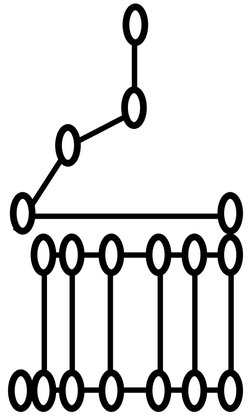
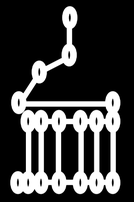
Lecture: Getters, Setters, and Constructors

ENGR 2730:Computers in Engineering





A class is a user-defined type that is a packaged set of data (attributes) **and** functions (behaviors)

```
#include <string>
// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H

// Time class definition
class Time {
public:
   void setTime(int, int, int); // set hour, minute and second
   std::string toUniversalString() const; // 24-hour time format string
   std::string toStandardString() const; // 12-hour time format string
private:
   unsigned int hour{0}; // 0 - 23 (24-hour clock format)
   unsigned int minute{0}; // 0 - 59
   unsigned int second{0}; // 0 - 59
};

#endif
```

Time.h

Classes **encapsulate** implementation details so that a "user" only has to focus on the public **interface**…

...in other words, the implementation details are "hidden" from the user (**information hiding**)

"User" cares about the interface and not the implementation

class "interface"

```
#include <string>
// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H

// Time class definition
class Time {
public:
   void setTime(int, int, int); // set hour, minute and second
   std::string toUniversalString() const; // 24-hour time format string
   std::string toStandardString() const; // 12-hour time format string
private:
   unsigned int hour{0}; // 0 - 23 (24-hour clock format)
   unsigned int minute{0}; // 0 - 59
   unsigned int second{0}; // 0 - 59
};

#endif
```

Time.h

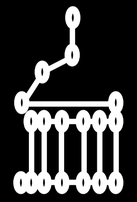
```
#include <iostream>
#include "Time.h"
int main()
{
    Time t1;
    t1.setTime(3,30,45);
    cout << t1.toUniversalTime() << endl;
}
```

instantiated object

class "implementation"

```
// set new Time value using universal time
void Time::setTime(int h, int m, int s) {
   // validate hour, minute and second
   if ((h >= 0 && h < 24) && (m >= 0 && m < 60) && (s >= 0 && s < 60)) {
      hour = h;
      minute = m;
      second = s;
   }
   else {
      throw invalid_argument(
         "hour, minute and/or second was out of range");
   }
}

// return Time as a string in universal-time format (HH:MM:SS)
string Time::toUniversalString() const {
   ostringstream output;
   output << setfill('0') << setw(2) << hour << ":"
      << setw(2) << minute << ":" << setw(2) << second;
   return output.str(); // returns the formatted string
}

// return Time as string in standard-time format (HH:MM:SS AM or PM)
string Time::toStandardString() const {
   ostringstream output;
   output << ((hour == 0 || hour == 12) ? 12 : hour % 12) <<
      << setfill('0') << setw(2) << minute << ":" << setw(2)
      << second << (hour < 12 ? " AM" : " PM");
   return output.str(); // returns the formatted string
}
```

Time.cpp



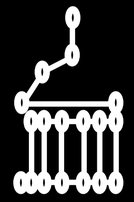
Get and Set Member Functions

```
class Circle{
public:
    // const prevents getRadius from changing m_radius
    double getRadius() const {
        return m_radius;
    }

    // setRadius prevents m_radius being set to a negative number
    void setRadius(double radius) {
        if (radius >= 0){
            m_radius = radius;
        } else {
            m_radius = 0;
        }
    }

private:
    double m_radius;
};
```

```
int main() {
    Circle c1;

    c1.setRadius(-5);
    cout << "radius = " << c1.getRadius() << endl;

    return 0;
}
```

Output:

```
radius = 0
```



Data Hiding/Encapsulation … Validation

Software Engineering Observation 9.6

Making data members `private` *and controlling access, especially write access, to those data members through* `public` *member functions helps ensure data integrity.*

Error-Prevention Tip 9.4

The benefits of data integrity are not automatic simply because data members are made `private`*—you must provide appropriate validity checking.*

Data Member manipulation: Data Validation

- Set functions can be programmed to validate their arguments and reject any attempts to set the data to bad values, such as
  - a negative body temperature
  - a day in March outside the range 1 through 31
  - a product code not in the company's product catalog, etc.
  - An invalid hour/minute/second

```
// Using data member assignment
void Time::setHour(const int h)
{
  // validate hour, minute and second
  if (h >= 0 && h < 24) {
    hour = h;
  }
  else {
    hour = 0;
  }
}
```

Exception Handling Throw Syntax

- In C++ when something “bad” or “exceptional” occurs, the mechanism for providing feedback regarding the exception is called “exception handling”

- You “throw” an exception object to exit the function (and ultimately the program).
- For CIE the throwing of exceptions will cause your programs to fail.
  This is OK for CIE, but not for most applications.

```
#include <stdexcept>

// Using data member assignment
void Time::setHour(const int h)
{
  // validate hour, minute and second
  if (h >= 0 && h < 24) {
    hour = h;
  }
  else {
    throw invalid_argument(
          “hour was out of range");
  }
}
```

Get and Set Member Functions

- Generally, data members are declared private. This is so they don't accidently get set to an invalid value, i.e., month = 13 or radius = -5.
- Private data members are only accessible to class functions. Class functions are responsible keeping data member values valid.
- Private data members need to have "getter" and "setter" class methods to be accessed outside the class.
- Get functions return the value of the data member and are generally prefixed with "get" followed by the name of the data member, i.e., getRadius().
- Set functions set the value of the data member to the input value and are generally prefixed with "set" followed by the name of the data member, i.e., setRadius(radius).
- One important responsibility of a setter method is to ensure that a data member never gets assigned to an invalid value.

Constructor basics

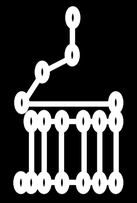
Constructor basics

*A **constructor** is a member function that is automatically called to initialize an object when the object is declared.*

The name of the constructor must be the same as the name of the class. If the constructor has parameters, then the arguments for the constructor call must be given after the object name (at the point where the object is declared).

no parameters = default constructor

from Data Structures and Other Objects Using C++



Class constructor

Set default value of radius to zero if omitted during instantiation

```
class Circle{
public:
    Circle(double radius = 0) {
        setRadius(radius);
    }

    // const prevents getRadius from changing m_radius
    double getRadius() const {
        return m_radius;
    }

    // setRadius prevents m_radius being set to a ne
    void setRadius(double radius) {
        if (radius >= 0){
            m_radius = radius;
        } else {
            m_radius = 0;
        }
    }

private:
    double m_radius;
};
```

Use setRadius in constructor to initialize m_radius

c1 initialized to have radius zero

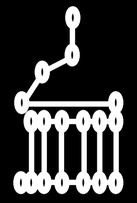
```
int main() {
    Circle c1;
    Circle c2(5);
    Circle c3(-5);

    cout << "radius = " << c1.getRadius() << endl;
    cout << "radius = " << c2.getRadius() << endl;
    cout << "radius = " << c3.getRadius() << endl;

    return 0;
}
```

Output:

```
radius = 0
radius = 5
radius = 0
```

Class Constructor

- The appropriate class constructor is called each time an object is instantiated. Note: there may be more than one constructor for a class.
- The purpose of the constructor is to properly initialize the class data members.
- An efficient way to write a constructor is to use the setter functions.
- A constructor does not have a return type.
- The name of the constructor is the name of the class.



Class constructor

Set default value of radius to zero if omitted during instantiation

```
class Circle{
public:
    Circle(double radius = 0) {
        setRadius(radius);
    }

    // const prevents getRadius from changing m_radius
    double getRadius() const {
        return m_radius;
    }

    // setRadius prevents m_radius being set to a ne
    void setRadius(double radius) {
        if (radius >= 0){
            m_radius = radius;
        } else {
            m_radius = 0;
        }
    }

private:
    double m_radius;
};
```

Use setRadius in constructor to initialize m_radius

c1 initialized to have radius zero

```
int main() {
    Circle c1;
    Circle c2(5);
    Circle c3(-5);

    cout << "radius = " << c1.getRadius() << endl;
    cout << "radius = " << c2.getRadius() << endl;
    cout << "radius = " << c3.getRadius() << endl;

    return 0;
}
```

Output:

```
radius = 0
radius = 5
radius = 0
```

Data encapsulation

Software Engineering Observation 9.2

Using an object-oriented programming approach often requires fewer arguments when calling functions. This benefit derives from the fact that encapsulating data members and member functions within a class gives the member functions the right to access the data members.

Error-Prevention Tip 9.3

The fact that member-function calls generally take either no arguments or fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

Software Engineering Observation

Software Engineering Observation 9.3

Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or fewer arguments than function calls in non-object-oriented languages. Thus, the calls, the function definitions and the function prototypes are shorter. This improves many aspects of program development.