C++ Programming I

Basics of Object-Oriented Programming I
Class and Objects

C++ Programming
March 15, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- **Object-Oriented Programming**
- **Class and Objects**
  - Encapsulation
  - Abstraction
- **Constructor**
  - Declaration and Implementation
  - Default Constructor
  - Constructor Overloading
  - Initialization Lists

Object-Oriented Programming

Lecture 4

Dr. P. Arnold

Bern University
of Applied Sciences

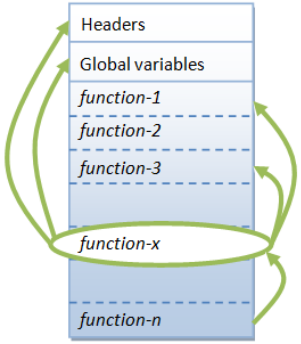
Object-Oriented Programming

Intro

- Object-Oriented Programming (OOP) is the term used to describe a programming approach based on **objects** and **classes**
- The object-oriented paradigm allows us to organise software as a collection of objects that consist of both **data** and **behaviour**
- This is in contrast to conventional functional programming practice that only loosely connects data and behaviour
- The object-oriented programming approach encourages:
  1. Modularisation
  2. Software re-use
- An object-oriented programming language generally supports four main features:
  1. **Classes**
  2. **Objects**
  3. **Inheritance**
  4. **Polymorphism**
- **Why OOP**

Object-Oriented Programming

Drawbacks of Traditional Procedural-Oriented Programming Languages

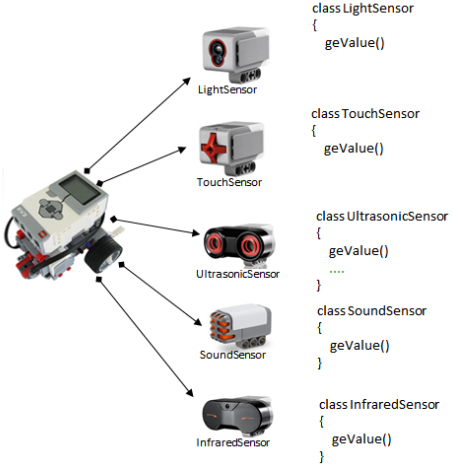


A function (in C) is not well-encapsulated

- The procedural-oriented programs are made up of functions. Function are likely to reference global variables and other functions, therefore **difficult to reuse**
- Functions are **not well-encapsulated** as a self-contained reusable unit
- The traditional procedural-languages **separate the data structures and algorithms**

Object-Oriented Programming

OOP Approach - Lego Robot Example



- OOP permits higher level of abstraction for solving real-life problems!
- Ease in software design as you think in the problem space rather than the machine's bits and bytes

The principle of object-oriented programming is that an object is a logical entity built of **data and algorithms**.

An example:

- A toaster = **Object**
- works with bread = **Data**
- while toasting the bred = **Method** / **Algorithm**
- so that the bread gets toasted = **Status Change**

A toaster without bread makes no sense!

Toast-bread without a toaster also makes no sense!

Note:

Data (member variables) and corresponding algorithms, the **methods** (member functions), are grouped to an entity, the **object** defined by **the class**.

Object-Oriented Programming

Example using `struct`

```
1  #include <iostream>
2
3  struct DateStruct
4  {
5      int year;
6      int month;
7      int day;
8  };
9
10 void print(DateStruct &date)
11 {
12     std::cout << date.year << "/" << date.month << "/"
           << date.day;
13 }
14
15 int main()
16 {
17     DateStruct today{2020, 10, 14}; // uniform
           initialization
18
19     today.day = 16; // use dot operator for access
20     print(today);
21
22     return 0;
23 }
```

Object-Oriented Programming

Comparison with `class`

```
1  struct DateStruct
2  {
3      int year;
4      int month;
5      int day;
6  };
7
8  class DateClass
9  {
10 public:
11     int m_year;
12     int m_month;
13     int m_day;
14 };
```

- Note that the only significant difference is the keyword `public:`:

Note:

`struct` is public by default
`class` is private by default

Object-Oriented Programming

Member Function

```
1  class DateClass
2  {
3  public:
4      int m_year;
5      int m_month;
6      int m_day;
7
8      void print() // defines a member function named
           print()
9      {
10         std::cout << m_year << "/" << m_month << "/" <<
               m_day;
11     }
12 };
```

- In addition to holding data, classes can also contain functions!
- All member function calls are associated with an object of the class
- No object has to be passed

Object-Oriented Programming

Example using `class` with member function

```
1  #include <iostream>
2
3  class DateClass
4  {
5  public:
6      int m_year;
7      int m_month;
8      int m_day;
9
10     void print()
11     {
12         std::cout << m_year << "/" << m_month << "/" <<
               m_day;
13     }
14 };
15
16 int main()
17 {
18     // create "instance" of class DateClass = "object"
19     DateClass today{2020, 10, 14};
20
21     today.m_day = 16; // use dot operator for access
22     today.print(); // use dot operator for calls
23     return 0;
24 }
```

- When we call “today.print()”, we’re telling the compiler to call the print() member function, associated with the today object
- The associated object is **implicitly passed** to the member function

Class and Objects

Bern University
of Applied Sciences

Classes and Objects

Class Human Being

Imagine you are writing code to model a human being:

- **Object**
  - Human being
- **Data**
  - Name
  - Date of birth
  - Gender
- **Method**
  - introduceSelf()
  - gettingOlder()
  - ...

The construct to group the attributes (data) that defines a human and the activities (methods) a human can perform using these available attributes is a class.

Classes and Objects

Declaring a Class

A class is declared using the keyword `class` as follows:

- `class NameOfClass{...};` // Pascal Case for objects

For a human being:

```
1  class Human
2  {
3     // member variables / i.e. members
4     string m_name;
5     string m_dateOfBirth;
6     int m_age;
7
8     // member functions / i.e. methods
9     void introduceSelf();
10    void gettingOlder();
11 }; // declarations end with ';'
```

- Using the “m_” prefix for member variables helps distinguish member variables from function parameters or local variables inside member functions
- By convention, class names should begin with an upper-case letter

Note:

With the keyword `class` C++ provides a powerful way to **encapsulate** member data and member functions working with those

Classes and Objects

An Instance of a Class

A class is construction plan only!

- The declaration has no effect on program execution
- To use the features of a class create an instance of the class called object

```
1  // Creating an object of type double and type class
2
3  double pi = 3.1459; // a variable of type double
4  Human firstMan; // firstMan: an object of class Human
5
6  // Dynamic creaton using new
7
8  int* intPtr = new int; // an integer allocated
       dynamically
9  delete intPtr; // de-allocate memory
10
11 Human* rareHumanPtr = new Human(); // dynamic allocation
       of Human
12 delete rareHumanPtr; // de-allocate Human
```

Classes and Objects

Accessing Members

- Instance `firstMan` is an object of `class human` with accessible members
- Use the object to access the members methods and attributes through the dedicated operators `.` and `->`

```
1  // On the stack, we access members using the Dot
       Operator (.)
2  Human firstMan;
3  firstMan.m_dateOfBirth = "1987";
4  firtMan.introduceSelf();
5
6  // On the heap, we access members using the Pointer
       Operator (->)
7  Human* rareHumanPtr = new Human();
8  rareHumanPtr->m_dateOfBirth = "1987";
9  rareHumanPtr->introduceSelf();
10
11 // Or use the indirection operator (*) following the dot
       operator
12 (*rareHuman).introduceSelf();
```

Classes and Objects

Example `Class Human`

Declaration of `class Human`:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Human
6  {
7  public:
8     string m_name;
9     int m_age;
10
11    void introduceSelf()
12    {
13       cout << "I am " + m_name << " and am ";
14       cout << m_age << " years old" << endl;
15    }
16 };
```

- Note the new keyword `public`
- Attributes and methods are declared `public`

Classes and Objects

Example `Class Human`

```
1  int main()
2  {
3      // An object of class Human with attribute m_name as
           "Adam"
4      Human firstMan;
5      firstMan.m_name = "Adam";
6      firstMan.m_age = 30;
7
8      // An object of class Human with attribute m_name as
           "Eve"
9      Human firstWoman;
10     firstWoman.m_name = "Eve";
11     firstWoman.m_age = 28;
12
13     firstMan.introduceSelf();
14     firstWoman.introduceSelf();
15 }
```

Output:

```
1  I am Adam and am 30 years old
2  I am Eve and am 28 years old
```

Warning

This is bad programming style! E.g. Anybody can change your name! **Member variables should never be public!**

We need to learn features that help you to protect members your class should keep hidden from those using it!

Data Encapsulation

`public` and `private`

To avoid direct access to members the keyword `public` and `private` are used:

- `private` members can only be accessed by member functions of the same class
- `public` members are accessible from outside

Passive access in `struct`

```
1  struct DataContainer
2  {
3     // components
4     int value;
5  };
6
7  struct DataContainer d;
8
9  d.value=0; // passive access
```

Note

`struct` is `public` by default

Active access in `class`

```
1  class DataContainer
2  {
3     int m_value;
4     void set(int v) {m_value=v;} // setter (write access)
5     int get(){return(m_value);} // getter (read access)
6  };
7
8  DataContainer d;
9
10 d.m_value=0; //Is this OK?
11 d.set(0); // ... and this
12 printf("member m_value = %d\n", d.get()); // ?
```

Data Encapsulation

public and private

```
1  class DataContainer
2  {
3  public: // methods
4      void set(int value) {m_value=value;} // setter
           (write access)
5      int get(){return m_value;} // getter (read access)
6
7  private: // members
8      int m_value;
9  };
10
11 int main()
12 {
13     DataContainer d;
14
15     d.set(0); // OK, active write access
16     d.m_value=0; // compile error - cannot access
           private member
17
18     printf("member m_value = %d\n", d.get()); // OK,
           active read access
19 }
```

- C++ enables the designer of the class how members are accessed and manipulated by setter and getter methods
- Access control is checked at compile time

Consistency

Data Validity

Back to `class Human`

```
1  class Human
2  {
3  public:
4      // Verify correct input. i.e. non-zero & non-negative
5      void setAge(int age)
6      {
7          if(age > 0)
8              m_age = age;
9          else
10             m_age = 0;
11     }
12
13 private: // member data
14     int m_age;
15 };
```

- Besides encapsulation setter methods enable data consistency

Abstraction of Data

using keyword `private`

```
1  #include <iostream>
2  using namespace std;
3
4  class Human
5  {
6  public:
7      void setAge(int age)
8      {
9          if(age > 0)
10             m_age = age;
11         else
12             m_age = 0;
13     }
14
15     // Human lies about his / her age (if over 30)
16     int getAge()
17     {
18         if (m_age > 30)
19             return m_age-2;
20         else
21             return m_age;
22     }
23
24 private:
25     // Private member data:
26     int m_age;
27 };
```

Abstraction of Data

using keyword `private`

```
1  int main()
2  {
3      Human firstMan;
4      firstMan.setAge(35);
5
6      Human firstWoman;
7      firstWoman.setAge(22);
8
9      cout << "Age of firstMan " << firstMan.getAge() <<
           endl;
10     cout << "Age of firstWoman " << firstWoman.getAge()
           << endl;
11
12     return 0;
13 }
```

Output:

```
1  Age of firstMan 33
2  Age of firstWoman 22
```

Summary

Meaning:

- **Data abstraction** refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details
- Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability
- Can change internal implementation of class independently without affecting the user
- Helps to increase security of an application or program as only important details are provided to the user

Constructor

Constructor

Object-Cycle

- A constructor is a special initialization function (method) existing for every class
- The method is always called when an instance is created
- The constructor can define all values of the newly created instance
- An explicit initialization is no longer required!

1. Given by a construction plan any number of similar objects can be built
2. Within its lifetime each object fulfils its tasks, i.e. running through its states
3. When finished, the object is disposed automatically when out of scope

Constructor

Declaration and Implementation

```
1  // Declaration of a constructor
2  class Human
3  {
4  public:
5      Human(); // declaration only
6  };
7
8
9  // Inline implementation (definition) of a constructor
10 class Human
11 {
12 public:
13     Human()
14     {
15         // constructor code
16     }
17 }; // declarations end with ';'
```

- The constructor is e.g. declared in the header file, e.g. `human.h`
- The constructor can be declared and defined in the header file.

Constructor

Declaration and Implementation

```
1  // Defining the constructor outside the class
2
3  // human.h
4  class Human
5  {
6  public:
7      Human(); // constructor declaration
8  };
9
10 // Constructor implementation (definition)
11 // human.cpp
12 Human::Human()
13 {
14     // constructor code
15 } // definition ends without ';'!
```

- The constructor is declared in the header file and defined in the source, e.g. `human.cpp`
- Declaration and implementation are separated

Note:

Declarations end with '`;`' and definitions end without semicolon!

Constructor

Default Constructor & Class Member Variables

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Human
6  {
7  public:
8      Human() // Default Constructor
9      {
10         m_name = "";
11         m_age = 0; // initialize valid values
12
13         cout << "Constructed an instance of class Human"
               << endl;
14     }
15
16 private:
17     string m_name;
18     int m_age;
19 };
```

- The constructor is the perfect place to initialize member variables to a valid value, i.e. `m_age = 0`

Note:

A constructor without arguments is called the **default constructor**. Programming a default constructor is optional.

Constructor

Overloading Constructors

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Human
6  {
7  public:
8      Human() // default constructor
9      {
10         m_age = 0; // initialized to ensure no junk value
11         cout << "Default constructor: ";
12         cout << "name and age not set" << endl;
13     }
14     // overloaded constructor
15     Human(string name, int age)
16     {
17         m_name = name;
18         m_age = age;
19         cout << "Overloaded constructor creates ";
20         cout << m_name << " of " << m_age << " years" <<
               endl;
21     }
22
23 private:
24     string m_name;
25     int m_age;
26 };
```

Constructor

Overloading Constructors - Usage

```
1  int main()
2  {
3     Human firstMan; // use default constructor
4     Human firstWoman("Eve", 20); // use overloaded
          constructor
5  }
```

Output:

```
1  Default constructor: name and age not set
2  Overloaded constructor creates Eve of 20 years
```

- Members can be set or not
- It's good programming style to set all member variables at object instantiation to guarantee the object is ready to use

Constructor

No Default Constructor

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Human
6  {
7  private:
8     string m_name;
9     int m_age;
10
11 public:
12    Human(string name, int age)
13    {
14       m_name = name;
15       m_age = age;
16       cout << "Overloaded constructor creates " << name;
17       cout << " of age " << m_age << endl;
18    }
19
20    void introduceSelf()
21    {
22       cout << "I am " + m_name << " and am ";
23       cout << m_age << " years old" << endl;
24    }
25 };
```

- Enforce object instantiation with minimal paramters.

Constructor

No Default Constructor - Usage

```
1  int main()
2  {
3     Human noName(); // compile error!
4     Human firstMan("Adam", 25);
5     Human firstWoman("Eve", 28);
6
7     firstMan.introduceSelf();
8     firstWoman.introduceSelf();
9  }
```

Output:

```
1  Overloaded constructor creates Adam of 25 years
2  Overloaded constructor creates Eve of 28 years
3  I am Adam and am 25 years old
4  I am Eve and am 28 years old
```

- No default constructor is generated by the compiler
- Private member variables `name` and `age` are set at instantiation
- The humans attributes, e.g. `name` are not allowed to change!

Constructors

Initialization Lists -

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Human
6  {
7  private:
8      string m_name;
9      int m_age;
10
11 public:
12     Human(string name, int age)
13         :m_name(name), m_age(age)
14     {
15         cout << "Constructed a human called " << m_name;
16         cout << ", " << m_age << " years old" << endl;
17     }
18 };
```

- More efficient
- Respect order of declaration!
- Members are initialized in the order they're declared in your class, not the order you initialize them in the constructor!
- This is to help prevent errors where the initialization of b depends on a or vice-versa

Constructors

Initialization Lists - Order of Declaration Matters!

```
1  class Order
2  {
3      // order of initialisation
4  public:
5      Order(int i) : m_a(++i), m_b(++i), m_c(++i) {}
6
7      // order of declaration
8  private:
9      int m_a;
10     int m_c;
11     int m_b;
12 };
```

- Most people assume `a=1`, `b=2` and `c=3`
- But, in fact, `a=1`, `c=2` and `b=3`
- Why? Because the initializer expressions happen in the order the variables are declared in the class – not the order the initializer expressions appear in the constructor.

Note:

always check warnings of the compiler!

Constructors

Initialization Lists with Default Parameters

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Human
6  {
7  private:
8      string m_name;
9      int m_age;
10
11 public:
12     Human(string name = "Adam", int age = 25)
13         :m_name(name), m_age(age)
14     {
15         cout << "Constructed a human called " << m_name;
16         cout << ", " << m_age << " years old" << endl;
17     }
18 };
```

Constructors

Initialization Lists with Default Parameters - Usage

```
1  int main()
2  {
3      Human adam;
4      Human eve("Eve", 18);
5      return 0;
6  }
```

Output:

```
1  Constructed a human called Adam, 25 years old
2  Constructed a human called Eve, 18 years old
```

Thank You
Questions

???