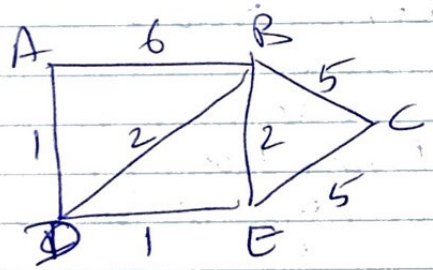


data structure that is type of atree used for dictionary, words, language etc

My Dictionary

CAR
CARD
CARDS
COT
COTS
TRIE
TRIED
TRIES
TRY

Dijkstra ← shortest path calc from A



Vertex	shortest dist from A	Prev vertex
A	0	
B	~~∞~~ ~~6~~ 3	~~A~~ D
C	~~∞~~ 7	~~D~~ E
D	~~∞~~ 1	A
E	~~∞~~ 2	D

Lower vicinity

{ Visited [A, D, E, B, C]
unvisited []

stand with 2 dicts - visited & unvisited.

BFS

- start with A
- Examine it's unvisited neighbors
- Calc dist of each neighbor from start vertex
- update the shortest distance & prev vertex
- Add current vertex to visited.