Note on Programming Style

Please note that the example codes included in this lecture note do *not* follow some of the guidelines set forth in the **Programming Style Guide**.

For instance, most of the codes are devoid of comments, **{** and **}** may not appear on separate lines, and more than one statement may appear on one single line of code.

This is done in a presentation environment due to lack of space, to minimize cluttering, and so on.

In writing actual programs (homework assignments included), therefore, students should refer to the Programming Style Guide and *not* the lecture notes for style guidelines.

Modified example using C **`struct`**

`date.h`
(interface)

```
#ifndef DATE_H
#define DATE_H

#include <iostream>

struct Date
{
  int day;
  char *month; /* pointer to dynamic array of just the right size
                  to store month's name as C-string */
  int year;
};

void Initialize(Date& d); // for initialization to some valid state
void CleanUp(Date& d);    // for release of dynamic memory
void SetDay(Date& d, int dd);
void SetMonth(Date& d, char *mm);
void SetYear(Date& d, int yy);
void ShowDate(std::ostream& out, const Date& d);

#endif
```

Modified example using C `struct`

`date.cpp`
(implementation)

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "date.h"
using namespace std;

void Initialize(Date& d)
{ d.day = 1; d.month = 0; d.year = 1; }

void CleanUp(Date& d)
{ delete [] d.month; }

void SetDay(Date& d, int dd)
{
   if (dd >= 1 && dd < 32)
      d.day = dd;
   else
      cerr << "SetDay() error: invalid day." << endl;
}
```

(*continued*)

Modified example using C `struct`

`date.cpp`
(implementation)

```
void SetMonth(Date& d, char *mm)
{
   if (mm == 0)
   {
      delete [] d.month;
      d.month = 0;
      return;
   }

   char *charPtr = new(nothrow) char [strlen(mm) + 1];
   if (charPtr == 0)
   {
      cerr << "SetMonth() error: \"new\"-failure." << endl;
      return;
   }

   strcpy(charPtr, mm);
   delete [] d.month;
   d.month = charPtr;
}
```

(*continued*)

Modified example using C **struct**

date.cpp
(implementation)

```
void SetYear(Date& d, int yy)
{
   if (yy > 0)
      d.year = yy;
   else
      cerr << "SetYear() error: invalid year." << endl;
}

void ShowDate(ostream& out, const Date& d)
{
   if (d.month == 0)
      out << d.day << " (undefined) " << d.year;
   else
      out << d.day << ' ' << d.month << ' ' << d.year;
}
```


Modified example using C **struct**

prog.cpp
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
  Date today;

  Initialize(today); // puts today in some valid state
  SetDay(today, 13);
  SetMonth(today, "September");
  SetYear(today, 1999);
  cout << "Today's date is ";
  ShowDate(cout, today);
  cout << endl;
  CleanUp(today);    // frees up dynamically allocated memory

  return EXIT_SUCCESS;
}
```

Modified example using `class` (#1)

`date.h`
(interface)

```
#ifndef DATE_H
#define DATE_H

#include <iostream>

class Date
{
private:
   int day; char *month; int year;

public:
   void Initialize();
   void CleanUp();
   void SetDay(int dd);
   void SetMonth(char *mm);
   void SetYear(int yy);
   void ShowDate(std::ostream& out) const;
};

#endif
```

```
void Initialize(Date& d);
void CleanUp(Date& d);
void SetDay(Date& d, int dd);
void SetMonth(Date& d, char *mm);
void SetYear(Date& d, int yy);
void ShowDate(std::ostream& out, const Date& d);
```

What else do you see that are different in these functions (compared to the C **`struct`** *version)?*

Modified example using `class` (#1)

`date.cpp`
(implementation)

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "date.h"
using namespace std;

void Date::Initialize()
{ day = 1; month = 0; year = 1; }

void Date::CleanUp()
{ delete [] month; }

void Date::SetDay(int dd)
{
   if (dd >= 1 && dd < 32)
      day = dd;
   else
      cerr << "SetDay() error: invalid day." << endl;
}
```

These are actually short for

```
this->day = 1;
this->month = 0;
this->year = 1;
delete [] this->month;
this->day = dd;
```

respectively

The **`this`** *pointer is provided* ***transparently*** *by C++ to every member function (except* ***static member functions****). It points to the* ***object that invokes the member function****.*

(continued)

Modified example using `class` (#1)

`date.cpp`
(implementation)

```
void Date::SetMonth(char *mm)
{
   if (mm == 0)
   {
      delete [] month;
      month = 0;
      return;
   }

   char *charPtr = new(nothrow) char [strlen(mm) + 1];
   if (charPtr == 0)
   {
      cerr << "SetMonth() error: \"new\"-failure." << endl;
      return;
   }

   strcpy(charPtr, mm);
   delete [] month;
   month = charPtr;
}
```

(*continued*)

Modified example using `class` (#1)

`date.cpp`
(implementation)

```
void Date::SetYear(int yy)
{
   if (yy > 0)
      year = yy;
   else
      cerr << "SetYear() error: invalid year." << endl;
}

void Date::ShowDate(ostream& out) const
{
   if (month == 0)
      out << day << " (undefined) " << year;
   else
      out << day << " " << month << " " << year;
}
```

Modified example using `class` (#1)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
  Date today;

  today.Initialize();
  today.SetDay(13);
  today.SetMonth("September");
  today.SetYear(1999);
  cout << "Today's date is ";
  today.ShowDate(cout);
  cout << endl;
  today.CleanUp();

  return EXIT_SUCCESS;
}
```

member functions *(or* ***methods****)*
versus
free functions *(or* ***ordinary functions****)*

Corresponding statements used in the C **`struct`** *version (for comparison):*

```
Initialize(today);
SetDay(today, 13);
SetMonth(today, "September");
SetYear(today, 1999);
cout << "Today's date is ";
ShowDate(cout, today);
cout << endl;
CleanUp(today);
```

What don't you like about this version of **`class`** *implementation of* **`Date`***?*

Modified example using `class` (#1)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
  Date today;

  today.Initialize();
  today.SetDay(13);
  today.SetMonth("September");
  today.SetYear(1999);
  cout << "Today's date is ";
  today.ShowDate(cout);
  cout << endl;
  today.CleanUp();

  return EXIT_SUCCESS;
}
```

What a pain to have to call **`Initialize()`** *each time I declare a* **`Date`** *object!*
Would it bomb if I forget to call it?
Is there no way that I can initialize a **`Date`** *object to some desired value when I declare it?*
(like what I am so used to doing in **`int i = 0;`***)*
(perhaps in this case it would be something like **`Date today(13, "September", 1999);`***)*

Constructors to the rescue ...

- A ***constructor*** is a *special member function* of a class that is used to *initialize* objects of the class as they are created
  - using supplied argument(s) or default value(s)
- A constructor is *automatically* called
  - each time a class object is created (comes into existence)
- A constructor is *syntactically different* from other member functions of the class in that
  - it has the *same name as the class*
  - it has *no return type* (not even **`void`**)

Overloading Constructors

A special constructor called *Default Constructor*

- Constructors *can be overloaded*
  - to provide a variety of ways for initializing class objects
- A ***default constructor*** is a constructor that
  - has *no parameters* (*i.e.*, parameterless)
  - is called when a class object is declared without supplying any arguments, as in

```
Date today;
```

Overloading Constructors

Another special constructor called *Copy Constructor*

- (Repeat:) Constructors *can be overloaded*
  - to provide a variety of ways for initializing class objects
- A ***copy constructor*** is a constructor that
  - has *exactly ONE parameter* AND the *data type of the parameter* is the *same as the constructor's class*
  - is called when a class object is declared with an existing object of the same class being supplied as the only argument, as in
    `Date copyOfToday(today);`
    `Date copyOfToday = today;`
    assuming `today` *is an existing* `Date` *object*
    (these are ***not*** the only possible forms of this type of statement)
  - is also called whenever a program generates copies of an object (*e.g.*, when an *object is passed by value* to a function, when a function *returns a copy of an object*, or when temporary copies of an object are generated for other reasons)

Overloading Constructors

Other constructors ("non-special")

- (Repeat:) Constructors *can be overloaded*
  - to provide a variety of ways for initializing class objects
- Constructors bearing other *different signatures*
  - may be defined if desirable
- For instance, for our `Date` data type, it may be desirable to also define the following "non-special" constructors:
  - a one-parameter (for `day`) constructor
  - a two-parameter (for `day` and `month`) constructor
  - a three-parameter (for `day`, `month` and `year`) constructor

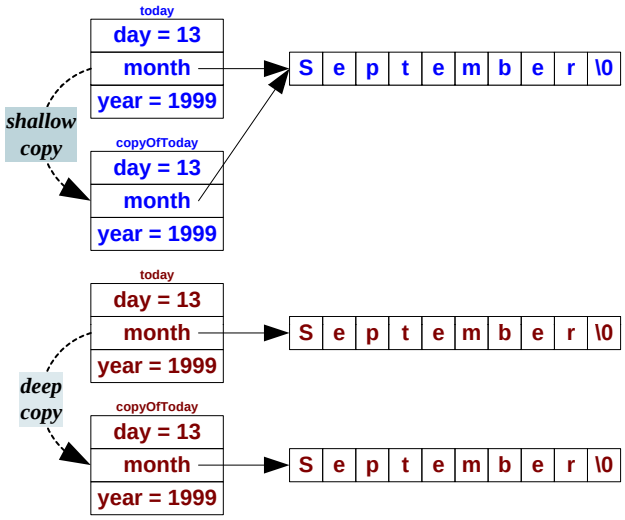
Automatic Default Constructor

- If we *don't define **any** constructor* for a class
  - the compiler will automatically generate a default constructor
  - that *does nothing*
- If we define one or more constructors with parameter(s)
  - we should also define a default (parameterless) constructor
  - because the compiler ***no longer will*** supply an automatic default constructor
  - *i.e.*, it's a *you-do-it-all-or-you-totally-don't-do-it-at-all* deal
- In general, we should define our own default constructor
  - because the compiler-generated default constructor will in general does nothing to help us initialize objects to some *consistent* (*i.e.* valid) state

Automatic Copy Constructor

- If we *don't define a copy constructor* for a class
  - the compiler will automatically supply a copy constructor
  - that initializes a new object by *simply copying the values (bit by bit) of each of the data members of the existing object into the corresponding data members of the new object*
    ➔ this is called ***memberwise copying*** or ***shallow copying***
- For many classes, shallow copying is sufficient
  - then we don't have to define copy constructors for them
- For some classes (usually those that contain data members that are *pointers pointing to dynamically allocated memory*), shallow copying is insufficient and ***deep copying*** is needed
  - then we must define our own copy constructors for them

Is shallow copying sufficient for our **`Date`**?



We can provide the following overloaded constructors (shown only as prototypes) for our **`Date`**

```
Date();                          //default constructor
Date(int dd);                    //1-parameter constructor
Date(int dd, char *mm);          //2-parameter constructor
Date(int dd, char *mm, int yy); //3-parameter constructor
Date(const Date& d);             //copy constructor
```

*The parameter for the copy constructor cannot be passed by **value**. (Why?)*
*It "can" be passed by **reference** but should be passed by **const reference**. (Why?)*

We can also use C++'s "*function-with-default-arguments*" feature to simplify our task, by replacing

```
Date();                           //default constructor
Date(int dd);                     //1-parameter constructor
Date(int dd, char *mm);           //2-parameter constructor
Date(int dd, char *mm, int yy); //3-parameter constructor
Date(const Date& d);              // copy constructor
```

with

```
Date(int dd = 1, char *mm = 0, int yy = 0);
Date(const Date& d);              // copy constructor
```

assuming (*1*, *null address*, *0*) are our intended defaults for (**`day`**, **`month`**, **`year`**), respectively

We can write our constructors for **`Date`** as

```
Date::Date(int dd, char *mm, int yy)
{
   if (dd >= 1 && dd < 32)
   { day = dd; }
   else
      cerr << "Date() error: invalid day." << endl;

   if (mm == 0)
   { month = 0; }
   else
   {
      char *charPtr = new(nothrow) char [strlen(mm) + 1];
      if (charPtr == 0)
         cerr << "Date() error: \"new\"-failure." << endl;
      else
      { strcpy(charPtr, mm); month = charPtr; }
   }

   if (yy > 0)
   { year = yy; }
   else
      cerr << "SetYear() error: invalid year." << endl;
}
```

… and

```
Date::Date(const Date& d)
{
   day = d.day;
   year = d.year;

   if (d.month == 0)
   {
      month = 0;
      return;
   }

   char *charPtr = new(nothrow) char [strlen(d.month) + 1];
   if (charPtr == 0)
   {
      cerr << "Date() error: \"new\"-failure." << endl;
      return;
   }
   strcpy(charPtr, d.month);
   month = charPtr;
}
```


But using the ***initializer list*** is better (why?), so

```
Date::Date(const Date& d) : day(d.day), year(d.year)
{
   if (d.month == 0)
   {
      month = 0;
      return;
   }
   char *charPtr = new(nothrow) char [strlen(d.month) + 1];
   if (charPtr == 0)
   {
      cerr << "Date() error: \"new\"-failure." << endl;
      return;
   }
   strcpy(charPtr, d.month);
   month = charPtr;
}
```

*We must use the **initializer list** syntax to initialize a class member that is a*
(1) **`const`**
*(2) **reference** to another class member*

UPSHOT *Use an **initializer list** (instead of statements in function body) if possible unless there are other considerations that dictate otherwise.*

PITFALL *The **initializer** list syntax can **only** be used in **constructors**.*

Modified example using `class` (#1)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
  Date today;

  today.Initialize();
  today.SetDay(13);
  today.SetMonth("September");
  today.SetYear(1999);
  cout << "Today's date is ";
  today.ShowDate(cout);
  cout << endl;
  today.CleanUp();

  return EXIT_SUCCESS;
}
```

Constructors to the rescue here!

What else don't you like about this version of **`class`** *implementation of* **`Date`***?*

Modified example using `class` (#1)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
  Date today;

  today.Initialize();
  today.SetDay(13);
  today.SetMonth("September");
  today.SetYear(1999);
  cout << "Today's date is ";
  today.ShowDate(cout);
  cout << endl;
  today.CleanUp();

  return EXIT_SUCCESS;
}
```

Isn't it just as painful to have to call **`CleanUp()`** *each time I am done using a* **`Date`** *object? And I bet you something bad will likely happen if I forget to call* **`CleanUp()`** *where I am supposed to! If* **`CleanUp()`** *represents what its name implies,that some "house-keeping chore" needs to be performed after I am done using a* **`Date`** *object, can you not make the computer do it for me automatically?*

Destructors to the rescue ...

- A ***destructor*** is a *special member function* of a class that is used to perform *any cleanup processing* that is needed when objects of the class go out of scope (existence)
- An object's destructor is *automatically* called
  - when the object goes out of scope (existence)
- A destructor is *syntactically different* from other member functions of the class in that
  - it has the *same name as the class* that is preceded by ~ (tilde)
  - it has *no return type* (not even **`void`**)
- There can only be ONE destructor for a class
  - *i.e.*, destructors *cannot* be overloaded
  - and there should only be ONE way to destroy an object
- All destructors are *parameterless*

Compiler-Supplied Destructor

- If we *don't define a destructor* for a class
  - the compiler will automatically supply one
  - that *does nothing*
- For many classes, there is no cleanup processing needed when objects of the classes go out of scope (existence)
  - then we don't have to define any destructors
  - because the compiler-supplied destructors are sufficient
- In general, we should define a destructor if we are defining a *class that contains data members that directly make use of dynamic memory*
  - in fact, the *primary function* (where applicable) of *destructors* is simply to *release (free) dynamic memory*

Our **Date** has a data member **month**
that directly makes use of dynamic memory
so we need to define a destructor for the class

```
Date::~Date()
{
   delete [] month;
}
```

It should not be surprising that the function body of the destructor is identical to that of **Cleanup()** since they are meant to perform the same function

The *big difference* between them is that the destructor is called *automatically* but **Cleanup()** is not

Modified example using **class** (#2)

date.h
(interface)

```
#ifndef DATE_H
#define DATE_H

#include <iostream>

class Date
{
private:
   int day; char *month; int year;

public:
   Date(int dd = 1, char *mm = 0, int yy = 1);
   Date(const Date& d);
   ~Date();
   void SetDay(int dd);
   void SetMonth(char *mm);
   void SetYear(int yy);
   void ShowDate(std::ostream& out) const;
};

#endif
```

Modified example using `class` (#2)

`date.cpp`
(implementation)

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "date.h"
using namespace std;

Date::Date(int dd, char *mm, int yy)
{
   SetDay(dd);   //calling another member function
   SetYear(yy);  //calling another member function
   // Cannot simply call SetMonth() due to "delete [] month;" in same
   if (mm == 0) { month = 0; return; }
   char *charPtr = new(nothrow) char [strlen(mm) + 1];
   if (charPtr == 0)
   {
      cerr << "SetMonth() error: \"new\"-failure." << endl;
      return;
   }
   strcpy(charPtr, mm);
   month = charPtr;
}
```

Whenever we need a block of code identical to the body of an existing member function, we'd want to make a call to that function instead of reproducing the block of code. Why?

Do you see why **`delete [] month;`** *in* **`SetMonth()`** *will cause problem?*

(continued)

Modified example using `class` (#2)

`date.cpp`
(implementation)

```
Date::Date(const Date& d) : day(d.day), year(d.year)
{
   if (d.month == 0) { month = 0; return; }
   char *charPtr = new(nothrow) char [strlen(d.month) + 1];
   if (charPtr == 0)
   {
      cerr << "Date() error: \"new\"-failure." << endl; return;
   }
   strcpy(charPtr, d.month);  month = charPtr;
}

Date::~Date() { delete [] month; }

void Date::SetDay(int dd)
{
   if (dd >= 1 && dd < 32)
      day = dd;
   else
      cerr << "SetDay() error: invalid day." << endl;
}
```

(continued)

Modified example using `class` (#2)

`date.cpp`
(implementation)

```
void Date::SetMonth(char *mm)
{
  if (mm == 0) {delete [] month; month = 0; return; }
  char *charPtr = new(nothrow) char [strlen(mm) + 1];
  if (charPtr == 0)
  {cerr << "SetMonth() error: \"new\"-failure." << endl; return;}
  strcpy(charPtr, mm); delete [] month; month = charPtr;
}

void Date::SetYear(int yy)
{
  if (yy > 0) { year = yy; }
  else
  { cerr << "SetYear() error: invalid year." << endl; }
}

void Date::ShowDate(ostream& out) const
{
  if (month == 0) { out << day << " <null> " << year; }
  else { out << day << " " << month << " " << year; }
}
```


Modified example using `class` (#2)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
   Date defaultDate, today(13, "September", 1999), copyOfToday(today);

   cout << "Default date is ";
   defaultDate.ShowDate(cout);
   cout << endl;
   cout << "Today's date is ";
   today.ShowDate(cout);
   cout << ",\nwhich should be identical to ";
   copyOfToday.ShowDate(cout);
   cout << endl;

   return EXIT_SUCCESS;
}
```

There's still something that's not right with this implementation of **`Date`**.
Can you see it?
(See next slide for a clue.)

Modified example using `class` (#2)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
   Date defaultDate, today(13, "September", 1999), copyOfToday;
   copyOfToday = today;

   cout << "Default date is ";
   defaultDate.ShowDate(cout);
   cout << endl;
   cout << "Today's date is ";
   today.ShowDate(cout);
   cout << ",\nwhich should be identical to ";
   copyOfToday.ShowDate(cout);
   cout << endl;

   return EXIT_SUCCESS;
}
```

Clue to another flaw with this implementation of **`Date`**.
(How is this program different from the one in the last slide?)

Modified example using `class` (#2)

`prog.cpp`
(application)

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
   Date defaultDate, today(13, "September", 1999), copyOfToday = today;
   // Date defaultDate, today(13, "September", 1999), copyOfToday(today);
   // Date defaultDate, today(13, "September", 1999), copyOfToday;
   // copyOfToday = today;

   cout << "Default date is ";
   defaultDate.ShowDate(cout);
   cout << endl;
   cout << "Today's date is ";
   today.ShowDate(cout);
   cout << ",\nwhich should be identical to ";
   copyOfToday.ShowDate(cout);
   cout << endl;

   return EXIT_SUCCESS;
}
```

By the way, how does this one compare?

Textbook Readings

- Chapter 2
  - Section 2.2
  - Page2 53-4 (Value Semantics and Copy Constructor)
  - Pages 62-64 (Default Arguments/Default Constructor)
- Chapter 4
  - Page 170-171 (The Destructor)
  - Section 4.4