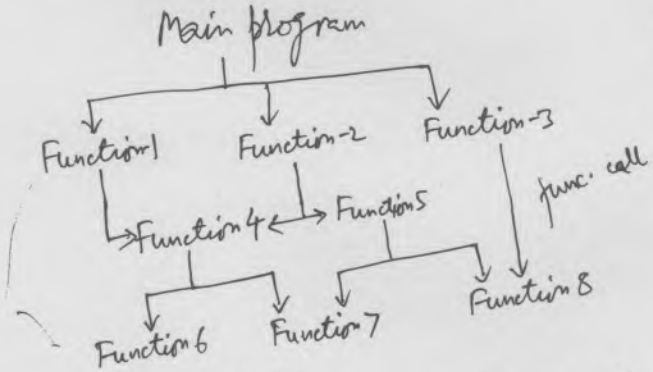
introduction to C++

machine language
↓
assembly language
↓
(functional) Procedure-oriented language e.g., C ↓ better C
↓
object-oriented language e.g., C++

= due to increasing complexity in programming
C++ allows programs to be structured for
- clarity
- extensibility
- maintenance without loss of efficiency

C
//

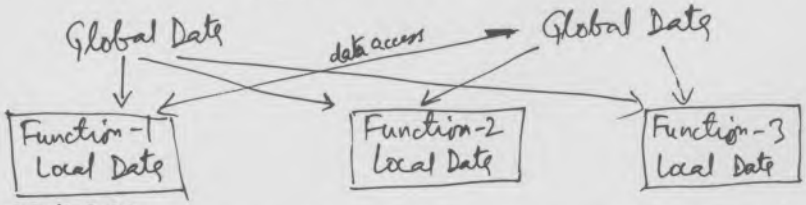
Procedure (functional)-oriented Programming



had poor data structure, so programmers were often unable to pass all data with out of functions. Functions were forced to refer to large blocks of memory. As a result, when the data become corrupted, it was difficult to determine the function that caused it.

- Functions are basic building blocks
- divide program/problem into several smaller problems, each of which can be solved independently by its own function.

Data Function relationship in procedure-oriented language

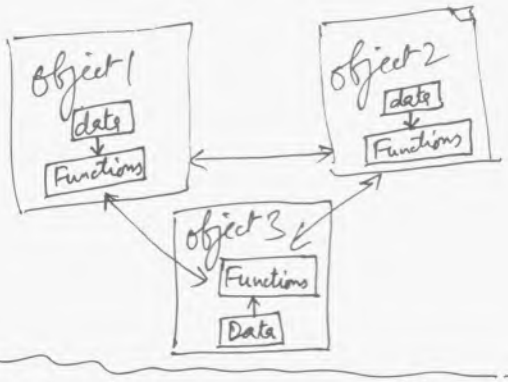


in large program,
- difficult to identify data accessibility among functions and data
- insecure global data

Object-oriented Programming –

- divide (break down) problem into subgroups of related parts that take into account both code and data related to each group.
- combines the best ideas of structured programming with several new concepts that simplifies the task of programming.

OO paradigm =



Procedure-oriented prog.	object-oriented prog.
– prime focus on the function and procedures that operate on data	– more emphasis on the data
– divided into functions	– divided into objects
– data and functions are separate entities	– data and functions are integral entity.
– data move freely around the systems from one function to another (via. function call)	– data is hidden from external functions so, can not be accessed by external functions (secure data)
– programs design follows "top-down approach"	– program design follows "bottom up approach"

= C++ ??

- better (advanced) C, so most of concepts of C also apply to C++.
- Stroustrup confined Object-oriented features of Simula 67 and retained the power and elegance of C, so, it is an extension of C with class construct feature of simula67

= features (characteristics)

- OO programming (object-based programming allows programmers to design applications from a point of view more like a communication between objects that on a structured sequence of code)
- reusability of code
- portability
- Brevity (code written in C++ is very short in comparison with other languages, since the use of special characters is preferred before key word, saving effort.
- modular programming (for project management, for saving time in recompiling the complete application, for linking C++ code with other code in C or Assembler or other languages.
- C competibility
- speed (due to reduced size of code, due to duality as high level language and low-level language)

= Structure of C++ programs

```
#include <iostream>  // include statement
void main()
{
std::cout<<"welcome to All of You";
}
```

a normal C++ program would contain four parts ~~as shown is~~

= steps in execution of C++

(source code) program creation (in text editor)

↓

(object code) program compilation

↓

linking program with C++ Library functions etc.

↓

program execution

= Commands for executing a C program in Unix OS

ed or vi ⟶ text editor
ed filename ⟶ creating a file
CC filename ⟶ compile and link the program
.c or .cc ⟶ extension of program file
a.out ⟶ executing program

.c/.cc source code
↓ CC command
a.out object code

C++ Tokens (smallest individual units)

= C++ is case-sensitive

5

- keywords
- identifiers
- constants
- strings
- Operators

keywords

asm
auto
break
case
catch
char, 8
class
const
continue
default
delete
do
double, 64
else
enum
extern
float, 32
for
friend
goto
if
inline
int, 16
long
new
operator
private
protected
public
register
return
short
signed
sizeof
static
struct
switch
template
this
throw
try
typedef
union
unsigned
virtual
void
volatile
while

identifiers

names

neither spaces nor marked letters can be part of an identifier.

only letters, digits, and underline are valid

ex.. id
name
dept_code

constants

numeric constants
- integer constant
- Real const.

character constants
- single char. const.
- string const.

Operators

\+ − * / %

variable = expression;

relational operators (compare): < <= > >= == !=

incr. ++ decr. −− [++i, i++]

cond. op. (.≥.) ? : :

logical operator (combine): && || !

* Basic data types *

C++ Data Types

- Built-in
  - integral
    - int
    - char
  - void
  - Floating point
    - float
    - double
- user-defined type
  - structure
  - union
  - class
  - Enumeration
- derived Type
  - array
  - function
  - pointer

* variable (named memory location) ⇒ to hold a value

- variable names
  - case-sensitive
  - consists of alphabets, digits, underscores
  - no commas, no spaces, no special symbols, no keywords
  - first character must be alphabet.

= Special operators in C++ =

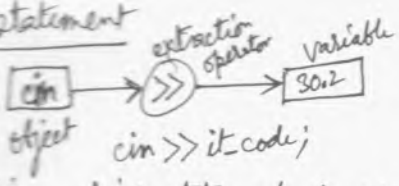
Scope resolution operator		::
pointer-to-member declarator		::*
" " " operator		->*
memory release operator		delete
Line feed	"	endl
memory allocation	"	new
Field width	"	setw
comma	"	,
sizeof	"	sizeof
pointer	"	*, &
new	"	new
delete	"	delete

* Type casting in C++ *

Type name (expression);

ex. a = float (speed);

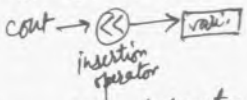
Input statement



cin >> it_code;

Output statement

cout << "Hi!";



decision making statement ⇒ if , Switch, Conditional operator, goto

if (testexpr.) statement;	if..else	Nested if..else	Else..if

Switch (expression) → test only for equality
{ case : Const. value
- - -
- -
}

for (init exp1 ; condition ; exp2)
{ - - -
}

while (condition) { - - - }	do { - - - } while (condition);

break; (can be used to force immediate termination of a loop)

```
- - -
for(t=0; t<100; t++)
  { cout << "\n t=" << t;
      if(t==10) break;
  }
-}-
```

continue; (used to force next iteration of loop (transfer to the beginning of loop again) and skipping any code in between.

```
- - -
{ - if(*str!=' ') continue;
    space++;
  }
```

goto label;
- - -
label: (variable)

```
Begin:
- -
  goto end;
goto begin;
end:;
```

7

Array: int sal[5]; (0: 2500, 1: 2501, 2: 2600, 3: 2700)

sal[0] = 2500;

2D: int Arr[2][2] ⇒ Array of Array

ROW0	ROW1	ROW2
4 5 6 9	1 3 5 7	89 10 6 3

initialization of arrays:

int a[2][2] = {{0,1},{3,2}};
= {0,1,3,2};

strings ⇒ Character array

character array: char n[] = {'D','O','V','E','\0'};
= {"DOVE"};

char. array of string: D | O | V | E | \0 → null character

```
int sq(int n)
{ int r;
  r = n*n;
  return(r);
}
```

Function: return_type function_name(type parameter_list)
{ - - ;
- - ;
}

✓ declaration: return func(); prototype e.g. int square(int num);

✓ definition: int f(){- -}

✓ function call: f(); disp(2,3); x(k); y = f(2);

prototype: return_type function_name(type parameter_list);

* function can pass parameters *

call by reference

```
int main(){- - -
  p(i,j);
}
function call → void p(int &i, int &j)   (alias arguments)
{ - -
}
```

call by value

```
int main(){ - -
  - -
  p(1,2);
  - -
}
```

function call → p(i,j){- - } — creates a new set of variables, p(k,l); copies the arguments values into them; compute

inline function: inline function1(float i){- - -
}

- (expanded in line when it is invoked)
- compiler replaces function call when it is invoked.
- faster

```
int main(){ -
  float j;
  - - -
  function1(i);
}
```

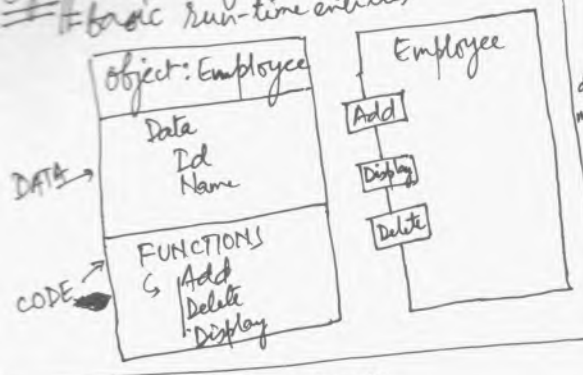
* string functions =

- strlen = length of string
- strlwr = convert string to lowercase
- strupr = " " uppercase
- strcat = appends one string at the end of another
- strcpy = copies a string into another
- strcmp = compares two strings
- strdup = string duplication
- strchr = finds first occurrence of a given character in string
- strrchr = " last " " " " "
- strstr = finds first occurrence of a string in another string
- strset = sets all character of a string to a given character
- strrev = reverse a string

Object-oriented programming concepts

objects, classes, data Abstraction, Data Encapsulation, Inheritance, Polymorphism, Data Binding, Message passing

Object = instance of class
= basic run-time entities



Class

```
class Class_name
{
private:                    → by default
    member variable declarations;
    function    "
public:
    member variable decl.;
    function decl.;
protected:
    .
    .
};
```

data member ← member variable

Class specification

- class Declaration — describes the type and scope of its members
- class function definitions — describes how the class functions are implemented

Accessing class member:

```
Objectname.functionname(args);
x.getdata(20,5.7);
```

Member function definition

- outside the class definition
  - with identity label
  - scope resolution operator ::
- inside the class definition

```
class Book
{ int x;
  .
  .
public:
  void inputData(int x, int y);
  void outputData(void)
  {
    .
    .
  }
};
```

= member functions
- can access private data of belonging class
- can call other member function without (.) dot operator.

```
class Books
{
  .
  .
public:
  void getdata(int x, int z);
  .
  .
};
inline void books::getdata(int x, int z)
{
  .
  .
}
```

making outside function inline

Private member function =
- can only be called by other member functions of its class.

Memory allocation for objects =

member functions are created and placed in the memory space only once when they are defined — common for all objects

static member function (visible only within class, but its life time is entire program)

Array of objects = array of variables that are of type class

Objects as Function arguments:

Function argument passing:
- pass by value ← copy of an object is passed (changes to obj. inside func. do not affect the actual obj.)
- pass by reference ← address of object is passed (changes to object inside the function will reflect in the actual object)

constant member functions: (does not alter any data in the class)

```
void add(int,int) const;
Double get_balance() const;
```

Pointer to members:

```
class Stud { private:
                int id;
           };
int stud::*info = &stud::id
```

```
cla = &stud;
cout << cla->*info;
```

dereferencing operator ->* is used to access a member when pointers are used to both the object and the member.

data abstraction:
- emphasizes similarity of objects and ignore their differences
- " significant details " " unimportant details
- class is (ADT) Abstract data Type
- focuses on outside view of object only

encapsulation mechanism = binds together data and code
= it is achieved through data hiding (hiding structure of object and its implementation methods)

Polymorphism: (one function has many forms with respect to no. of parameters, type of parameters, order of appearance)

- **static poly:** (compiler binds call to function at compilation time itself)
- **dynamic poly:** (compiler binds call to the function at runtime depending upon context)

Dynamic Binding (linking of function call to the code to be executed in response to the call)
↓
(run-time)

Message Communication:=

Message passing:

```
students mark(name);
```

students → Objectname; mark → message; name → information

(objects have life cycle. so, they can be created and destroyed)

creating classes that define objects and their behaviors
↓
creating objects from class definitions
↓
establishing communication among objects

VC++

```
#include <iostream>
using namespace std;
void main ()
{ cout << "Hello!";
  cout << endl;
}
```

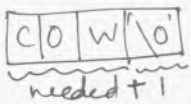
=* single quotes for single/one character –

char i = ' ';

=* double quotes for multiple characters –

char i[10] = " ";

=* array size should be one more than needed size. (needed size + 1) → for '\0' null character termination



char i[3+1] => char i[4].

---

func: getline for taking the line with array size

cin.getline(name, 10);

#include <string.h>

func: string copy

strcpy(name, "Gufran Ahmed");

---

passing by reference (& address of)

```
#include <iostream>
using namespace std;
void result(float, float&);
void main()
{ float num = 0.0;
  float square = 0.0;
  cout << "enter the number:" << endl;
  cin >> num;
  result(num, square);
  cout << "square of number i:" << square << endl;
}
void result(float num, float &square)
{ square = num * num;
}
```

* sending output to a data file for later retrieval

```
#include <fstream.h>

void main()
{
   ofstream outFile;
   outFile.open("filename.DAT");
   outFile << "put this into file outFile "filename.DAT" << endl;
}
```

variable = obj

.txt

— C++

= OOP (objects which contain data and functions to manipulate that data.)
= inheritance (reuse code and add data & functionality without making any changes to existing code)
= data encapsulation (allow data to be accessed only by selected functions)
= polymorphism (using operators and functions in different ways depending on what they are operating on)
i.e; creation of multiple definitions for functions and operators.

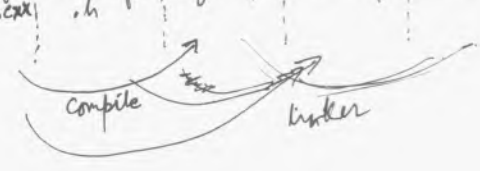
(compile time polymorphism)
(i) Operator overloading (use operators to operate on user-defined data types)
(ii) Function overloading (multiple functions with same name and similar operations but different data types or number of arguments.)

object (single unit that contains data and function)
* treat an application as a collection of objects. However, each object is organized into self-contained unit (class).

class _
{ member functions/methods ⟷ (function)
attributes ← (data)
};

* memory is allocated to each object of a class.
* The attributes in an object of a class are different from other objects of the same class but shares the same copy of methods.
* Abstraction (data abstraction) = reducing the data in an application to its basic attributes. (a simplified description, or a specification of a system that emphasizes some of the system details or properties while suppressing others.)
* By applying abstraction, an application has only the essential attributes and procedures.
* Encapsulation (ability to contain and hide information about an object, such as internal data structures or code.)
(eg the attributes of an object are manipulated only by the methods of the object.)
* Polymorphism:
static polymorphism (an entity existing in different physical forms simultaneously.)
involves the binding of functions on the basis of number, type, sequence of arguments, ...)
dynamic polymorphism (an entity changing its form depending on the circumstances.)
(A function exists in more than one form, and the calls to its various forms are resolved dynamically when the program is executed.)
(dynamic binding)

File types: source code file, header file, object file, make file, executable file
.c .cc .C .cxx .cpp .cxx .h



VC++6

VC++
* integrated Window Development Environment
  - Build apps (C/C++ and the Win32 API)
  - Build apps using (C/C++ and MFC) ✓
  - Build controls using ATL ✓
* Focus is on building apps using C++ and MFC

.cpp file → source file
folder Debug / .obj file → object file after compilation
" " / .pch → pre compile header file (once compiled) streamline the process of compilation
.dsp → project file
.dsw → workspace file

The Build Process

* VC++'s builtin compiler compiles source files into .obj (object) files
* " resource compiler compiles resources, data in the form of strings, icons, bitmaps, etc. that the project requires (.res).
* VC++ invokes the linker to add necessary libraries to the projects' object files, the resources and creates the executable (.exe)

Resources

* are edited by resource editors to visually create menus and dialogs
* are stored in project's resource script file (.rc).
* this text file can be viewed with editor, but edits should be made through tools.

Data Structures with C++

Operators: Prefix — x y — postfix (y: infix)

if (x==) ... else ... switch(x) { case ... }

Logical/boolean operators: && || !

* chaining assignments operators work from right to left.
* chaining arithmetic operators and chaining input/output operators work from left to right.

Iteration

- goto labelname; ⟶ labelname: ... goto labelname;
- do{ ... }while(...);
- while(){ };
- for(initialization; condition; statement){ ... };

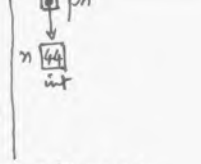
Functions: – can return a value to it
– is declared by one-line prototype and defined by its complete implementation below main()

String: – string class
– length(), substr(), (substring)
– are compared lexicographically (according to their dictionary ordering) ⟶ COMPUTABLE is less than COMPUTARE
<, ==, >
– concatenate (+)
– find() = to search for substrings in strings ⟶ find(substr)

Files: – include headers, #include<ctype.h>, <fstream>
– this " class, ifstream, ofstream
– * create objects of " "
– call getline(...)

Pointers: – variable whose value either is 0 or is the address of some other variable or object.
(null pointer)
(pointer points to the variable or object whose address it stores.)

* uninitialized pointer (dangling pointer) has unpredictable value.
* address of variable or object is the address of its first byte of memory storage.
* a pointer's type = pointer to xxxx (type of variable/object to which it points)
* The identifier for "pointer xxxx" is xxxx*



* &n = reference operator/address of operator is used to denote memory address of variable/object
* dereference operator (*p) evaluates to the value of variable/object to which p points. (i = *pn) ⇒ i = 44

derived type ← int* pn = &n
int m = *pn = n

& reference ⇄ * dereference

✓ Always initialize pointer variables

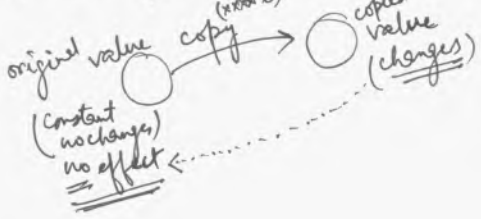
reference to (Alias) = synonym for an existing variable/object

(xxxx&)

int n = 44
int& rn = n

* the value of variable/object changes everywhere.
(pass by reference)

* (pass by value) = (value changes happen only locally.)



new operator creates object at runtime. Xxxx* p = new Xxxx();
delete " to terminate " " "
→ class
→ calls class constructor to create new object and assigned a pointer to this object.

Array — usually processed with for loops.
(static at compilation time)
— initialized with initializer list → string a[] = {"appli", "grape"};
can be

dynamic array created at runtime: — using new operator
(variable dimension)
xxxx* a = new xxxx[n];

* passing an array to a function :=

```
void sort(double a[], int n);
void sort(double* a, int n);
```
} same (equivalent)

sort(a, 10);

array name → a (constant pointer) → 0 1 2 3 . . (array)

(in multidimensional array, use typedef to define array type)

Classes (user-defined type) contains data members and functions (member functions).

```
class Point
{ ...
};
```
→ Point.h (interface for Point class

```
X(const X&);  // copy constructor
~X();  // destructor
X& operator=(const X&);  // assignment operator
```

Recursion:

```
long f(int n)
{ if(n<2) return 1;
  return n*f(n-1);
}
```

(factorial) iteration

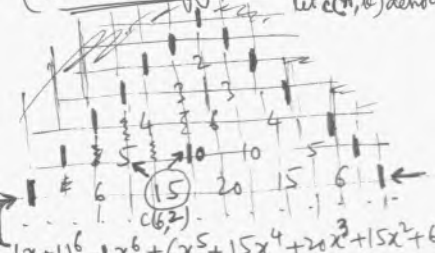
```
long f(int n)
{ long f=1;
  for(int i=2; i<=n; i++)
    f*=i;
  return f;
}
```

(Fiboracci) $F_n = F_{n-1} + F_{n-2}$

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

(Binomial coefficients) = By arranging them in a triangle, each interior number is the sum of the two directly above it. Let c(n,k) denote the coefficient in row number n and column number k (counting from 0).

$$c(n,k) = c(n-1,k-1) + c(n-1,k), \text{ for } 0 < k < n$$



$(x+1)^6 = 1x^6 + 6x^5 + 15x^4 + 20x^3 + 15x^2 + 6x + 1$

```
long c(int n, int k)
{ if(k==0 or k==n) return 1;
  return c(n-1,k-1)+c(n-1,k);
}
```

```
long c(int n, int k)
{ long c=1;
  for(int j=1; j<=k; j++)
    c = c*(n-j+1)/j;
  return c;
}
```

Euclidean Algorithm: (subtract repeatedly the smaller number n from the larger number m until the resulting difference d is smaller than n. Then repeat the same steps with d in place of n and with n in place of m. Continue until the two numbers are equal. That number is greatest common divisor, GCD.)

494 − 130 = 364
364 − 130 = 234
234 − 130 = 104
130 − 104 = 26
104 − 26 = 78
78 − 26 = 52
52 − 26 = 26

```
long gcd(long m, long n)
{ if(m==n) return n;
  else if(m<n) return gcd(m, n-m);
       else return gcd(m-n, n);
}
```

* In strong principle (second principle), inductive hypothesis allows to assume that all the preceding statements are true.
* In week " (first ") " " " " " " " only single " statement is true.
but these two principles are equivalent.

Complexity analysis of recursive algorithms (depend on the solubility of its recurrence relation.)

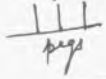
Let T(n) be the number of steps required to carry out the algorithm on a problem of size n.

dynamic programming: (to implement the recurrence relation by storing previously computed values in an array instead of recomputing them with recursive function calls.)

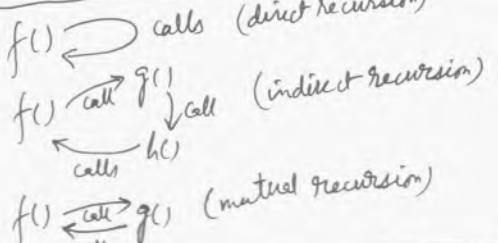
```
long fib(int n)
{ if(n<2) return n;
  long* f = new long[n];
  f[0] = 0;
  f[1] = 1;
for(int i=2; i<n; i++)
   f[i] = f[i-1]+f[i-2];
   return f[n-1]+f[n-2];
}
```

Tower of Hanoi: — move smaller n-1 disks from peg x to peg z.
— " remaining disk from peg x to peg y.
— " smaller n-1 disks from peg z to peg y.

hanoi(n,x,y,z)




```
void hanio(int n, char x, char y, char z)
{ if(n==1)
   cout<< "move top disk";
  else
  { hanio(n-1, x, y, y);
    hanio(1, x, y, z);
    hanio(n-1, y, x, z); }
}
```



STACKS (LIFO) –

standard C++ stack container class template:

```
template<class T> class stack
{ public:
    stack();
    stack(const stack&);
    ~stack();
    stack& operator=(const stack&);
    int size() const;
    bool empty() const;
    T& top();
    void push(const T&);
    void pop();
    ...
};
```

T = type of object

this called is defined in standard <stack> header.

As a template, the interface given above can be used only by specifying the type of object that is to be stored in its instance.

```
stack<type of object> object;
stack<int> S1;        S1.top();
stack<Card> S4;
stack<string> S3;     S3.top() = "McGraw-Hill";
```

Applications of stacks:

postfix notation (reverse Polish notation, RPN), ex. $3*(4+5) \Rightarrow 345+*$ (infix)

Queue (FIFO): container

```
template <class T> class queue
{public:
   queue();
   queue(const queue&);
  ~queue();
   queue& operator=(const queue&);
   int size() const;
   bool empty() const;
    T& front();
    T& back();
    void push(const T&);
    void pop();
    ....
    ...
};
```

T = type of object

defined in <queue> header.

```
T& front();
queue<int> q1;
queue<string> q2;        q2.push("hiasoft");
queue<person> q3;
queue<stack<int>> q4;
```

q4 = queue of integer stacks

queue<type of object> object;

List: (sequential container that can insert and delete elements locally) in

```
template <class T> class list
{public:
    list();
    list(const list&);
    list(int, const T& = T());
    list(int);
    list(iterator, iterator);
   ~list();
    list& operator=(const list&);
    void assign(int, const T& = T());
    void assign(iterator, iterator);
    void resize(int);
    void swap(list&);
    bool empty() const;
    int size() const;
    iterator begin();
    iterator end();
    T& front();
    T& back();
    void push_front(const T&);
    void pop_front();
    void push_back(const T&);
    void pop_back();
    iterator insert(iterator, const T& = T());
    void insert(iterator, int, const T& = T());
    void insert(iterator, iterator, iterator);
    iterator erase(iterator);
    iterator erase(iterator, iterator);
    void remove(const T&);
    void clear();
    void reverse();
    void unique();
    void merge(list&);
    void splice(iterator, list&);
    void splice(iterator, list&, iterator);
    void splice(iterator, list&, iterator, iterator);
};
```

Iterator: (object, capable of moving down (or up) a list from one element to the next.)

[array[subscript] — similar — alternative — list(iterator)] similarity

table (map / lookup table / associative array / dictionary) ≒
- a container that allows direct access by any index type.
- it works like an array or vector except that the index variable need not be an integer.
- a sequence of pairs (key, value) like $f(key) = value$

```
template<class T1, class T2>
class pair
{ T1 first;
  T2 second;
  pair(): first(T1()), second(T2()) { }
  pair(const T1& x, const T2& y): first(x), second(y) { }
    - - - -
};
```