

Data Structures & Algorithms in C++

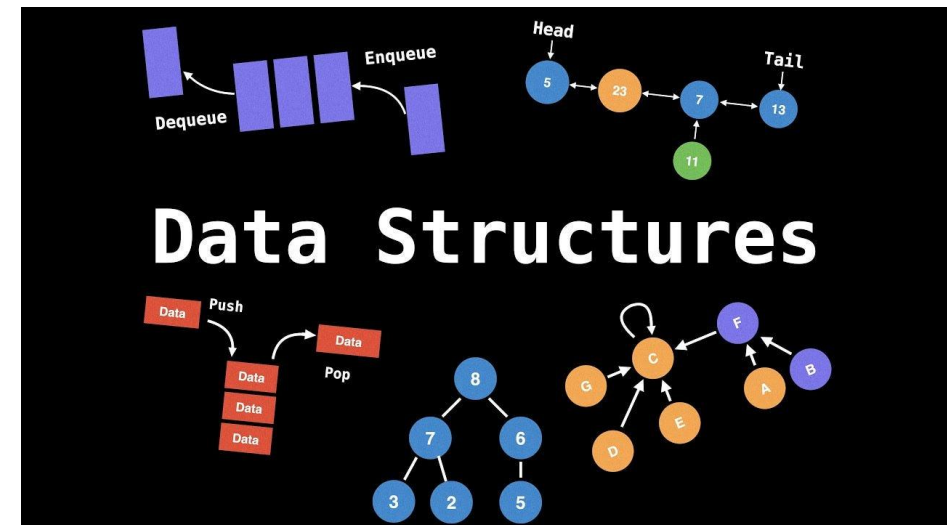
WEEK 01

Introduction

- Concept of Data Structures
- Data Structures and Algorithms
- Classification of Data Structures
- Data Expressions
- Control Flow, Functions, Classes

Data Structures ?

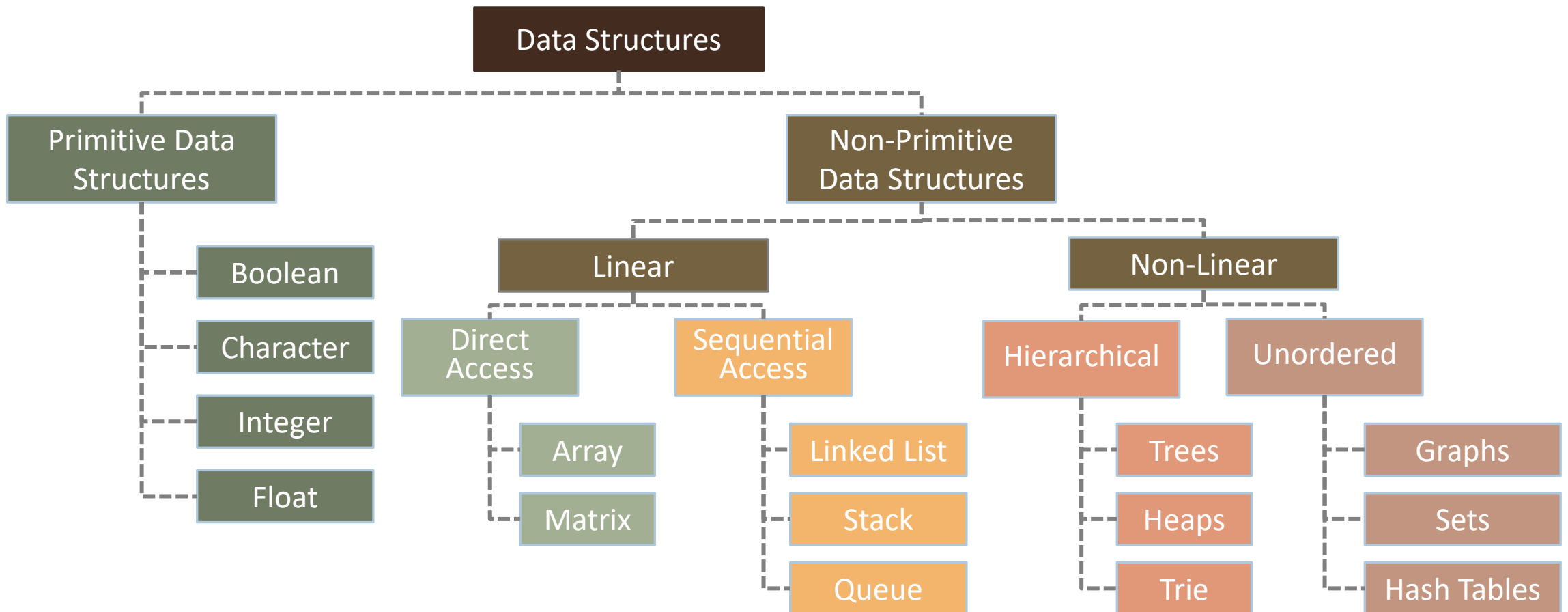
- A Data Structure is a specific way of organizing and storing data within a program, allowing for **efficient** access and manipulation of that data
 - Essential ingredients in creating fast and powerful algorithms
 - Help to manage and organize data
 - Make code cleaner and easier to understand



Data Structures and Algorithms

- An algorithm is a procedure used for solving a problem or performing a computation
 - A step-by-step procedure or formula for solving a problem
 - Examples: sorting, searching, traversal
- Data structures are containers for data, algorithms are tools for manipulating these structures
- Basic components of an **Algorithm**
 - Input, Output, **Clearness**, **Finiteness**, **Effectiveness**

List of Data Structures



Fundamental Data Types

➤ Primitive Data Structures

- **bool** : boolean value, either true or false
- **char** : character
- **short** : short integer
- **int** : integer
- **long** : long integer
- **float** : single-precision floating-point number
- **double** : double-precision floating-point number
- **enum** : a user-defined type that can hold any of a set of discrete values

Complex Data Types

➤ Pointers

- a variable that holds the value of such an [address](#)
- address-of operator : &
- dereferencing : accessing an object's value from its address

```
char ch = 'Q';  
char* p = &ch;           // p holds the address of ch  
cout << *p;              // outputs the character 'Q'  
ch = 'Z';                 // ch now holds 'Z'  
cout << *p;              // outputs the character 'Z'  
*p = 'X';                 // ch now holds 'X'  
cout << ch;              // outputs the character 'X'
```

- ## ➤ References
- ```
string author = "Samuel Clemens";
string& penName = author;
penName = "Mark Twain";
```

# Complex Data Types

## ➤ Pointers vs. References

| Aspect             | Pointer                                                                   | Reference                                                                                  |
|--------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <b>Definition</b>  | A variable that stores the memory address of another variable             | An alias (alternative name) for an already existing variable                               |
| <b>Syntax</b>      | Uses the * operator to declare and dereference                            | Uses the & operator for declaration (not to be confused with the & in address-of)          |
| <b>Nullability</b> | Can be assigned nullptr to indicate it points to no valid memory location | Must always reference a valid variable; cannot be null                                     |
| <b>Indirection</b> | Must be dereferenced using * to access or modify the value it points to   | Directly accesses the value of the variable it references, without requiring dereferencing |
| <b>Usage</b>       | Commonly used for dynamic memory management and data structures           | Simplifies syntax for passing variables by reference and aliasing objects                  |



# Complex Data Types

---

## ➤ Arrays

- A collection of elements of the **same type**
- Referenced by its **index**
- No built-in run-time checking for out of bounds (vs. STL vector)

```
double f[5]; // array of 5 doubles: f[0], ..., f[4]
int m[10]; // array of 10 ints: m[0], ..., m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]]; // outputs f[4], which is 2.5
```

- The name of an array is equivalent to a pointer to the array's initial element

```
char c[] = {'c', 'a', 't'};
char* p = c; // p points to c[0]
char* q = &c[0]; // q also points to c[0]
cout << c[2] << p[2] << q[2]; // outputs "ttt"
```

# Complex Data Types

---

## ➤ Strings

- Fixed-length array of characters that ends with the null character (C-style strings)
- C++ provides **STL strings**
  - Concatenated using “+” operator
  - Compared with each other using lexicographic order (“<”, “>”,...)

```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;
string u = s + " or " + t;
if (s > t)
 cout << u;
```

```
string s = "John";
int i = s.size();
char c = s[3];
s += " Smith";
```

# Complex Data Types

---

## ➤ C-Style Structures

- Useful for storing an aggregation of elements of the **different type**

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
 string name; // passenger name
 MealType mealPref; // meal preference
 bool isFreqFlyer; // in the frequent flyer program?
 string freqFlyerNo; // the passenger's freq. flyer number
};

Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };
```

- ## ➤ C++ provides a much more powerful and flexible construct called a **class** with both data and functions

# Complex Data Types

---

## ➤ Pointer and **dynamic memory allocation (new)**

```
Passenger *p;
// ...
p = new Passenger; // p points to the new Passenger
p->name = "Pocahontas"; // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";
```

- Because C++ does not provide automatic garbage collection, all dynamically allocated objects should be **explicitly** deleted (cf. memory leak)

```
char* buffer = new char[500]; // allocate a buffer of 500 chars
buffer[3] = 'a'; // elements are still accessed using []
delete [] buffer; // delete the buffer
```

# Complex Data Types

## ➤ Comparison between new and malloc

| Aspect                     | new (in C++)                                                                 | malloc (in C)                                                                       |
|----------------------------|------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <b>Purpose</b>             | Used to allocate memory and automatically call the constructor of the object | Used to allocate memory, but does not call any constructors                         |
| <b>Syntax</b>              | Type* pointer = new Type;<br>ex) int* p = new int;                           | void *pointer = malloc(size_t size);<br>ex) int *p = (int *)malloc(sizeof(int));    |
| <b>Memory deallocation</b> | using delete<br>ex) delete p;                                                | using free<br>ex) free(p);                                                          |
| <b>Type safety</b>         | Type-safe; no need for explicit casting<br>ex) int* p = new int;             | Not type-safe; requires explicit casting<br>ex) int *p = (int*)malloc(sizeof(int)); |

# Complex Data Types

---

## ➤ Namespaces and Using statements

- allow a group of related names to be defined in one space

- fully qualified name : ex) myglobals::cat

```
namespace myglobals {
 int cat;
 string dog = "bow wow";
}
```

- the using statement makes some or all of the names from the namespace accessible

```
using std::string; // makes just std::string accessible
using std::cout; // makes just std::cout accessible
```

```
using namespace myglobals; // makes all of myglobals accessible
```

# Expressions

## ➤ Operator Precedence

| Type                    | Operators                                    |
|-------------------------|----------------------------------------------|
| scope resolution        | namespace_name :: member                     |
| selection/subscripting  | class_name.member pointer->member array[exp] |
| function call           | function(args)                               |
| postfix operators       | var++ var--                                  |
| prefix operators        | ++var --var +exp -exp ~exp !exp              |
| dereference/address     | *pointer &var                                |
| multiplication/division | * / %                                        |
| addition/subtraction    | + -                                          |
| shift                   | << >>                                        |
| comparison              | < <= > >=                                    |
| equality                | == !=                                        |
| bitwise and             | &                                            |
| bitwise exclusive-or    | ^                                            |
| bitwise or              |                                              |
| logical and             | &&                                           |
| logical or              |                                              |
| conditional             | bool_exp ? true_exp : false_exp              |
| assignment              | = += -= *= /= %= >>= <<= &= ^=  =            |

# Expressions

---

## ➤ Type casting

```
int i1 = 18;
int i2 = 16;
double dv1 = i1 / i2; // dv1 has value 1.0
double dv2 = double(i1) / double(i2); // dv2 has value 1.125
double dv3 = double(i1 / i2); // dv3 has value 1.0
```

### ■ Explicit cast operators

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = static_cast<int>(d1); // i1 has value 3
int i2 = static_cast<int>(d2); // i2 has value 3
```

### ■ Implicit cast operators

```
int i = 3;
double d = 4.8;
double d3 = i / d; // d3 = 0.625 = double(i)/d
int i3 = d3; // i3 = 0 = int(d3)
 // Warning! Assignment may lose information
```



# Control Flow

---

➤ If Statement

➤ Switch Statement

➤ While/Do-While Loops

➤ For Loop

```
// Initialize an int array
int num[5] = {1,2,3,4,5};
```

```
// Ranged for loop
for (int n : num)
 std::cout << n << std::endl;
```

```
// Nested for loop
for (int i=0; i<5; i++)
 std::cout << num[i] << std::endl;
```

# Functions

---

## ➤ Argument Passing

### ■ Call by Reference, Call by Value

```
void f(int value, int& ref) { // one value and one reference
 value++; // no effect on the actual argument
 ref++; // modifies the actual argument
 cout << value << endl; // outputs 2
 cout << ref << endl; // outputs 6
}

int main() {
 int cat = 1;
 int dog = 5;
 f(cat, dog); // pass cat by value, dog by ref
 cout << cat << endl; // outputs 1
 cout << dog << endl; // outputs 6
 return EXIT_SUCCESS;
}
```

### ■ Array arguments : use pointer ( $T[] \rightarrow T^*$ )

# Functions

---

## ➤ Overloading

- **Function overloading** : same name with different argument lists

```
void print(int x) // print an integer
{ cout << x; }
```

```
void print(const Passenger& pass) { // print a Passenger
 cout << pass.name << " " << pass.mealPref;
 if (pass.isFreqFlyer)
 cout << " " << pass.freqFlyerNo;
}
```

- **Operator overloading** : +, \*, +=, ==, <<, ...

```
bool operator==(const Passenger& x, const Passenger& y) {
 return x.name == y.name
 && x.mealPref == y.mealPref
 && x.isFreqFlyer == y.isFreqFlyer
 && x.freqFlyerNo == y.freqFlyerNo;
}
```

# Classes

---

## ➤ Class Structure

- Data members (member variables) + member functions (methods)

```
class Counter { // a simple counter
public:
 Counter(); // initialization
 int getCount(); // get the current count
 void increaseBy(int x); // add x to the count
private:
 int count; // the counter's value
};
```

- Access control
  - “Private” means that they are accessible only from within the class
- Member functions
  - Access functions, Update functions

# Classes

## ➤ Constructors

- Default constructors, copy constructors

```
class Passenger {
private:
 // ...
public:
 Passenger(); // default constructor
 Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
 Passenger(const Passenger& pass); // copy constructor
 // ...
};
```

- Initializer list

```
 // constructor using an initializer list
Passenger::Passenger(const string& nm, MealType mp, string ffn)
 : name(nm), mealPref(mp), isFreqFlyer(ffn != "NONE")
 { freqFlyerNo = ffn; }
```

## ➤ Destructors

```
class Vect {
public:
 Vect(int n);
 ~Vect();
 // ... other public members omitted
private:
 int* data;
 int size;
};

Vect::Vect(int n) {
 size = n;
 data = new int[n];
}

Vect::~~Vect() {
 delete [] data;
}
```

# Classes

## ➤ Constructors (Cont.)

```
Passenger::Passenger() { // default constructor
 name = "--NO NAME--";
 mealPref = NO_PREF;
 isFreqFlyer = false;
 freqFlyerNo = "NONE";
}
```

```
 // constructor given member values
Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
 name = nm;
 mealPref = mp;
 isFreqFlyer = (ffn != "NONE"); // true only if ffn given
 freqFlyerNo = ffn;
}
```

```
 // copy constructor
Passenger::Passenger(const Passenger& pass) {
 name = pass.name;
 mealPref = pass.mealPref;
 isFreqFlyer = pass.isFreqFlyer;
 freqFlyerNo = pass.freqFlyerNo;
}
```

```
Passenger p1; // default constructor
Passenger p2("John Smith", VEGETARIAN, 293145); // 2nd constructor
Passenger p3("Pocahontas", REGULAR); // not a frequent flyer
Passenger p4(p3); // copied from p3
Passenger p5 = p2; // copied from p2
Passenger* pp1 = new Passenger; // default constructor
Passenger* pp2 = new Passenger("Joe Blow", NO_PREF); // 2nd constr.
Passenger pa[20]; // uses the default constructor
```

# Classes

---

- Every class that allocates its own objects using **new** should:
  - Define a **destructor** to free any allocated objects
  - Define a **copy constructor**, which allocates its own new member storage and copies the contents of member variables
  - Define an **assignment operator**, which deallocates old storage, allocates new storage, and copies all member variables

```
Vect& Vect::operator=(const Vect& a) {
 if (this != &a) {
 delete [] data;
 size = a.size;
 data = new int[size];
 for (int i=0; i < size; i++) {
 data[i] = a.data[i];
 }
 }
 return *this;
}
```

```
Vect a(100);
Vect b = a;
Vect c;
c = a;
```

# Classes

## ➤ Friend function

```
class SomeClass {
private:
 int secret;
public:
 // ... // give << operator access to secret
 friend ostream& operator<<(ostream& out, const SomeClass& x);
};

ostream& operator<<(ostream& out, const SomeClass& x)
{ cout << x.secret; }
```

## ➤ Class friendship

```
class Vector { // a 3-element vector
public: // ... public members omitted
private:
 double coord[3]; // storage for coordinates
 friend class Matrix; // give Matrix access to coord
};

class Matrix { // a 3x3 matrix
public:
 Vector multiply(const Vector& v); // multiply by vector v
 // ... other public members omitted
private:
 double a[3][3]; // matrix entries
};

Vector Matrix::multiply(const Vector& v) { // multiply by vector v
 Vector w;
 for (int i = 0; i < 3; i++)
 for (int j = 0; j < 3; j++)
 w.coord[i] += a[i][j] * v.coord[j]; // access to coord allowed
 return w;
}
```



# Classes

## - STL (Standard Template Library)

- A collection of useful classes for common data structures

| STL            | Descriptions                              |
|----------------|-------------------------------------------|
| stack          | Container with last-in, first-out access  |
| queue          | Container with first-in, first-out access |
| deque          | Double-ended queue                        |
| vector         | Resizable array                           |
| list           | Double linked list                        |
| priority_queue | Queue ordered by value                    |
| set            | Set                                       |
| map            | Associated array (Dictionaries)           |

- Vector (vs. Array)

- A dynamic container from the STL that can resize itself as needed
- Provide bounds checking with `at()` member function, throwing exceptions for out-of-bound access
- Copying the contents of one vector to the other

```
int i = // ...
cout << scores[i]; // index (range unchecked)
buffer.at(i) = buffer.at(2 * i); // index (range checked)
vector<int> newScores = scores; // copy scores to newScores
scores.resize(scores.size() + 10); // add room for 10 more elements
```

# Classes

## - STL (Standard Template Library)

### ➤ STL string class

|                                 |                                                                                                      |
|---------------------------------|------------------------------------------------------------------------------------------------------|
| <code>s.find(p)</code>          | Return the index of first occurrence of string <i>p</i> in <i>s</i>                                  |
| <code>s.find(p, i)</code>       | Return the index of first occurrence of string <i>p</i> in <i>s</i> on or after position <i>i</i>    |
| <code>s.substr(i,m)</code>      | Return the substring starting at position <i>i</i> of <i>s</i> and consisting of <i>m</i> characters |
| <code>s.insert(i, p)</code>     | Insert string <i>p</i> just prior to index <i>i</i> in <i>s</i>                                      |
| <code>s.erase(i, m)</code>      | Remove the substring of length <i>m</i> starting at index <i>i</i>                                   |
| <code>s.replace(i, m, p)</code> | Replace the substring of length <i>m</i> starting at index <i>i</i> with <i>p</i>                    |
| <code>getline(is, s)</code>     | Read a single line from the input stream <i>is</i> and store the result in <i>s</i>                  |

### ➤ Example

```
string s = "a dog"; // "a dog"
s += " is a dog"; // "a dog is a dog"
cout << s.find("dog"); // 2
cout << s.find("dog", 3); // 11
if (s.find("doug") == string::npos) { } // true
cout << s.substr(7, 5); // "s a d"
s.replace(2, 3, "frog"); // "a frog is a dog"
s.erase(6, 3); // "a frog a dog"
s.insert(0, "is "); // "is a frog a dog"
if (s == "is a frog a dog") { } // true
if (s < "is a frog a toad") { } // true
if (s < "is a frog a cat") { } // false
```

# Classes

## - An Example Program

### ➤ Header file

```
#ifndef CREDIT_CARD_H // avoid repeated expansion
#define CREDIT_CARD_H

#include <string> // provides string
#include <iostream> // provides ostream

class CreditCard {
public:
 CreditCard(const std::string& no, // constructor
 const std::string& nm, int lim, double bal=0);
 // accessor functions
 std::string getNumber() const { return number; }
 std::string getName() const { return name; }
 double getBalance() const { return balance; }
 int getLimit() const { return limit; }

 bool chargeIt(double price); // make a charge
 void makePayment(double payment); // make a payment
private:
 std::string number; // credit card number
 std::string name; // card owner's name
 int limit; // credit limit
 double balance; // credit card balance
};

// print card information
std::ostream& operator<<(std::ostream& out, const CreditCard& c);
#endif
```

### ➤ Out-of-class member functions

```
#include "CreditCard.h" // provides CreditCard

using namespace std; // make std:: accessible
 // standard constructor
CreditCard::CreditCard(const string& no, const string& nm, int lim, double bal) {
 number = no;
 name = nm;
 balance = bal;
 limit = lim;
}

// make a charge
bool CreditCard::chargeIt(double price) {
 if (price + balance > double(limit))
 return false; // over limit
 balance += price;
 return true; // the charge goes through
}

void CreditCard::makePayment(double payment) { // make a payment
 balance -= payment;
}

// print card information
ostream& operator<<(ostream& out, const CreditCard& c) {
 out << "Number = " << c.getNumber() << "\n"
 << "Name = " << c.getName() << "\n"
 << "Balance = " << c.getBalance() << "\n"
 << "Limit = " << c.getLimit() << "\n";
 return out;
}
```

# Classes

## - An Example Program

### ➤ Test function

```
#include <vector> // provides STL vector
#include "CreditCard.h" // provides CreditCard, cout, string

using namespace std; // make std accessible

void testCard() { // CreditCard test function
 vector<CreditCard*> wallet(10); // vector of 10 CreditCard pointers
 // allocate 3 new cards
 wallet[0] = new CreditCard("5391 0375 9387 5309", "John Bowman", 2500);
 wallet[1] = new CreditCard("3485 0399 3395 1954", "John Bowman", 3500);
 wallet[2] = new CreditCard("6011 4902 3294 2994", "John Bowman", 5000);

 for (int j=1; j <= 16; j++) { // make some charges
 wallet[0]->chargeIt(double(j)); // explicitly cast to double
 wallet[1]->chargeIt(2 * j); // implicitly cast to double
 wallet[2]->chargeIt(double(3 * j));
 }

 cout << "Card payments:\n";
 for (int i=0; i < 3; i++) { // make more charges
 cout << *wallet[i];
 while (wallet[i]->getBalance() > 100.0) {
 wallet[i]->makePayment(100.0);
 cout << "New balance = " << wallet[i]->getBalance() << "\n";
 }
 cout << "\n";
 delete wallet[i]; // deallocate storage
 }
}
```

### ➤ Main function & Output

```
int main() {
 testCard();
 return EXIT_SUCCESS;
}
```

Card payments:  
Number = 5391 0375 9387 5309  
Name = John Bowman  
Balance = 136  
Limit = 2500  
New balance = 36

Number = 3485 0399 3395 1954  
Name = John Bowman  
Balance = 272  
Limit = 3500  
New balance = 172  
New balance = 72

Number = 6011 4902 3294 2994  
Name = John Bowman  
Balance = 408  
Limit = 5000  
New balance = 308  
New balance = 208  
New balance = 108  
New balance = 8