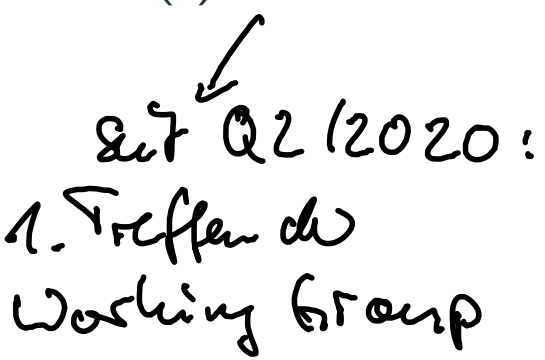
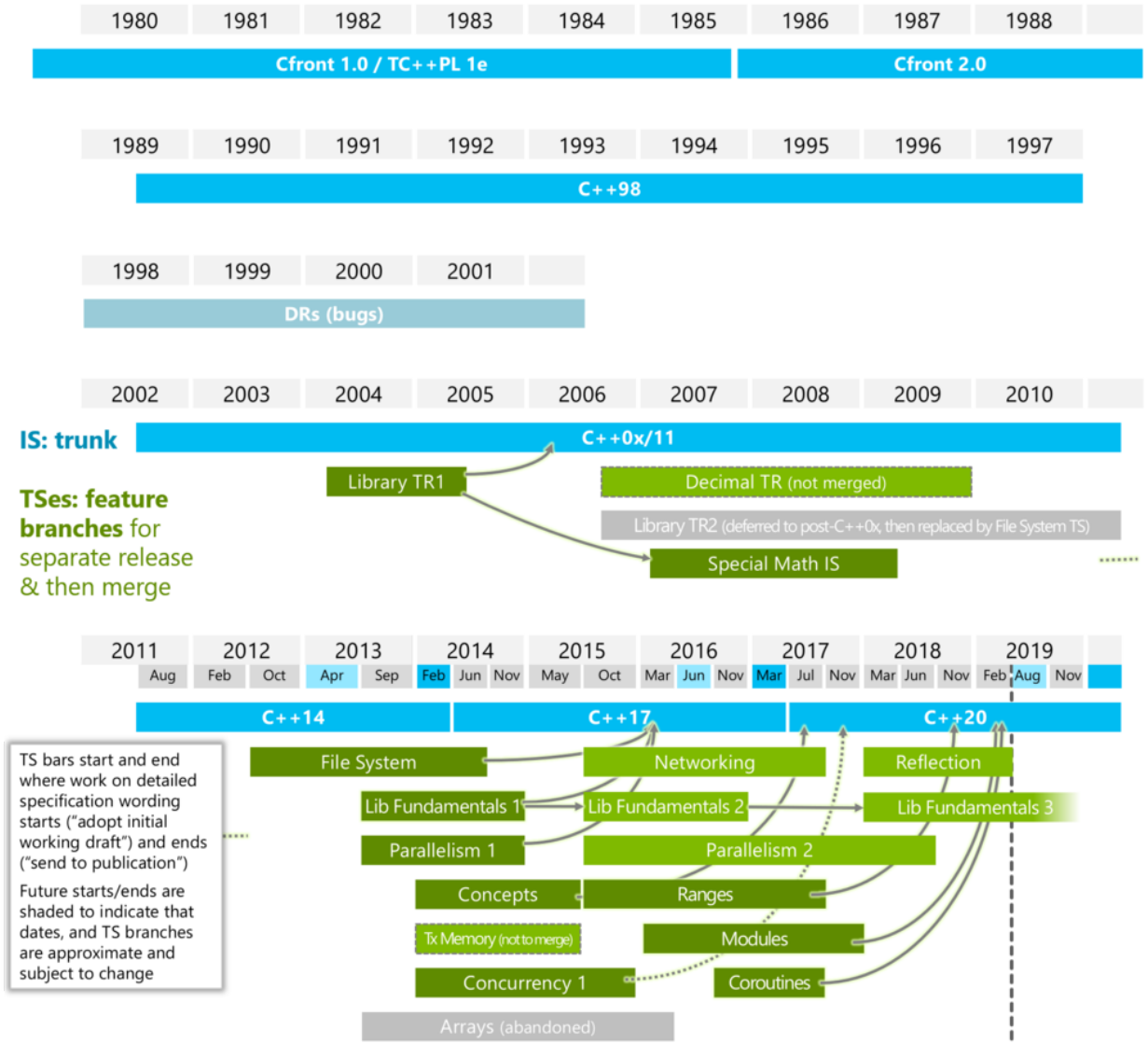
Kapitel 3: C++-Grundlagen

Charakterisierung

- Objektorientierte Programmiersprache
- In den 1980er Jahren wesentlich von Bjarne Strostrup bei AT&T entwickelt
- 1989 In der ersten Version veröffentlicht
- 1998: ISO-Standard C++98
- Seither fortlaufend weiterentwickelte ISO Standards: C++03/11, C++14, C++17, in Arbeit: C++20 und (!) C++23



C++-Standardentwicklung

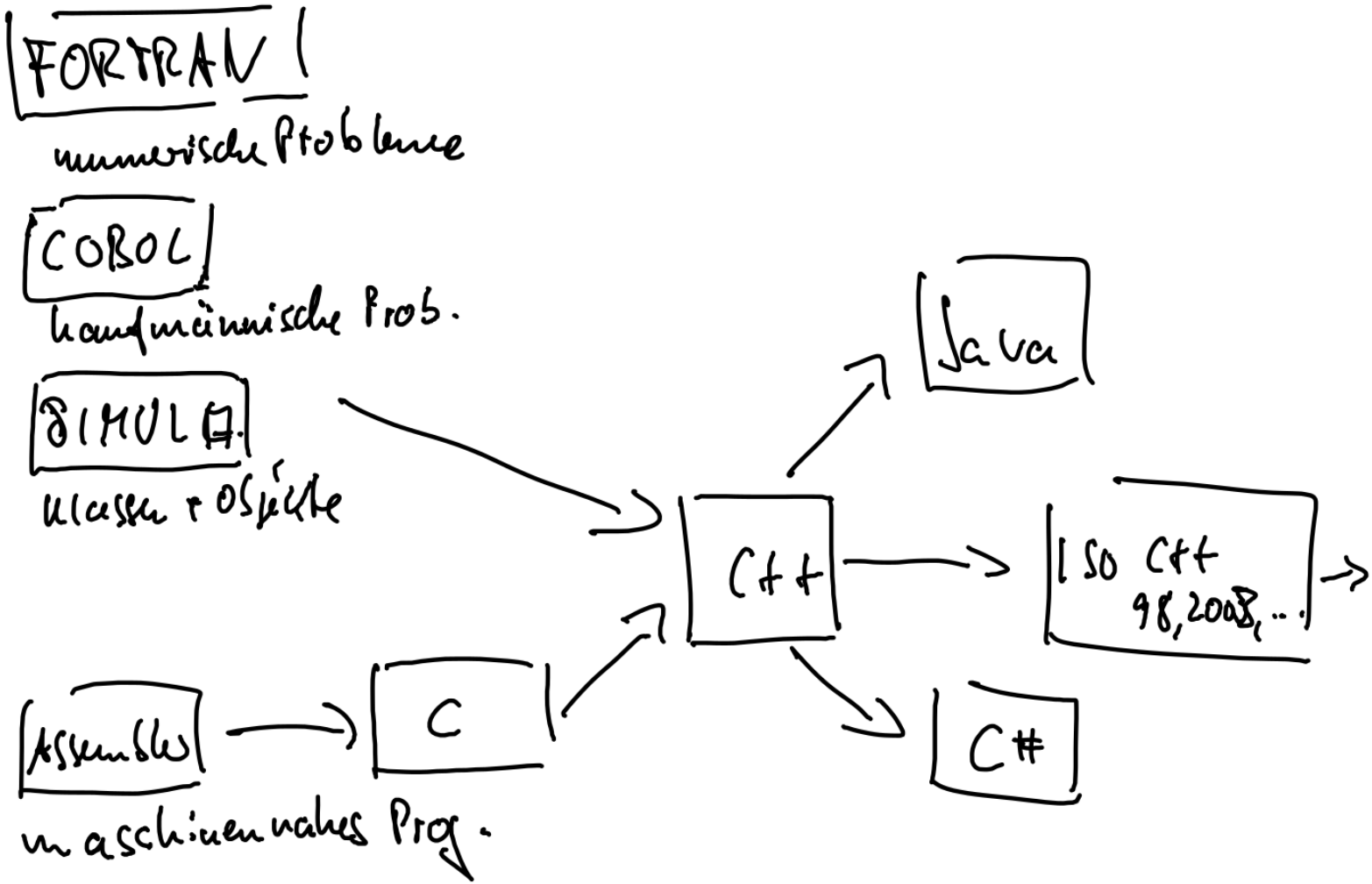


Quelle: https://isocpp.org/std/status

Einflüsse und Ziele

- Situation in den 1960er/70er Jahren: Wahl zwischen domänenspezifischen Hochsprachen (FORTRAN, COBOL) oder C/Assembler
- Stroustups Beobachtung: Simula als Domänen-neutrale Hochsprache, aber nicht maschinennah
  - Starke Abstraktion durch Klassen (dort eingeführt!)
  - Allgemein verwendbar
- C/Assembler: zu wenig Abstraktionsmöglichkeiten
- Ziel: beides kombinbieren $\Rightarrow$ C++

C++-Stammbaum (vereinfacht)



Ursprüngliche Ziele von C++

- Sichere Typisierung (Type Safety): unsichere Operationen kapseln
- Ressourcenkontrolle (Resource Safety): nicht nur Speicher!
- Performance: in den kritischen Bereichen fast jeder Anwendung relevant
- Vorhersagbarkeit (Predictability): für Echtzeit-Bedingungen
- Erlernbarkeit (Teachability): einfache Probleme sollten keinen komplexen Code erfordern
- Lesbarkeit (Readability): mensch- und maschinenlesbar

siehe auch: "The Essence of C++", Vortrag B. Stroustrup/University of Edinburgh/2014

Heutige und zukünftige Ausrichtung von C++

- Maschinnenahes Programmieren ermöglichen (analog C)
  - Direkter Zugriff auf Speicher
  - Native Typen bilden Speicherworte auf Prozessorebene ab
  - Direkte Übersetzung in Maschinencode
- Starke Abstraktion, wenig Laufzeit- und Speicher-Overhead
  - Klassenabstraktionen, generische und funktionale Programmierung
  - Nur minimale Laufzeit-Typinformationsverarbeitung
  - Mechanismen transparent aber Aufwand bekannt/abschätzbar
- Wichtige Design-Konstante: Abwärtskompatibilität
  - Garantiert langzeitstabile Entwicklung
  - Nutzbare Codebasen (Assets, IP, stabile Systeme) über Jahrzehnete möglich

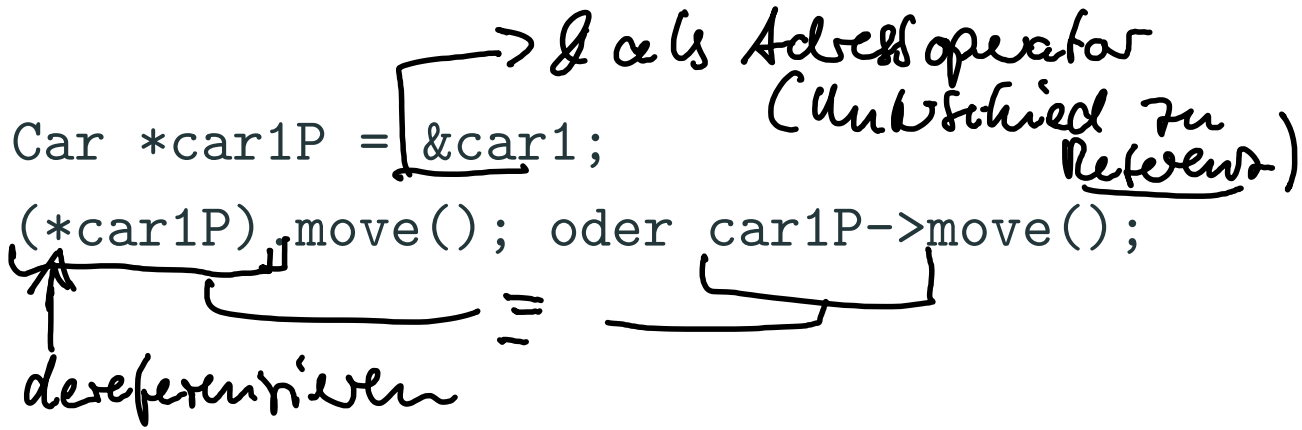
Pointer als Alias

Pointer aus C bekannt

(I) Adresse einer Variablen

(II) Alternative Möglichkeit, auf die Variable zuzugreifen

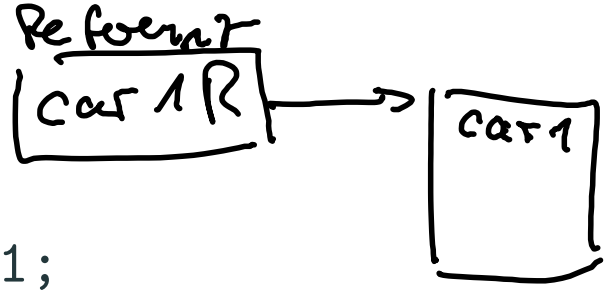
```
Car car1;          Car *car1P = &car1;
car1.move();       (*car1P).move(); oder car1P->move();
```



- `->` ist als Ersatz für `(* ).` leichter zu schreiben (und zu lesen)
- Weitere Pointer möglich: mehrere *Aliase* für dasselbe Objekt

Referenzen

C++-*Referenzen* (auch als `Alias` bezeichnet) bieten (II) ohne (I) :




```
Car car1;
```

```
Car &car1R = car1;
```

↳ keine Zuweisung, sondern Initialisierung eines Referenzvariablen

```
car1.move();
```

↑ Objektvariable

```
car1R.move();
```

↳ Referenzvariable, am ∘-Zugriff nicht erkennbar

↳ & bei Deklarationen : Referenz

Referenzen (2)

- Syntax für Member-Zugriff sieht aus wie bei "normaler" Objektvariable (durch .)
- Referenzen *müssen* bei der Definition sofort initialisiert werden

  ⇒ Car &car2r; nicht erlaubt (weil nicht initialisiert)

  ⇒ nur gültige Referenzen möglich
- Technisch intern über Pointer gelöst (den man aber nicht sieht, analog Java-Objektreferenzen)
- Typname hier: `Car &` (analog `Car *` für Pointer) bezeichnet als "Car Reference" oder "Reference to Car"
- keine null-Referenzen (wie NULL-Pointer)
- Spätere Zuweisungen an Referenz ändern das Objekt, nicht die Referenz

Verwendung von Referenzen

```
// Verändern von übergebenen Objekten

void f1(struct Address adr) {  adr.zipCode = 65185; }

void f2(struct Address &adr) {  adr.zipCode = 65185; }

int main(void) {
   struct Address adr1 = {..,65203 /*zipCode*/,...};
   f1(adr1);       f2(adr1);
}
```

↳ Objekt, pass by value (copy!)

↳ Reference, pass by reference (alias, keine Kopie!)

verändert nur die Kopie von adr1 (keine Änderung nach Rückkehr)

verändert das Original-Objekt, auf das per Referenz verwiesen wird.

Unterscheidung der Aufrufstellen nur durch Blick auf Prototyp zu erkennen

const &

```
// Objekte übergeben, die nicht verändert werden sollen:

void f1(struct Address adr) {  adr.zipCode = 65185; }

void f2(struct Address const &adr) {  adr.zipCode = 65185; }
```

const-Referenz: Zielobjekt darf nicht verändert werden (über diese Referenz)

↳ nicht zulässig: const Ref.

Vorteile const & (f2) gegenübner Pass-by-Value (f1):

- Overhead des Kopierens großer Argument-Objekte entfällt
- Offensichtlicher Fehler (beim Compilieren), wenn versucht wird, das Objekt zu verändern

Überladen von Funktionen

Auch für "normale" C++-Funktionen (nicht nur Member Functions):

- Ein Funktionsname kann für mehrere *Signaturen* definiert werden.
- Welche Variante bei einem Aufruf ausgeführt wird, wird dann nach anzahl und Typ der Argumente entschieden.
- *Signatur*: Anzahl und Typen der Funktionsparameter. Im engeren Sinne `nicht` der return-Typ (aber sinnvoll, diesen mit zu betrachten).
- *Parameter* einer Funktion werden bei der *Deklaration* der Funktion angegeben
- *Argumente* sind die Werte, die beim Aufruf tatsächlich als Parameterwerte übergeben werden

Überladen von Funktionen: Beispiele

```
void f(int i) {}
void f(int i, char c) {}


void main(){
   f(5);      f(5 ,a);
}



class Car {
  Car() {}
  Car(int initMileage) {}
}

Car car1;     Car car2(10);
```

Default-Argumente

Bei beiliebigen Funktionen möglich: Standardwerte angeben

- Diese Argumente können dann beim Aufruf weggelassen werden
- Nach Defaults dürfen keine Parameter ohne Default folgen
- Vorsicht: Doppeldeutigkeiten möglich, (aber erst beim Aufruf!)

```
void f(int i) {}
void f(char c) {}
void f(int i=5, char c='x') {}

void main(){
   f('a');      f(3,'b');    f();     f(5);
}
```

C++-Objekte

```
class Car {
public:
   int getWeight()
    {return this->weight;
         // this-> hier optional}

private:
   int weight;
};


void main() {
Car car1, car2;
car1.getWeight(); // this zeigt in getWeight() auf car1
}
```

Data Members

- Jedes Objekt hat seine eigenen, separaten Data Members
- Zugriff in Member Functions direkt über den Variablennamen des Members oder über `this->member`
- Wenn `public`: auch Zugriff von außen über `objekt.member` oder `objektPointer->member`
- Ausnahme: `static` Data Members:
  - eine einzige Variable für die *Klasse* (für alle Objekte der Klasse)
  - Zugiff wie auf normale Data Members oder mit `Klassenname::member`

Data Members: Beispiel

```
class Car {
public:
   int getWeight() {return weight}

private:
   int weight;
   static int numberOfCars;
};

int Car::numberOfCars = 0;

void main() {
Car car1, car2;
// bedeutet alles dasselbe:
car1.numberOfCars    car2.numberOfCars Car::numberOfCars
// unterschiedlich:
car1.getWeight()     car2.getWeight()
}
```

Member Functions

- Code einer Member Funtion exitiert 1x für alle Objekte
- Beim Aufruf zeigt `this` auf das Objekt, an dem die Funktion ausgeführt wird
- Aufruf anderer Member Functions desrselben Klasse einfach über `methode()` (oder `this->methode()`
- Wenn `public`: auch Zugriff von außen über `objekt.methode()` oder `objektPointer->methode()`
- Ausnahme: `static` Member Functions:
  - Beim Aufruf *kein* `this` verfügbar
  - Nur Zugriff auf `static` Data Members
  - Aufruf von außen mit `Klassenname::methode()`

Member Functions: Beispiel

```
class Car {
public:
   int getWeight() {return weight}
   static int getNumberOfCars() {weight=5; return numberOfCars;}
private:
   int weight;
   static int numberOfCars;
};

int Car::numberOfCars = 0;

void main() {
Car car1, car2;
// bedeutet alles dasselbe:
car1.getNumberOfCars()    car2.getNumberOfCars()  Car::getNumberOfCars()
}
```

Objekterzeugung

1. Speicherallokation (lokal: auf Stack, mit `new`: auf Heap, global: Speicher schon allokiert), Größe: sizeof *Klasse*
2. Initialisierung der Basisklassen (rekursiv)
   in der Reihenfolge der Deklaration
3. Initialisierung der Data Members (rekursiv)
   in der Reihenfolge der Deklaration
4. Aufruf des Ctor-Body
5. Abbau eines Objekts: in umgekehrter Reihenfolge

```
class Car : public Vehicle {
private:
  int weight;
  int year;
};
```

Constructor (Ctor)

- Member Function(s) mit Klassenname als Methodenname
- Werden Bei Erzeugung eine Objekts aufgerufen
- *Body* des Ctor wird als letzter Schritt aufgerufen
- Data Members sind dann bereits initialisiert!
- Kann nicht (ohne Tricks) separat aufgerufen werden

```
class Car {
public:
  Car(int initWeight) { weight = initWeight; }
private:
  int weight;
};
```

Initializer List

- Basisklassen und Data Members werden *vor* Ausführung des Ctor-Body initialisiert!
- Passiert "automatisch", wenn nicht anders angegeben:
  - Baissklassen und Members werden mit *Default Ctor* initiatlisiert
  - Fehler, wenn kein Default-Ctor exisitert!
- Lösung: angeben, *wie* (mit welchen Argumenten) diese Ctor aufgerufen werden sollen: Initializer List
- Nach einem `:` *vor* dem Ctor Body, Elemente durch `,` getrennt

```
class Car : public Vehicle {
public:
  Car(int initWeight, int yearBuilt): Vehicle(yearBuilt), weight(initWeight) { }
private:
  int weight;
};
```

Destruktor (Dtor)

- Member Function(s) mit `~Klassenname` als Methodenname
- Keine Parameter
- Wird Bei Abbau eine Objekts aufgerufen
- Dtor wird als erster Schritt aufgerufen
- Kann prinzipiell separat aufgerufen werden (sollte aber nicht)
- Wichtiger Mechanismus für viele “Tricks” wie Smart Pointer

```
class Car {
public:
  ~Car() { cout << "I'm leaving" << endl; }
};
```

Kopieren von Objekten

- Vorgabe, wenn nichts Eigenes definiert: Element-wise copy
- Beim Kopieren eines Objekts wird jedes Data Member separat in das korrespondierende Data Member des Zielobjekts kopiert
- ggf. rekursives Kopieren
- Kopieren erfolgt häufiger als gedacht!
  z.B. bei Zuweisungen oder bei Parameterübergabe oder -rückgabe ohne Pointer oder Referenz

Spezielle Constructoren

Diese Constructors werden vom Compiler generiert, wenn die Klasse sie nicht selbst definiert:

```
class Car {
public:
  Car(); // default CTOR
  Car(Car &carToCopy); // Copy Ctor
};
Car car1; // default Ctor
Car car2(car1);  // copy Ctor
Car car2 = car2; // auch copy Ctor (sieht nur wie Zuweisung aus!)
```

- Wird ein Ctor selbst definiert, so generiert der Compiler `keinen` default Ctor mehr.
- Wird der copy Ctor selbst definiert, so wird kein copy Ctor generiert.

Copy Assignment Operator

Eine Klasse kann selbst definieren, wie eine Zuweisung zweier Objekte der Klasse funktioniert:

```
class Car {
public:
  Car & operator=(Car &carToCopy) {this->weight = carToCopy.weight;}
};
Car car1, car2;
Car car1 = car2; // hier eine echte Zuweisung, weil car1 schon existiert hat
```

- Wird der Copy Assignment Operator nicht selbst definiert, so wird vom Compiler einer generiert, der Element-wise Copy aller Data Members durchführt

Canonical Class Form

Standardform einer Klasse mit allen Elementen, die generiert werden, wenn sie nicht vorhanden sind:

```
class Car {
public:
   // default ctor
   Car();

   / destructor
   ~Car();

   // copy ctor
   Car(const Car &otherCar);

   // copy assignment operator
   Car & operator=(const Car &other Car);
};
```