CS-202

C++ Classes – Constructor(s) (Pt.1)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 3nd Project will be announced today Thursday 9/14.

2nd Project Deadline was this Wednesday 9/13.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!

Today's Topics

C++ Classes Cheatsheet

- Declaration
- Members, Methods, Interface
- Implementation – Resolution Operator (`::`)
- Instantiation – Objects
- Object Usage – Dot Operator (`.`)
- Object Pointer Usage – Arrow Operator (`->`)
- Classes as Function Parameters, Pass-by-Value, by-(`const`)-Reference, by-Address
- Protection Mechanisms – `const` Method signature
- Classes – Code File Structure

Constructor(s)

Destructor

Class Cheatsheet

Declaration:

```
class Car {
    public:
    float AddGas(float gallons);
    float GetMileage();
    char m_licensePlates[9];
    protected:
    float m_gallons;
    float m_mileage;
    private:
    bool SetEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Class (Type) Name

- Type Name is up to you to declare!

- Members in Brackets
- Semicolon

Conventions:

- Begin with Capital letter.
- **`mixedCase`** for phrases.
- General word for Class of Objects.

Class Cheatsheet

Declaration:

```
class Car  {
   public:
    float AddGas(float gallons);
    float GetMileage();
    char m_licensePlates[9];
   protected:
    float m_gallons;
    float m_mileage;
   private:
    bool SetEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Access Specifiers

- Provide Protection Mechanism

Encapsulation - Abstraction:

- "Data Hiding"

Class Cheatsheet

Declaration:

```
class Car  {
   public:
    float AddGas(float gallons);
    float GetMileage();
    char m_licensePlates[9];
   protected:
    float m_gallons;
    float m_mileage;
   private:
    bool SetEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Member Variables

- All necessary Data inside a single Code Unit.

Conventions:

- Begin with **m_<variable_name>**.

Encapsulation - Abstraction:

- Abstract Data Structure

Class Cheatsheet

Declaration:

```
class Car  {
    public:
    float AddGas(float gallons);
    float GetMileage();
    char m_licensePlates[9];
    protected:
    float m_gallons;
    float m_mileage;
    private:
    bool SetEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Member Function / Class Methods

- All necessary Data & Operations inside a single Code Unit.

Conventions:

- Capitalize first letter.

Encapsulation - Abstraction:

- Abstract Data Structure

Class Cheatsheet

Usual-case Class Interface Design:

```
class Car  {
   public:
   float AddGas(float gallons);
   float GetMileage();
   bool SetEngineTiming(double[16]);
   private:
   char m_licensePlates[9];
   float m_gallons;
   float m_mileage;
   double m_engineTiming[16];
};
```

`public` Class Interface:
- Class Methods

`private` Class Access:
- Class Data

Class Interface to Member Data should "go through" Member Functions.

Class Cheatsheet

Class Implementation:

```
class Car  {
    …
    bool AddGas(float gallons);
    float GetMileage();
};

float Car::AddGas(float gallons){
  /* actual code here */
}

float Car::GetMileage(){
 /* actual code here */
}
```

An Implementation *needs* to exist for Class Methods

Scope Resolution Operator

(::)

- Indicates which Class Method this definition implements.

Class Cheatsheet

Class Instantiation - Implicit:

```
<type_name> <variable_name>;
```

```
Car myCar;
```

Object

Create (Construct) a variable of specific Class type.

Will employ "*Default Constructor*"

- Compiler will auto-handle *Member Variables*' initialization !

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Usage:

```
<variable_name>.<member_name>;
```

Dot Operator – Member-of

(.)

- Which Object this Member references.

```
Car myCar;
float mileage = myCar.getMileage();
strcpy(myCar.m_licensePlates,"Gandalf");
```

Member Variables & Member Functions

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Pointers:

```
<type_name>* <variable_name_Pt>;

Car myCar;

Car* myCar_Pt;

myCar_Pt = &myCar;
(*myCar_Pt).GetMileage();
```

Object

Pointer to Object

- Dereferencing to get to Object.
  Works the same as any pointer.

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Pointer Usage:

```
<variable_name_Pt>-><member_name>;
```

Arrow Operator – Member-access (->)

- Structure (Class) Pointer Dereference

```
Car myCar;
Car* myCar_Pt = &myCar;

myCar_Pt->GetMileage();
strcpy(myCar_Pt->m_licensePlates,"Gandalf");
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Pointer Usage:

```
<variable_name_Pt>-><member_name>;
```

Arrow Operator – Member-access (->)

- Structure (Class) Pointer Dereference

Why?
Chaining Operator Precedence (. , ->)

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

```
(*(*(*topClass).subClass).subSubClass).method();
topClass->subClass->subSubClass->method();
```

Class Cheatsheet

Class Object in Function – By-Value:

```
Car myCar;
strcpy(myCar.m_licensePlates,"Gandalf");
printCapPlatesMileage(myCar);
cout << myCar.m_licensePlates;

void printCapPlatesMileage(Car car){
  char* lP = car.m_licensePlates;
  while (*lP = toupper(*lP)){ ++lP; }
  cout << car.m_licensePlates << endl;
  cout << car.GetMileage() << endl;
}
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:
Will work with Local Object Copy !

Class Cheatsheet

Class Object in Function – By-Reference:

```
Car myCar;
strcpy(myCar.m_licensePlates,"Gandalf");
printModifyCapPlates(myCar);
cout << myCar.m_licensePlates;

void printModifyCapPlates(Car& car){
  char* lP = car.m_licensePlates;
  while (*lP = toupper(*lP)){ ++lP; }
  cout << car.m_licensePlates << endl;
}
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:

Will modify Object Data !

Class Cheatsheet

Class Object in Function – By-`const`-Reference:

```
Car myCar;
strcpy(myCar.m_licensePlates,"Gandalf");
printCapPlates(myCar);
cout << myCar.m_licensePlates;

void printCapPlates(const Car& car){
  char* lP = (char*)malloc(sizeof(
                car.m_licensePlates));
   strcpy(lP,car.m_licensePlates);

   char* lP_0 = lP;
   while (*lP = toupper(*lP)){ ++lP; }

   cout << lP_0 << endl;
}
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:
Not allowed to modify Object Data !

Class Cheatsheet

Class Object in Function – By-Address:

```
Car myCar;
Car* myCar_Pt = &myCar;
strcpy(myCar_Pt->m_licensePlates,"Gandalf");
printModifyCapPlates(myCar_Pt);
cout << myCar.m_licensePlates;

void printModifyCapPlates(Car* car_Pt){
  char* lP = car_Pt->m_licensePlates;
  while (*lP = toupper(*lP)){ ++lP; }
  cout << car_Pt->m_licensePlates
       << endl;
}
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage();
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:

Will modify Object Data !

Class Cheatsheet

Protection Mechanisms – `const` Method signature:

A "promise" that Method doesn't modify Object

```
Car myCar;
cout << myCar.GetMileage() << endl;
cout << myCar.AddGas(10.0F) << endl;

float Car::GetMileage() const {
  return m_mileage;
}
float Car::AddGas(float gallons) {
  if (m_gallons += gallons > MAX_GALLONS)
    m_gallons = MAX_GALLONS;
  return m_gallons;
}
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Protection Mechanisms – Access Specifiers:

`public`

Anything that has access to a ***`Car`*** Object (scope-wise) also has access to all **`public`** Member Variables and Functions.

- "Normally" used for Functions.
- Need to have at least one **`public`** Member.

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Protection Mechanisms – Access Specifiers:

`private`

Members (Variables and Functions) that can ONLY be accessed by Member Functions of the ***`Car`*** Class.

- Cannot be accessed in `main()`, in other files, or by other functions.

- If not specified, Members default to **`private`**.
- Should specify anyway – good coding practices!

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Protection Mechanisms – Access Specifiers:

protected

Members that can be accessed by:

- Member Functions of the ***Car*** Class.
- Member Functions of any *Derived* Class.

A *Derived* Class

```
class Hybrid : Car  {
  …
  float GasToElectricRatio();
};

float Hybrid::GasToElectricRatio(){
    if (m_gallons < …){ return …; }
}
```

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Member Functions – Accessors ("Getters")

Name starts with `Get`, ends with Member name.
Allows retrieval of non-`public` Data Members.

```
float Car::GetMileage() const {
  return m_mileage;
}
```

Note: Don't generally take in arguments.

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Member Functions – Mutators ("Setters")

Name starts with `Set`, ends with Member name.
Controlled changing of non-`public` Data Members.

```
bool Car::SetEngineTiming(double t_in[16]){
  for (int i=0;i<16;++i){
    if (tin[i]<… || tin[i]>…){ return false; }
  }
  for (int i=0;i<16;++i){
    m_engineTiming[i]=tin[i];
  }
  return true;
}
```

Note: In simple case, don't `return` anything (`void`).
In controlled setting, return success/fail (`bool`).

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Member Functions – Facilitators ("Helpers")

Provide support for the Class's operations.

```
float Car::AddGas(float gallons) {
  if (m_gallons += gallons > MAX_GALLONS)
    m_gallons = MAX_GALLONS;
  return m_gallons;
}
```

Note:
`public` if generally called outside Function.
`private`/`protected` if only called by Member Functions.

```
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons;
  float m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Classes and Code File Structure

Class Header File: `Car.h`

```
#ifndef CAR_H
#define CAR_H
#define NUMVALVES 16
class Car  {
  public:
  float AddGas(float gallons);
  float GetMileage() const ;
  char m_licensePlates[9];
  protected:
  float m_gallons, m_mileage;
  private:
  bool SetEngineTiming(double[16]);
  double m_engineTiming[NUMVALVES];
};
#endif
```

Class Source File: `Car.cpp`

```
#include <iostream>
#include <Car.h>

#define MAX_GALLONS 20.0

float Car::GetMileage() const {
  return m_mileage;
}
float Car::AddGas(float gallons) {
  if (m_gallons += gallons > MAX_GALLONS)
    m_gallons = MAX_GALLONS;
  return m_gallons;
}
bool Car::SetEngineTiming(double t_in[16]){
  for (int i=0;i<16;++i){
        if (tin[i]<… || tin[i]>…) return false;
  }
  for (int i=0;i<16;++i){
    m_engineTiming[i]=tin[i];
  }
  return true;
}
```

Class Cheatsheet

Classes and Code File Structure

Program File: `car_prog.cpp`

```
#include <iostream>
#include <…>
#include <Car.h>
int main(){
   Car myCar;
   Car* myCar_Pt = &myCar;

   strcpy(myCar_Pt->m_licensePlates,"Gandalf");
   printCapPlates(myCar_Pt);
   cout << myCar.m_licensePlates;

   cout << myCar.GetMileage() << endl;
   cout << myCar.AddGas(10.0F) << endl;
   return 0;
}
```

Constructor(s)

Description

Special Class Methods that "build" an Object.

- Object Initialization.
- Supply default values (If necessary).

Remember:

Implicit initialization (*Default* Constructor)

- Automatically called when an object is created.

Implicit: `Date myDate;`

Explicit: `Date myDate(1,1,1917);`

```
class Date{
  public:
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Description

Called when a Class is *Instantiated*.

- C++ won't automatically initialize Member Variables.

Default Constructor:

- Basic no-argument constructor, can have one or none in a Class.
- If Class has no Constructors, the C++ Compiler will *make* a *Default*.

Overloaded Constructors:

- Constructors that take in arguments, can have none or many in a Class.
- Appropriate version called based on number and type of arguments passed when an Object is created (*Instantiated*).

Constructor(s)

Syntax - General

Syntax:

- For Function Prototype:

```
<class_name>(…);
Date(…);
```

- For Function Definition:

```
<class_name>::<class_name>(…){
  /* class_name constructor code */
}
Date::Date(…){
  /* Date constructor code */
}
```

Note: A Special Function!

- Has no `return` type.
- Has same name as Class.

```
class Date{
  public:
  Date();
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Syntax – The Default *ctor*

Default (empty) Constructor:

- Function Prototype:

```
Date();
```

- Function Definition:

```
Date::Date(){
  /* default constructor code */
}
```

Note:

- The compiler will (implicitly) provide a *Default* Constructor if none is specified.

Note: A Special Function!

- Has no **return** type.
- Has same name as Class.

```
class Date{
  public:
  Date();
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Syntax – The Default *ctor*

Default (empty) Constructor:

- Function Prototype:

```
Date();
```

- Function Definition:

```
Date::Date(){
   m_month = 1;
   m_day = 1;
   m_year = 1917;
   PrintDay();
   /* or even no code at all? */
}
```

Note: A Special Function!

- Has no **return** type.
- Has same name as Class.

```
class Date{
  public:
  Date();
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Syntax – The Parametrized `ctor`

Overloaded (parametrized) Constructor:

- Function Prototype:

```
Date(int month, int day, int year);
```

- Function Definition:

```
Date::Date(int month, int day, int year){
  m_month = month;
  m_day = day;
  m_year = year;
  PrintDay();
}
```

Note: A Special Function!

- Has no `return` type.
- Has same name as Class.

```
class Date{
  public:
  Date(int month,
    int day, int year);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Implementations

Overloaded (parametrized) Constructor Definition:

- Simple.

```
Date::Date(int month, int day, int year){
  m_month = month;
  m_day = day;
  m_year = year;
}
```

Missing: Technically, nothing, but...

- Validation of the information being passed in!

```
class Date{
  public:
  Date(int month,
    int day, int year);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Implementations

Overloaded (parametrized) Constructor Definition:

- Controlled.

```
Date::Date(int month, int day, int year){
  if (month>0 && month<=12) {
    m_month = month; }
  else { m_month = 1; }
  if (day>0 && day<=31) {
    m_day = day; }
  else { m_day = 1; }
  if (year>=1917 && year<=2017) {
    m_year = year; }
  else { m_year = 1; }
}
```

```
class Date{
  public:
  Date(int month,
    int day, int year);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Implementations

Overloaded (parametrized) Constructor Definition:

- Controlled /w better coding.

```
Date::Date(int month, int day, int year){
  if (month>0 && month<=MAX_MONTH) {
    m_month = month; }
  else { m_month = 1; }
  if (day>0 && day<=MAX_DAY) {
    m_day = day; }
  else { m_day = 1; }
  if (year>=MIN_YEAR && year<=MAX_YEAR) {
    m_year = year; }
  else { m_year = 1; }
}
```

```
class Date{
  public:
  Date(int month,
    int day, int year);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Implementations

Overloaded (parametrized) Constructor Definition (Controlled /w even better coding):

```
Date::Date(int month, int day, int year){
   SetMonth(month);
   SetDay(day);
   SetYear(year);
}
```

Why?

- Allows Code Re-Use !

```
class Date{
  public:
  Date(int month,
    int day, int year);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Overloading

Can have multiple versions of the Constructor:

- *Overloading* the Constructor.

Different constructors for when:

- All Member values are known.
- No Member values are known.
- Some subset of Member values are known.

Note:

- If you define a Constructor with arguments you have to define a *Default* (empty) constructor.
- A good coding practice to always define a 1-liner Default (empty) Constructor.

```
class Date{
  public:
   Date();
   Date(int month,
     int year);
   Date(int month,
     int day, int year);
   void SetMonth(int m);
   void SetDay(int d);
   void SetYear(int y);
   void PrintDay() const;
   void ShiftNextDay();
  private:
   int m_month;
   int m_day;
   int m_year;
};
```

Constructor(s)

Overloading

Can have multiple versions of the Constructor:

- *Overloading* the Constructor.

Constructor invoked with full range of arguments:

- Constructor to set user-supplied Member values.

```
Date::Date(int month, int day, int year){
  SetMonth(month);
  SetDay(day);
  SetYear(year);
}
```

```
class Date{
  public:
  Date();
  Date(int month,
    int year);
  Date(int month,
    int day, int year);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Overloading

Can have multiple versions of the Constructor:

- *Overloading* the Constructor.

Constructor invoked with no arguments:

- Constructor to set default Member values.

```
Date::Date(){
   SetMonth(DEFAULT_MONTH);
   SetDay(DEFAULT_DAY);
   SetYear(DEFAULT_YEAR);
}
```

```
class Date{
  public:
   Date();
   Date(int month,
     int year);
   Date(int month,
     int day, int year);

   void SetMonth(int m);
   void SetDay(int d);
   void SetYear(int y);
   void PrintDay() const;
   void ShiftNextDay();
  private:
   int m_month;
   int m_day;
   int m_year;
};
```

Constructor(s)

Overloading

Can have multiple versions of the Constructor:

- *Overloading* the Constructor.

Constructor invoked with some arguments:

- Constructor to set some user-supplied Member values, the rest set to default Member values.

```
Date::Date(int month, int year){
  SetMonth(month);
  SetDay(DEFAULT_DAY);
  SetYear(year);
}
```

```
class Date{
  public:
  Date();
  Date(int month,
    int year);
  Date(int month,
    int day, int year);

  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Overloading Caveats

Consider 2 Overloaded Constructor versions:

```
Date::Date(int month, int year){
  SetMonth(month);
  SetDay(DEFAULT_DAY);
  SetYear(year);
}
Date::Date(int month, int day){
  SetMonth(month);
  SetDay(day);
  SetYear(DEFAULT_YEAR);
}
```

```
class Date{
  public:
  Date();
  Date(int month,
    int year);
  Date(int month,
    int day);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Overloading Caveats

Consider 2 Overloaded Constructor versions:
What the Compiler "sees":

```
Date::Date(int month, int year){ /* … */ }
```

```
Date(int , int);
```

Function Prototype Signature

```
Date::Date(int month, int day){ /* … */ }
```

```
Date(int , int);
```

Function Prototype Signature

```
class Date{
  public:
  Date();
  Date(int month,
    int year);
  Date(int month,
    int day);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Default Parameters

Not really meaningful to have numerous Constructors just to set default Member values.

- A lot of code duplication.
- Can set Default Parameter values in Constructor.

Function Prototype Syntax:

```
Date(int month, int day=DFLT_D,
     int year=DFLT_Y, bool printFlg=false);
```

Note:

- Use constants !
- Only Change in Constructor Prototype !

```
class Date{
  public:
  Date();
  Date(int month,
   int day=DFLT_D,
   int year=DFLT_Y,
   bool printFlg=false);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Default Parameters

Not really meaningful to have numerous Constructors just to set default Member values.

- A lot of code duplication.
- Can set Default Parameter values in Constructor.

Function Prototype Syntax (NO!) :

```
Date(int month=DFLT_M, int day=DFLT_D,
     int year=DFLT_Y, bool printFlg=false);
```

```
Date();
```

Attention: Same Prototype Signature !

- This is still ambiguous!

```
Date d;
```

```
error: call of overloaded 'Date()'
       is ambiguous
```

```
class Date{
  public:
  Date();
  Date(int month=DFLT_M,
   int day=DFLT_D,
   int year=DFLT_Y,
   bool printFlg=false);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Default Parameters

Function Implementation Syntax:

```
Date::Date(int month, int day, int year,
           bool printFlg){
  SetMonth(month);
  SetDay(day);
  SetYear(year);
  if (printFlg){ PrintDay(); }
}
```

Note:

- Function implementation doesn't change !
- If parameters are not provided, they will be set to Prototype Default Parameters.

```
class Date{
  public:
  Date();
  Date(int month,
   int day=DFLT_D,
   int year=DFLT_Y,
   bool printFlg=false);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Default Parameters

Call with Default Parameters:

```
Date defaultCtorDate;
```

Default Constructor call (no parameter list)

```
Date myBDayPrinted(5, 15, 1985, true);
Date myBDay(5, 15, 1985);
Date halloween(10, 31);
Date july(4);
// defaultDate:     4196816/0/4196304
// myBDayPrinted:   5/15/1985
// myBDay:          5/15/1985
// halloween:       10/31/1917
// july:            4/1/1917
```

Output: 5,15,1985

```
class Date{
  public:
   Date();
   Date(int month,
    int day=DFLT_D,
    int year=DFLT_Y,
    bool printFlg=false);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Default Parameters

Call with Default Parameters – Caveats:

- Sequential Interpretation of Default Parameters in Constructor Prototype:

```
Date(int month, int day=DFLT_D,
     int year=DFLT_Y, bool printFlg=false);
```

No skipping Parameters! Can only do:

```
Date myFullDatePrinted(2, 9, 2017, true);
```
or
```
Date myFullDate(2, 9, 2017);
```
or
```
Date myMonthDayOnly(2, 9);
```
or
```
Date myMonthOnly(2);
```

```
class Date{
  public:
  Date();
  Date(int month,
   int day=DFLT_D,
   int year=DFLT_Y,
   bool printFlg=false);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructor(s)

Syntax – The Copy `ctor`

Copy (class-object) Constructor:

- For Function Prototype:

```
Date(const Date &);
```

- For Function Definition:

```
Date::Date(const Date &date){
  /* Date date object-const-ref copy code */
}
```

Note:

- The compiler will (implicitly) provide a *Copy* Constructor if none is specified.

```
class Date{
  public:
  Date();
  Date(int month,
   int day=DFLT_D,
   int year=DFLT_Y,
   bool printFlg=false);
  Date(const Date &);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month, m_day,
      m_year;
};
```

Constructor(s)

Implementations

Copy (class-object) Constructor Definition:

- Copy-over Member Data.

```
Date::Date(const Date &date){
  m_month = date.m_month;
  m_day = date.m_day;
  m_year = date.m_year;
}
```

Same Class:

- Access to `private` Member Data of input Object.

```
class Date{
  public:
  Date();
  Date(int month,
   int day=DFLT_D,
   int year=DFLT_Y,
   bool printFlg=false);
  Date(const Date &);
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  void PrintDay() const;
  void ShiftNextDay();
  private:
  int m_month, m_day,
      m_year;
};
```

Destructor(s)

Description

Called when a Class goes out-of-scope or is freed from the heap (by `delete`).

- Not necessary in simple cases.
- But, have to take care of Cleaning-Up resources that won't automatically go away.

Destructor

- Has the name `~ClassName()`, has no `return` value.
- Can have one or none in a Class.

Destructor will automatically (without writing any code in it) call Destructor of any Data Member Objects.

- But NOT Data Member Pointers !
- Define a Destructor if you need to return resources, deallocate pointer memory, etc.

Designing a Class

Ask yourself:

- What properties must each Object have, what data types should each be?
- Which should be `private`? Which should be `public`?
- What operations must each Object have?
- What Accessors, Mutators, Facilitators?
- What parameters must each of these have?
- `const`, by-Value, by-Reference, Default?
- What `return` value should each of these have?

Rules of thumb:

- Data usually `private`, operations usually `public`.
- Ususally 1 Mutator & 1 Accessor per Data Member.

CS-202

Time for Questions !