CIS 190: C/C++ Programming

Classes in C++
Outline

• Header Protection
• Functions in C++
• Procedural Programming vs OOP
• Classes
  – Access
  – Constructors
Headers in C++

• done same way as in C

• including user “.h” files:
  ```
  #include "userFile.h"
  ```

• including C++ libraries
  ```
  #include <iostream>
  ```
An example

typedef struct bar{
  int a;
} BAR;

#include "bar.h"

typedef struct foo{
  BAR x;
  char y;
} FOO;

#include "bar.h"

int main()
{
  BAR i;
  FOO j;

  /* ... */

  return 0;
}

main.c
An example

```c
typedef struct bar{
    int a;
} BAR;

#include "bar.h"

#include "foo.h"

typedef struct foo{
    BAR x;
    char y;
} FOO;

int main()
{
    BAR i;
    FOO j;
    /* ... */
    return 0;
}
```

when we try to compile this…

```c
typedef struct bar{
    int a;
} BAR;

#include "bar.h"

#include "foo.h"

typedef struct foo{
    BAR x;
    char y;
} FOO;

#include "bar.h"

#include "foo.h"

int main()
{
    BAR i;
    FOO j;
    /* ... */
    return 0;
}
```
typedef struct bar{
    int a;
} BAR;

#include "bar.h"
#include "foo.h"

int main()
{
    BAR i;
}

In file included from foo.h:1:0,
    from main.c:2:
bar.h:1:16: error: redefinition of 'struct bar'
In file included from main.c:1:0:
bar.h:1:16: note: originally defined here
In file included from foo.h:1:0,
    from main.c:2:
bar.h:3:3: error: conflicting types for 'BAR'
In file included from main.c:1:0:
bar.h:3:3: note: previous declaration of 'BAR' was here
What the compiler is “seeing”

*bar.h*

```c
typedef struct bar{
    int a;
} BAR;

```

*foo.h*

```c
typedef struct bar{
    int a;
} BAR;

typedef struct foo{
    BAR x;
    char y;
} FOO;

```

*main.c*

```c
#include "bar.h"

int main() {
    BAR i;
    FOO j;
    /* ... */
    return 0;
}
```

```c
#include "foo.h"

```
What the compiler is “seeing”

typedef struct bar{
  int a;
} BAR;

bar.h

typedef struct bar{
  int a;
} BAR;

typedef struct foo{
  BAR x;
  char y;
} FOO;

foo.h

typedef struct bar{
  int a;
} BAR;

typedef struct foo{
  BAR x;
  char y;
} FOO;

int main() {
  BAR i;
  FOO j;
  /* ... */
  return 0;
}

main.c
Header Protection

• we want to have the definition of the BAR struct in both:
  – foo.h
  – main.c

• easiest way to solve this problem is through the use of header guards
Header Guards

• in each “.h” file, use the following:

```c
#ifndef BAR_H if not (previously) defined
#define BAR_H then define

[CONTENTS OF .H FILE GO HERE]

#endif /* BAR_H */ stop the “if” at this point (end of the file)
```
A fixed example

typedef struct bar{
    int a;
} BAR;

```c
#include "bar.h"
```

```c
typedef struct foo{
    BAR x;
    char y;
} FOO;

#include "bar.h"
#include "foo.h"

int main()
{
    BAR i;
    FOO j;

    /* ... */

    return 0;
}
```

```c
main.c
```
A fixed example

```c
#include "bar.h"
#include "foo.h"

int main() {
    BAR i;
    FOO j;
    /* ... */
    return 0;
}
```

```c
#define BAR_H
#endif /*BAR_H*/

typedef struct bar{
    int a;
} BAR;

#define FOO_H
#endif /*FOO_H*/

typedef struct foo{
    BAR x;
    char y;
} FOO;

```
Outline

• Header Protection
• Functions in C++
• Procedural Programming vs OOP
• Classes
  – Access
  – Constructors
Functions in C++

• very similar to functions in C
  – variable scope remains the same
  – can still pass things by value, or by reference
    • implicit (arrays) or explicit (pointers)

• a few differences from functions in C
  – no need to pass array length (just use empty brackets)

```cpp
void PrintArray (int arr []);
```
Using `const` in C++ functions

- when used on pass-by-value

```cpp
int SquareNum (int x) {
    return (x * x);
}

int SquareNum (const int x) {
    return (x * x);
}
```
Using `const` in C++ functions

- when used on pass-by-value

- no real difference; kind of pointless
  - changes to pass-by-value variables don’t last beyond the scope of the function

- conventionally: not "wrong," but not done
Using `const` in C++ functions

- when used on pass-by-reference

```c++
void SquareNum (int *x) {
    (*x) = (*x) * (*x); /* fine */
}

void SquareNum (const int *x) {
    (*x) = (*x) * (*x); /* error */
}
```
Using `const` in C++ functions

- when you compile the “const” version:

```cpp
void SquareNum (const int *x) {
    (*x) = (*x) * (*x);  /* error */
}
```

error: assignment of read-only location 'x'
Using `const` in C++ functions

- when used on pass-by-reference

- huge difference
  - prevents changes to variables, even when they are passed in by reference

- conventionally: use for user-defined types (structs, etc.) but don’t use for simple built-in types (int, double, char) except maybe arrays
Outline

• Header Protection
• Functions in C++
• Procedural Programming vs OOP
• Classes
  – Access
  – Constructors
Procedural Programming

• up until now, everything we’ve been doing has been **procedural programming**

• code is divided into multiple procedures
  – procedures operate on data (structures), when given correct number and type of arguments

• examples: PrintTrain(), ReadSingerFile(), DestroyList(), ProcessEvents(), etc.
Object-Oriented Programming

• now that we’re using C++, we can start taking advantage of **object-oriented programming**

• code and data are combined into a single entity called a class
  – each instance of a given class is an **object** of that class type

• OOP is more modular, and more transparent
Outline

• Header Protection
• Functions in C++
• Procedural Programming vs OOP
• Classes
  – Access
  – Constructors
Example: Date Struct

• implementing a date structure in C:

```c
typedef struct date {
    int    month;
    int    day;
    int    year;
} DATE;
```
Example: Date Class

• implementing a date class in C++:

```cpp
class Date {
public:
    int m_month;
    int m_day;
    int m_year;
};
```
Functions in Classes

• let’s add a function to the class that will print out the name of the month, given the number

```cpp
class Date {
public:
    void OutputMonth();
    int m_month;
    int m_day;
    int m_year;
};
```
void OutputMonth();

• nothing is passed in to the function because it only needs to look at the m\_month variable
  – which is a member variable of the Date class
  – just like OutputMonth()
void Date::OutputMonth() {
    switch (m_month) {
    case 1: cout << "January"; break;
    case 2: cout << "February"; break;
    case 3: cout << "March"; break;
    case 4: cout << "April"; break;
    /* etc */
    case 11: cout << "November"; break;
    case 12: cout << "December"; break;
    default:
        cout << "Error in Date::OutputMonth()";
    }
}
void Date::OutputMonth() {
    switch (m_month) {
    case 1:
        cout << "January"; break;
    case 2:
        cout << "February"; break;
    case 3:
        cout << "March"; break;
    case 4:
        cout << "April"; break;
    /* etc */
    case 11:
        cout << "November"; break;
    case 12:
        cout << "December"; break;
    default:
        cout << "Error in Date::OutputMonth()";
    }
}

include class name; more than one class can have a function with the same name.
void Date::OutputMonth() {
    switch (m_month) {
    case 1: cout << "January"; break;
    case 2: cout << "February"; break;
    case 3: cout << "March"; break;
    case 4: cout << "April"; break;
    case 11: cout << "November"; break;
    case 12: cout << "December"; break;
    default: cout << "Error in Date::OutputMonth()";
    }
}
OutputMonth

```cpp
void Date::OutputMonth() {
    switch (m_month) {
    case 1: cout << "January"; break;
    case 2: cout << "February"; break;
    case 3: cout << "March"; break;
    case 4: cout << "April"; break;
    /* etc */
    case 11: cout << "November"; break;
    case 12: cout << "December"; break;
    default:
        cout << "Error in Date::OutputMonth()";
    }
}
```

we can directly access `m_month` because it is a *member variable* of the `Date` class, to which the `OutputMonth()` function belongs.
Using the Date class

```cpp
Date today, birthday;

cout << "Please enter dates as DD MM YYYY" << endl;

// get today’s date
cout << "Please enter today’s date: ";
cin >> today.m_day >> today.m_month >> today.m_year;

// get user’s birthday
cout << "Please enter your birthday: ";
cin >> birthday.m_day >> birthday.m_month
   >> birthday.m_year;

// echo output
cout << "Today’s date is " << today.OutputMonth()
   << today.m_day << ", " << today.m_year << endl;
cout << "Your birthday is " << birthday.OutputMonth()
   << birthday.m_day << ", " << birthday.m_year << endl;
```
Using the Date class

```
Date today, birthday;
```

Variables `today` and `birthday` are instances of the class `Date`. They are both objects of type `Date`.

dates as DD MM YYYY” << endl;
today’s date: “;
today.m_month >> today.m_year;
your birthday: “;

```
cin >> birthday.m_day >> birthday.m_month
    >> birthday.m_year;
```

//echo output
```
cout << “Today’s date is “ << today.OutputMonth()
    << today.m_day << “, “ << today.m_year << endl;
cout << “Your birthday is “ << birthday.OutputMonth()
    << birthday.m_day << “, “ << birthday.m_year << endl;
```
Using the Date class

Date today, birthday;

cout << "Please enter dates as DD MM YYYY" << endl;

// get today’s date
// get today’s date
cout << "Please enter today’s date: ";
cin >> today.m_day >> today.m_month >> today.m_year;

// echo output
// echo output
cout << "Today’s date is ", " " << today.OutputMonth()
    << today.m_day << ", ", " " << today.m_year << endl;
cout << "Your birthday is " << birthday.OutputMonth()
    << birthday.m_day << ", " << birthday.m_year << endl;

when we are not inside the class (as we
were in the OutputMonth() function) we
must use the dot operator to access
today’s member variables
Using the Date class

Date today, birthday;

cout << "Please enter dates as DD MM YYYY" << endl;

// get today’s date
cout << "Please enter today’s date:"
cin >> today.m_day >>
    today.m_month >>
    today.m_year;

// get user’s birthday
cout << "Please enter your birthday:"
cin >> birthday.m_day >>
    birthday.m_month >>
    birthday.m_year;

// echo output
cout << "Today’s date is " << today.OutputMonth() <<
    " " << today.m_month << " " << today.m_day << " "
    << today.m_year << endl;
cout << "Your birthday is " << birthday.OutputMonth() <<
    " " << birthday.m_month << " " << birthday.m_day << " "
    << birthday.m_year << endl;

we also use the dot operator to call the member function
OutputMonth() on the Date object today; again, note that we
do not need to pass in the member variable m_month
Outline

• Header Protection
• Functions in C++
• Procedural Programming vs OOP
• Classes
  – Access
  – Constructors
Public, Private, Protected

• in our definition of the **Date** class, everything was **public** – this is not good practice!

• we have three different options for **access specifiers**, each with their own role:
  – public
  – private
  – protected
Example: Public, Private, Protected

class Date {
public:
    int m_month;
private:
    int m_day;
protected:
    int m_year;
};
Using Public, Private, Protected

• public
  — anything that has access to the `birthday` object also has access to `birthday.m_month`, etc.

• private
  — `m_day` can only be accessed by member functions of the `Date` class; cannot be accessed in `main()`, etc.

• protected
  — `m_year` can by accessed by member functions of the `Date` class and by member functions of any derived classes (we’ll cover this later)
class Date { 
public:
    void OutputMonth();
private:
    int m_month;
    int m_day;
    int m_year;
};
New member functions

• now that \texttt{m\_month}, \texttt{m\_day}, and \texttt{m\_year} are \textit{private}, how do we give them values, or retrieve those values?

• write public member functions to provide indirect, controlled access for the user
New member functions

• **accessor functions:**
  – allow retrieval of private data members
  – `GetMonth()`, `GetDay()`, `GetYear()`

• **mutator functions:**
  – allow changing the value of a private data member
  – `SetMonth()`, `SetDay()`, `SetYear()`

• **service functions:**
  – provide support for the operations
  – `OutputMonth()`
Access specifiers for Date class

class Date {
public:
    void OutputMonth();
    int GetMonth();
    int GetDay();
    int GetYear();
    void SetMonth(int m);
    void SetDay (int d);
    void SetYear (int y);
private:
    int m_month;
    int m_day;
    int m_year;
};
Outline

• Header Protection
• Functions in C++
• Procedural Programming vs OOP
• Classes
  – Access
  – Constructors
Constructors

• special *member functions* used to create (or “construct”) new objects

• automatically called when an object is created

• initializes the values of all data members
Date class Constructors

class Date {
public:
    void OutputMonth();
    Date (int m, int d, int y);
private:
    int m_month;
    int m_day;
    int m_year;
};
class Date {

public:

    void OutputMonth();

    Date (int m, int d, int y);

private:

    int m_month;
    int m_day;
    int m_year;

};

exact same name as the class
Date class Constructors

class Date {
public:
    void OutputMonth();
    Date (int m, int d, int y);

private:
    int m_month;
    int m_day;
    int m_year;
};
Constructor Definition

```cpp
Date::Date (int m, int d, int y)
{
    m_month = m;
    m_day = d;
    m_year = y;
}
```
Constructor Definition

• by using classes with private members and public functions, we can control almost everything

• can prevent “incorrect” values from being accepted by the constructor
Constructor Definition

Date::Date (int m, int d, int y)
{
    if (m > 0 && m <= 12) {
        m_month = m;
    } else { m_month = 1; }
    if (d > 0 && d <= 31) {
        m_day = d;
    } else { m_day = 1; }
    if (y > 0 && y <= 2100) {
        m_year = y;
    } else { m_year = 1; }
}
Overloading

• we can define multiple versions of the constructor – we can overload the function

• different constructors for:
  – when all values are known
  – when no values are known
  – when some subset of values are known
All Known Values

• have the constructor set user-supplied values

```cpp
Date::Date (int m, int d, int y)
{
    m_month = m;
    m_day = d;
    m_year = y;
}
```
All Known Values

• have the constructor set user-supplied values

```cpp
Date::Date (int m, int d, int y)
{
    m_month = m;
    m_day = d;
    m_year = y;
}
```
No Known Values

• have the constructor set all default values

```cpp
Date::Date ()
{
    m_month = 1;
    m_day = 1;
    m_year = 1
}
```
No Known Values

• have the constructor set all default values

```cpp
Date::Date() {
  m_month = 1;
  m_day = 1;
  m_year = 1
}
```

invoked when constructor is called with no arguments
Some Known Values

• have the constructor set some default values

```cpp
Date::Date (int m, int d)
{
    m_month = m;
    m_day = d;
    m_year = 1
}
```
Some Known Values

• have the constructor set some default values

```cpp
Date::Date(int m, int d) {
    m_month = m;
    m_day = d;
    m_year = 1
}
```

invoked when constructor is called with some arguments
Overloaded Date Constructor

• so far we have the following constructors:

    `Date::Date (int m, int d, int y);`
    `Date::Date (int m, int d);`
    `Date::Date ();`

• would the following be a valid constructor?

    `Date::Date (int m, int y);`
Avoiding Multiple Constructors

• defining multiple constructors for different known values is a lot of code duplication

• we can avoid this by setting default parameters in our constructors
Default Parameters

• in the *function prototype* **only**, provide default values you want the constructor to use

```
Date (int m = 3, int d = 6, int y = 2014);
```
Default Parameters

• in the function definition literally nothing changes

```cpp
Date::Date (int m, int d, int y) {
    m_month = m;
    m_day = d;
    m_year = y;
}
```
Using Default Parameters

• the following are all valid declarations:

```java
Date graduation(5, 19, 2014);
Date today;
Date halloween(10, 25);
Date july(4);
```
Using Default Parameters

• the following are all valid declarations:

```java
Date graduation(5,19,2014);
Date today;
Date halloween(10,25);
Date july(4);
```

// graduation: 5/19/2014
// today: 3/6/2014
// halloween: 10/25/2014
// july: 4/6/2014
Using Default Parameters

• the following are all valid declarations:

```java
Date graduation(5, 19, 2014);
Date today;
Date halloween;
Date july(4);

// graduation: 5/19/2014
// today:      3/6/2014
// halloween:  10/25/2014
// july:       4/6/2014
```

NOTE: when you call a constructor with no arguments, you do not give it empty parentheses
Default Constructors

• default constructor is provided by compiler
  – will handle declarations of Date instances

• but, if you create any other constructor, the compiler doesn’t provide a default constructor
  – so make sure you always create a default constructor too, even if its body is just empty