CS-202

C++ Classes & Dynamic Memory

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 7th Project Deadline is this Wednesday 4/3.

- PASS Sessions held Friday-Sunday-&-Monday-Tuesday, get all the help you need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

Today's Topics

Classes & Dynamic Memory

Dynamically Allocated Class Members

Dynamically Allocated Class Instances (Objects)

Class Memory Management

- Constructor(s)
- Destructor(s)
- Assignment

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myFunc(){
 int intVar = 0;
 cout << ++intVar;
}
```

Block-Scope local variable with `auto` Storage-Duration

```
int main(){
 myFunc();
 …
 cout << ++intVar;
}
```

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myFunc(){
 int intVar = 0;
 cout << ++intVar;
}
```

Block-Scope local variable with auto Storage-Duration

```
int main(){
 myFunc();
 ...
 cout << ++intVar;
}
```

Stack Frame pop'ed, variable Out-of-Scope.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myFunc(){
 int intVar = 0;
 cout << ++intVar;
}

int main(){
 myFunc();
 …
 cout << ++intVar;
}
```

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member variable, bound to (an) Object.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}
```

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

➢ As within a block scope, much of memory management is auto-handled.

```
void myFunc(){
 int intVar = 0;
 cout << ++intVar;
}
```

```
int main(){
 myFunc();
 …
 cout << ++intVar;
}
```

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member variable, bound to (an) Object.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}
```

```
int main(){
 myClassFunc();
 …
 cout << ++ mC .getIntVar();
}
```

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myFunc(){
 int intVar = 0;
 cout << ++intVar;
}

int main(){
 myFunc();
 …
 cout << ++intVar;
}
```

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member variable, bound to (an) Object.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}
```

What happens here?

```
int main(){
 myClassFunc();
 …
 cout << ++ mC.getIntVar();
}
```

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myFunc(){
 int intVar = 0;
 cout << ++intVar;
}

int main(){
 myFunc();
 …
 cout << ++intVar;
}
```

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member variable, bound to (an) Object.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}
```

What happens here?

```
int main(){
 myClassFunc();
 …
 cout << ++ mC .getIntVar();
}
```

Stack Frame of `myClassFunc` pop'ed, Object **mC** is Out-of-Scope.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}


int main(){
 myClassFunc();
 …
 cout << ++ mC .getIntVar();
}
```

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member *Variable*, bound to (an) Object.

- auto Storage-Duration.
- Goes away with Stack Frame that created it, Stack Frame that created the Class Object.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}
```

Class Constructor called as Block-Scope local variable is declared.

```
int main(){
 myClassFunc();
 …
 cout << ++ mC .getIntVar();
}
```

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member *Variable*, bound to (an) Object.

- `auto` Storage-Duration.
- Goes away with Stack Frame that created it, Stack Frame that created the Class Object.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntVar(0);
 cout << ++mC.getIntVar();
}
```

Class Constructor called as Block-Scope local variable is declared.

```
int main(){
 myClassFunc();
 …
 cout << ++ mC .getIntVar();
}
```

Class Destructor was called as Object went Out-of-Scope.

```
class MyClass {
 public:
  int getIntVar() const;
  void setIntVar(int);
 private:
  int m_intVar;
};
```

Class Member *Variable*, bound to (an) Object.

- **auto** Storage-Duration.
- Goes away with Stack Frame that created it, Stack Frame that created the Class Object.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
const int DFLT_ARR[ARRAY_MAX] = {1,2,3};
void myClassFunc(){
 MyClass mC;
 mC.setIntArr(DFLT_ARR);
 cout << *mC.getIntArr();
}


int main(){
 myClassFunc();
 …
 cout << * mC .getIntArr();
}
```

```
class MyClass {
 public:
  const int * getIntArr() const;
  void setIntArr(const int *);
 private:
  int m_intArr[ARRAY_MAX];
};
```

- `auto` Storage-Duration *Array*.
- Created-allocated with the Class Object.
- Goes away-deallocated with the Class Object.
- No need to handle Memory Management.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
const int DFLT_ARR[ARRAY_MAX] = {1,2,3};
void myClassFunc(){
 MyClass mC;
 mC.setIntArr(DFLT_ARR);
 cout << *mC.getIntArr();
}
```

Class Constructor called as Block-Scope, *Array* member is guaranteed to be *Allocated* (Note: POD case).

```
int main(){
 myClassFunc();
 …
 cout << * mC .getIntArr();
}
```

```
class MyClass {
 public:
  const int * getIntArr() const;
  void setIntArr(const int *);
 private:
  int m_intArr[ARRAY_MAX];
};
```

- `auto` Storage-Duration *Array*.
- Created-allocated with the Class Object.
- Goes away-deallocated with the Class Object.
- No need to handle Memory Management.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
const int DFLT_ARR[ARRAY_MAX] = {1,2,3};
void myClassFunc(){
 MyClass mC;
 mC.setIntArr(DFLT_ARR);
 cout << *mC.getIntArr();
}
```

Class Constructor called as Block-Scope, *Array* member is guaranteed to be *Allocated* (Note: Array of POD case).

```
int main(){
 myClassFunc();
 …
 cout << * mC .getIntArr();
}
```

Class Destructor was called as Object went Out-of-Scope, *Array* member is guaranteed to be *Deallocated* (Note: Array of POD case).

```
class MyClass {
 public:
  const int * getIntArr() const;
  void setIntArr(const int *);
 private:
  int m_intArr[ARRAY_MAX];
};
```

- `auto` Storage-Duration *Array*.
- Created-allocated with the Class Object.
- Goes away-deallocated with the Class Object.
- No need to handle Memory Management.

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntArr(DFLT_ARR);
 cout << *mC.getIntArr();
}


int main(){
 myClassFunc();
 …
 cout << * mC .getIntArr();
}
```

```
class MyClass {
 public:
  const int * getIntArr() const;
  void setIntArr(const int *);
 private:
  int * m_intArr;
};
```

A Pointer can be used to point to Dynamically Allocated memory.

- Pointer *Variable* created with the Class Object.
- Pointer *Variable* goes away with the Class Object.
- **Need to handle Dynamic Memory it points to.**

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntArr(DFLT_ARR);
 cout << *mC.getIntArr();
}
```

Class Constructor called as Block-Scope local *Variable* is only guaranteed to be *Defined*.

```
int main(){
 myClassFunc();
 …
 cout << * mC .getIntArr();
}
```

```
class MyClass {
 public:
  const int * getIntArr() const;
  void setIntArr(const int *);
 private:
  int * m_intArr;
};
```

A Pointer can be used to point to Dynamically Allocated memory.

- Pointer *Variable* created with the Class Object.
- Pointer *Variable* goes away with the Class Object.
- **Need to handle Dynamic Memory it points to.**

Classes can wrap Dynamic Memory Management

Member Variables are wrapped by the Class.

- As within a block scope, much of memory management is auto-handled.

```
void myClassFunc(){
 MyClass mC;
 mC.setIntArr(DFLT_ARR);
 cout << *mC.getIntArr();
}
```

Class Constructor called as Block-Scope local *Variable* is only guaranteed to be *Defined.*

```
int main(){
 myClassFunc();
 …
 cout << * mC .getIntArr();
}
```

Class Destructor was called as Object went Out-of-Scope, local *Variable* is destroyed, but not the memory it points to.

```
class MyClass {
 public:
  const int * getIntArr() const;
  void setIntArr(const int *);
 private:
  int * m_intArr;
};
```

A Pointer can be used to point to Dynamically Allocated memory.

- Pointer *Variable* created with the Class Object.
- Pointer *Variable* goes away with the Class Object.
- **Need to handle Dynamic Memory it points to.**

Dynamically Allocated Class Members

Class Constructor (ctor) – the case until now:

- No Dynamic Storage-Duration members.
- Constructor mainly for controlled Member initialization.
- Its presence is tentative, Class can be initialized via a combination of setEngTiming, getChassis.

Class Destructor (dtor) – the case until now:

- Nothing to do, **auto** Storage-Duration Members get deallocated when the Class Object is destroyed.
- An Object's Destructor when called, invokes the Destructor of all its Member Objects.

```
class Car  {
 public:
 Car();
 Car(const Chassis & chass,
    const double engT[VLV]=DFT_TIM);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * e);
 const double * getEngT() const;
 Chassis & getChassis();
 protected:
  Chassis m_chassis;
 private:
  double m_engTiming[VLV];
};
```

Dynamically Allocated Class Members

Class Constructor (**ctor**) – the case until now:

- No Dynamic Storage-Duration members.
- Constructor mainly for controlled Member initialization.
- Its presence is tentative, Class can be initialized via a combination of **setEngT**iming, **getChassis**.

Class Destructor (**dtor**) – the case until now:

- Nothing to do, **auto** Storage-Duration Members get deallocated when the Class Object is destroyed.
- An Object's Destructor when called, invokes the Destructor of all its Member Objects; In effect:
- Frees memory of **m_engineTiming** known-size array.
- Calls **dtor** of ***m_chassis***.

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
     const double engT[VLV]=DFT_TIM);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
  Chassis m_chassis;
 private:
  double m_engTiming[VLV];
};
```

Dynamically Allocated Class Members

Class Constructor (`ctor`) – with Dynamic Memory:

- A Raw Pointer member can be used to point to a memory location with data, but this has to be allocated.
- Otherwise: *Undefined Behavior* (best case scenario is a clear Segmentation Fault !)
- Typically, `ctor` might perform initial allocation.

Class Destructor (`dtor`) – with Dynamic Memory:

- An Object's `dtor` when called, invokes the `dtor` of all its Member Objects.
- But a Raw Pointer member is just a variable, if it points to Dynamic Memory it has to be explicitly Deallocated.
- Otherwise: Memory Leak !

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Dynamically Allocated Class Members

Class Constructor (`ctor`) – with Dynamic Memory:

- A Raw Pointer member can be used to point to a memory location with data, but this has to be allocated.
- Otherwise: *Undefined Behavior* (best case scenario is a clear Segmentation Fault !)
- Typically, `ctor` might perform initial allocation.

Class Destructor (`dtor`) – with Dynamic Memory:

- An Object's `dtor` when called, invokes the `dtor` of all its Member Objects.
- But a Raw Pointer member is just a variable, if it points to Dynamic Memory it has to be explicitly Deallocated.
- Otherwise: Memory Leak !

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Classes & Dynamic Memory

Dynamically Allocated Class Members

Class Constructor (ctor)

Parametrized Constructor:

```
Car::Car(const Chassis & chass,
         const double * engT, int numVlv){
  m_chassis = car.m_chassis; //Chassis assignment op/tor
  //dynamic memory allocation at instantiation
  m_numValve = numVlv;                    // assigns size !
  m_engTiming = new double[m_numValve];   // allocates !
```

Note: What will happen here in case an Exception is thrown?

```
}
```

Default Constructor:

```
Car::Car(){
  //Chassis object auto-created based on its default ctor
  //initialization of dynamic array size, no allocation
  m_numValve = 0;
  m_engTiming = NULL;
}
```

```
class Car  {
public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
protected:
  Chassis m_chassis;
private:
  double * m_engTiming;
  int m_numValve;
};
```

Classes & Dynamic Memory

Dynamically Allocated Class Members

Class Constructor (ctor)

Default Constructor (revisited):

```
Car::Car(){
  m_numValve = 0;        // initializes size !
  m_engTiming = NULL;    // initializes pointer !
}
```

Default Constructor – Bad

(E.g. similar if attempting to invoke Parametrized ctor with 0-size)

```
Car::Car(){
  m_numValve = 0;
  m_engTiming = new double[m_numValve];
}
```

Note:

- "When the value of the expression in a direct-new-declarator is zero, the allocation function is called to allocate an array with no elements."
- "The effect of dereferencing a pointer returned as a request for zero size is *Undefined*."
- "Even if the size of the space requested by new is 0, the request can fail."

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Dynamically Allocated Class Members

Class Constructor (ctor)

Copy-Constructor:

```
Car::Car(const Car & car){
  m_chassis = car.m_chassis; //Chassis assignment op/tor
  //dynamic memory allocation at instantiation
  m_numValve = car.m_numVlv;
  m_engTiming = new double[m_numValve];
  for (int i=0; i<m_numValve; ++i)
    m_engTiming[i] = car.m_engTiming[i];
}
```

Allocate & *Deep*-Copy

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Dynamically Allocated Class Members

Class Constructor (ctor)

Copy-Constructor:

```
Car::Car(const Car& car){
  m_chassis = car.m_chassis; //Chassis assignment op/tor
  //dynamic memory allocation at instantiation
  m_numValve = car.m_numVlv;
  m_engTiming = new double[m_numValve];
  for (int i=0; i<m_numValve; ++i)
    m_engTiming[i] = car.m_engTiming[i];
}
```

Allocate & *Deep*-Copy

Remember: The compiler automatically-synthesized one behaves like:

```
Car::Car(const Car & car){
  m_chassis = car.chass;  //ok (?)
  m_numValve = car.m_numVlv;  //ok
  m_engTiming = car.m_engTiming; //same memory (!)
}
```

Shallow-Copy

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Dynamically Allocated Class Members

Class Destructor (dtor)

Syntax:

```
Car::~Car(){
  //Chassis object auto-destroyed (its dtor called)
  //dynamic memory deallocation
  delete [] m_engTiming;
  //further cleanup ? – NO(…)
  m_engTiming = NULL;
  m_numValve = 0;
}
```

Necessary

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double* getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Dynamically Allocated Class Members

Class Destructor (dtor)

Syntax:

```
Car::~Car(){
  //Chassis object auto-destroyed (its dtor called)
  //dynamic memory deallocation
  delete [] m_engTiming;
  //further cleanup ? – NO(…)
  m_engTiming = NULL;
  m_numValve = 0;
}
```

Necessary

Not always a good idea

Remember: Destructor is last thing called before object lifetime ends:

- No sense incurring set overhead (think cases where 1000's of Objects are allocated/deallocated).
- Usually mentioned rationale of setting Pointers back to **NULL** suggests it as a safeguard mechanism, i.e. "what if my code tries to access memory after it's been deleted?"

```
class Car  {
public:
 Car();
 Car(const Chassis & chass,
    const double * engT, int numVlv);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
protected:
  Chassis m_chassis;
private:
 double * m_engTiming;
 int m_numValve;
};
```

Dynamically Allocated Class Members

Class Destructor (dtor)

Example:

```
Car * myCar_Pt = new Car();
myCar_Pt->~Car();
const double * deletedData_Pt = myCar_Pt->getEngT();
delete myCar_Pt;
```

a) Explicit Destructor Call deletes Object data.

b) What is the result in this "intermediate" state?

c) Object deletion completely frees Object memory.

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double* engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Note :
We are not calling **delete** on it whose side-effects we saw in Lectures 16 & 17, but invoking the class **dtor** !

Side-Note:
The previous are mostly used when exploiting Memory Re-use with the *Placement* **new** expression …

Dynamically Allocated Class Members

Class Destructor (dtor)

Example:

```
Car * myCar_Pt = new Car();
myCar_Pt->~Car();
const double * deletedData_Pt = myCar_Pt->getEngT();
delete myCar_Pt;
```

a) Explicit Destructor Call deletes Object data.

b) What is the result in this "intermediate" state?

c) Object deletion completely frees Object memory.

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double* engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

- After non-trivial Destructor called, Object "… no longer exists".
- "... after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways... The program has undefined behavior if ... the pointer is used to **access a non-static data member** or **call a non-static member function** of the object.
- Also, when trying to help with Debugging, it is better to annotate with characteristic values, e.g. `m_engTiming = 0xDEADBEEF;`

Class Objects in Dynamic Memory

Dynamically Allocated Class Object

Example:

```
Chassis superCarChassis( … );
Car * myCar_Pt = new Car(superCarChassis, dfltTims, 24);
double superCarTimings[24] = {…,…,…};
myCar_Pt->setEngT(superCarTimings);
myCar_Pt->getChassis().setColor(…);

delete myCar_Pt;
```

Everything as per usual Pointer notation.

Expression **new** - Invocation of Constructor:

- Functionally behaves as per the usual, allocating **auto** Storage-Duration members automatically and Dynamic Memory Members as instructed.

But is in itself entirely a Dynamically Allocated Variable:

- All its Members will be allocated on the Heap, regardless if they are "regular" objects or dynamically allocated variables.

```
class Car {
public:
 Car();
 Car(const Chassis & chass,
   const double * engT, int numVlv);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
protected:
  Chassis m_chassis;
private:
 double * m_engTiming;
 int m_numValve;
};
```

Class Objects in Dynamic Memory

Class Object

Example:

```
Chassis superCarChassis( … );

Car * myCar_Pt = new Car(superCarChassis, dfltTims, 24);

double superCarTimings[24] = {…,…,…};
myCar_Pt->setEngT(superCarTimings);
myCar_Pt->getChassis().setColor(…);

delete myCar_Pt;
```

Everything as per usual Pointer notation.

Expression **delete** - Invocation of Destructor:

- Functionally behaves the same, deallocating **auto** Storage-Duration members automatically and Dynamic Memory Members as instructed.

Remember:

- Calls Destructors of all Member Variables.
- Has to be explicitly instructed to delete any Dynamically Allocated space, will not work "magically" - recursively.

```
class Car  {
 public:
  Car();
  Car(const Chassis & chass,
    const double * engT, int numVlv);
  Car(const Car & car);
  ~Car();
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
 protected:
   Chassis m_chassis;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Class Objects in Dynamic Memory

By-Example (further)

- *Remember*: Composition
  Class contains Object of another Class type: Allocation / deallocation automatically handled.
  `Chassis m_chassis;`
- *Remember*: Aggregation
  Class references external Object via Pointer of another Class type. Memory Management for Object (might be) externally handled.
  `Driver * m_driver;`
- Class employs Dynamic Data. Memory Management handled in Class Methods.
  `double * m_engTiming;`

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

auto Storage-Duration.

Necessary for Object to be complete

Not a prerequisite

Necessary and NOT auto Storage-Duration

Classes Dynamic Memory Management

Class Object Destructor(s)

Destructor Automatic activation clauses:

- Global Object or Object with `static` Storage-Duration (Namespace-Scope), when program terminates:

a)
```
namespace ns{
  Car myGlobalCar;
  …
}
```

b)
```
{ …
  static Car myStaticCar;
  …
}
```

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object Destructor(s)

Destructor Automatic activation clauses:

- Global Object or Object with `static` Storage-Duration (Namespace-Scope), when program terminates:

a)
```
namespace ns{
    Car myGlobalCar;
    …
}
```

b)
```
{ …
    static Car myStaticCar;
    …
}
```

- Local Object (Block-Scope), when it goes Out-of-Scope:

c)
```
{
    Car myLocalCar;
    …
}
```

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object Destructor(s)

Destructor Automatic activation clauses:

- Global Object or Object with `static` Storage-Duration (Namespace-Scope), when program terminates:

a)
```
namespace ns{
   Car myGlobalCar;
   …
}
```

b)
```
{  …
   static Car myStaticCar;
   …
}
```

- Local Object (Block-Scope), when it goes Out-of-Scope:

c)
```
{
   Car myLocalCar;
   …
}
```

- Pointer to Object of Dynamic Storage-Duration gets `delete`'d:

d)
```
Car * myCar_Pt = new Car();
…
delete myCar_Pt;
```

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object *Copy*-Constructor(s)

Copy Constructor activation clauses:

- Explicit activation – instantiate an Object by making use of another Object – *Reminder*: (b) is Copy-Initialization :

a)
```
Car simpleCar(simpleChassis, 16);
Car myCar_cpy( simpleCar );
Car myCar_cpyInit = simpleCar;
```

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object *Copy*-Constructor(s)

Copy Constructor activation clauses:

➢ Explicit activation – instantiate an Object by making use of another Object – *Reminder*: (b) is Copy-Initialization :

a)
```
Car simpleCar(simpleChassis, 16);
Car myCar_cpy( simpleCar );
```
b)
```
Car myCar_cpyInit = simpleCar;
```
➢ Function **returns** Object By-Value:

c)
```
Car makeCar(const Driver * driver){
  Car tempCar;
  tempCar->setDriver( driver );
  return tempCar;
}
```

Local Object created

Copy-**ctor** to **return** value can be "*Elided*" but behavior is similar

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object *Copy*-Constructor(s)

Copy Constructor activation clauses:

- Explicit activation – instantiate an Object by making use of another Object – *Reminder*: (b) is Copy-Initialization :

a)
```
Car simpleCar(simpleChassis, 16);
Car myCar_cpy( simpleCar );
```
b)
```
Car myCar_cpyInit = simpleCar;
```

- Function **returns** Object By-Value:

c)
```
Car makeCar(const Driver * driver){
  Car tempCar;
  tempCar->setDriver( driver );
  return tempCar;
}
```
Local Object created

Copy-ctor to return value can be "*Elided*" but behavior is similar

- Function parameter is an Object passed By-Value:

d)
```
bool carInspector(Car car){
  const double * engT = car.getEngT();
  if ( … ){ return true; } else { return false; }
}
```
Copy-ctor to pass value

```
class Car  {
 public:
 Car();
 Car(Chassis chass, int numVlv=16);
 Car(const Car & car);
 ~Car();
 void setEngT(const double * engT);
 const double * getEngT() const;
 Chassis & getChassis();
 void setDriver(const Driver * d);
 protected:
 Chassis m_chassis;
 Driver * m_driver;
 private:
 double * m_engTiming;
 int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object Assignment

Remember: Default Object Assignment is *Shallow*-Copy.

- Dynamic Memory allocation/deallocation requires overloading of the assignment operator (**=**):

```
Car & Car::operator=(const Car & other){
m_chassis = other.m_chassis;
if (other.m_engTiming && …){ // possible checks
 delete [] m_engTiming;
 m_engTiming = new double[other.m_numValve];
 for(…){ m_engTiming[i]=other.m_engTiming[i]; }
}
if (other.m_driver){ // not NULL pointer
 // how do we want this? depends!
 delete m_driver;
 m_driver = new Driver(other.m_driver);
}
 return *this;
}
```

```
class Car  {
 public:
  Car();
  Car(Chassis chass, int numVlv=16);
  Car(const Car & car);
  ~Car();
  Car & operator=(const Car & other);
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
  void setDriver(const Driver * d);
 protected:
  Chassis m_chassis;
  Driver * m_driver;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object Assignment

The special case of Self-Assignment:

What if?

```
Car * car_Pt = new Car(simpleChassis, 16);
Car * anotherCar_Pt = car_Pt;
*anotherCar_Pt = *car_Pt;
```

Self-Assignment

```
Car & Car::operator=(const Car & other){
 …
 delete [] m_engTiming;
 if (other.m_numValve>0 && other.m_engTiming){
   m_numValve = other.m_numValve;
   try{
     m_engTiming = new double[other.m_numValve];
     for(…){ m_engTiming[i] = other.m_engTiming[i]; }
   } catch(...) { /* handle exception */ }
 } else{ m_numValve = 0; m_engTiming = NULL; }
 …
 return *this;
}
```

Deletes its own content, and then re-copies own newly allocated data.

```
class Car  {
 public:
  Car();
  Car(Chassis chass, int numVlv=16);
  Car(const Car & car);
  ~Car();
  Car & operator=(const Car & other);
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
  void setDriver(const Driver * d);
 protected:
  Chassis m_chassis;
  Driver * m_driver;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

Classes Dynamic Memory Management

Class Object Assignment

The special case of Self-Assignment:

Remember !

```
Car * car_Pt = new Car(simpleChassis, 16);
Car * anotherCar_Pt = car_Pt;
*anotherCar_Pt = *car_Pt;

Car & Car::operator=(const Car & other){
 if (this != &other){
   delete [] m_engTiming;
   if (other.m_numValve>0 && other.m_engTiming){
     m_numValve = other.m_numValve;
     try{
       m_engTiming = new double[other.m_numValve];
       for(…){ m_engTiming[i] = other.m_engTiming[i]; }
     } catch(...) { /* handle exception */ }
   } else{ m_numValve = 0; m_engTiming = NULL; }
 }
 return *this;
}
```

Check if calling object is the same as the one passed as parameter !

```
class Car  {
 public:
  Car();
  Car(Chassis chass, int numVlv=16);
  Car(const Car & car);
  ~Car();
  Car & operator=(const Car & other);
  void setEngT(const double * engT);
  const double * getEngT() const;
  Chassis & getChassis();
  void setDriver(const Driver * d);
 protected:
  Chassis m_chassis;
  Driver * m_driver;
 private:
  double * m_engTiming;
  int m_numValve;
};
```

CS-202

Time for Questions !