Akademin

Object Oriented Programming

C++ Language

Object Oriented Programming

- An object has properties and behaviours (data and functions)
- OOP means creating objects that tie together both properties and behaviors into a self-contained, reusable package. E.g. car.brand = “Volvo”; car.drive();
- In C++, classes, structs and unions can be used to make objects
- A class is a template used to create objects.
  - A class is a user defined data type and an object is an instances of the class
- To define a class, use the class keyword followed by a name, a code block, and a semicolon; like structs. E.g. class Rectangle {};
- C++ classes and structs support encapsulation, inheritance and polymorphism
- C++ unions support encapsulation but not inheritance and polymorphism

Class Members

- Class members can be declared inside a class
  - The main members are properties and methods. Properties are variables and they hold the state of objects and methods are functions and they define what the objects can do.
  - E.g. class Rectangle { double width, height; double getArea() { return width * height; } };
- Creating an instance(object) of a class is like declaring variables.E.g. Rectangle rect;
  - Any number of instances can be created
- Dot (.) and **->** are used to get access to class members.
- Members can be private, public and protected.
- To get access to members out of the class, we have to make them public using the public access specifier.
- Members of a class by default are private.

```
#include <iostream>
class Rectangle {
public:
   double width, height;
   double getArea() { return width * height; }
};
int main(void) {
   Rectangle rect;
   rect.width = 10.0;  rect.height = 20.0;
   std::cout << rect.getArea() << std::endl;
   return 0;
}
```

Class Members

- unions and structs can be used like classes to define data types.
  - Members by default are public.
- When all properties of a class are public we can initialize an instance using list initialization when we create it.
  E.g. Rectangle rect{5.6, 10.5};
- If properties are private, we don't have access to them and therefore we need to have special functions such as public constructors, getters or setters.
- A **constructor** is a special kind of member function that is automatically called when an object of the class is created. Constructors are typically used to initialize instances.

```cpp
#include <iostream>

class Rectangle {
    double width{0}, height{0}; // private members

public:
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
    double getArea() { return width * height; }
};

int main(void) {
    Rectangle rect;
    rect.setHeight(20.0);
    rect.setWidth(10.0);
    std::cout << "Area: " << rect.getArea() << std::endl;

    return 0;
}
```

Class Constructors

- Unlike normal member functions, constructors
  - Should be public and have the same name as the class
  - Have no return type (not even void)
- Constructors can have parameters
  - A constructor that has no parameter (or has params with default values) is called a **default constructor**
- A constructor can be overloaded and it is possible to have multiple constructors in a class. Even constructors can call others members. E.g. Rectangle() : Rectangle(0.0, 0.0){};
- If we don't define a constructor, the compiler implicitly defines the default constructor.

```
#include <iostream>

class Rectangle {
private:
   double width, height; // private members
public:
   // It can be Rectangle() : Rectangle(0.0, 0.0){};
   Rectangle() { width = 0.0; height = 0.0; }
   Rectangle(double w, double h)
   { width = w; height = h; }
   double getArea() { return width * height; }
};

int main(void) {
   Rectangle r1; // Calls the default constructor
   std::cout << r1.getArea() << std::endl;
   Rectangle r2(5.0, 15.0); // Rectangle(double, double)
   std::cout << r2.getArea() << std::endl;
   return 0;
}
```

Class Constructors

- If we define our own constructor(s), the compiler does generate the default one automatically. We can explicitly tell to the compiler using the default keyword to generate it.
- Properties in constructors can be initialized using a list.
  - E.g. Date(int y, int m, int d) : m_year{y}, m_month{m}, m_day{d} {/* ... */}
- Inside non-static member functions of a class a special keyword, this, which is a pointer to the current instance can be used. E.g. int getYear() { return this->m_year; }
- A class can also have an explicitly defined **destructor** which is a member function that is executed when an object of the class is destroyed. It is used to cleanup the object. A destructor should be public, has no return type, has no params and must have the same name as the class, preceded by a tilde (~)

```
class Date {
private:
   int m_year{1900},  m_month{1},  m_day{1};
public:
  // Date() = default;
  // normal non-default constructor
  Date(int year, int month, int day)
  { m_year = year; m_month = month; m_day = day;}
  // No implicit constructor provided
  // because we already defined our own constructor
  ~Date() { std::cout << "Destructor" << std::endl; }
};
int main(void) {
   Date date{}; // error: because default constructor
doesn't exist. We could have: Date() = default;
   Date today{2020, 1, 19}; // today is initialized
   return 0;
}
```

Copy Constructors

- A **copy constructor** is a special type of constructor used to create a new object as a copy of an existing object
- Like the default constructor, if you do not create a copy constructor for your class, a public copy constructor is generated by the compiler
- The default copy constructor initializes members of the copy directly from the object being copied
- We can explicitly define the copy constructor

```
#include <iostream>

class Point {
private:
   int x, y;
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   void print(void) { std::cout << "(" << x << ", " << y << ")\n"; }
};

int main(void) {
   Point p{2, 3};          // Uniform initialization
   Point q(4, 6);           // Direct initialization
   Point r = Point(6, 8); // Copy initialization
   Point c{p};             // The default copy constructor is called
   p.print(); q.print(); r.print(); c.print();
   return 0;
}
```

Copy Constructors

- Possible to make the copy constructor of a class private or delete it.

  E.g. Point(const Point &) = delete;

- When we use an anonymous object to initialize a new object, the compiler may omit calling the copy constructor in order to improve the performance. This process is called **elision**.
  - E.g. Point point{ Point{2, 3} };
    - The compiler may change it to

      Point point{2, 3};
    - And the copy constructor is not called

```
#include <iostream>

class Point {
private: int x{}, y{};
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   Point(const Point &point) : x{point.x}, y{point.y} // Copy constructor
   { std::cout << "Copy constructor called" << std::endl; }
   void print(void) { std::cout << "(" << x << ", " << y << ")\n"; }
};

int main(void) {
   Point p{2, 3};
   Point q{p};    // The copy constructor is called
   p.print();
   q.print();
   return 0;
}
```

Constructors and Implicit Type Conversions

- By default, C++ uses any constructor for implicit type conversion.
- In order to prevent implicit type conversions the constructors can be defined as explicit using the explicit keyword. E.g.
  - explicit Point(int m = 0, int n = 0) : x{m}, y{n} {}
  - Then codes like Point p = 10; are not allowed

```
#include <iostream>

class Point {
private: int x, y;
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   void print(void) { std::cout << "(" << x << ", " << y << ")\n"; }
};

int main(void) {
   Point p = 10; // Implicit type conversion; Point(10, 0) is called
   p.print();
   return 0;
}
```

- Possible to make pointer to function members. For example:
  - using func_t = void (Point::*)(void);
  - Point p; func_t fptr = &Point::print;
  - (p.*fptr)(); // Call print using the function pointer

Class - Const Member Functions

- When a const object gets initialized via constructor, we can not modify the variable members directly or indirectly
- const objects can only explicitly call const member functions.E.g. void print() const {}
- A **const member function** is a function member that guarantees it will not modify the object or call non-const member functions
- Const members can not return non-const references/pointers to variable members

```
#include <iostream>

class Date {
private:
   int year{1900}, month{1}, day{1};

public:
   double temp{};
   Date() = default;
   Date(int y, int m, int d) : year{y}, month{m}, day{d} {}
   void print()
   { std::cout << year << "-" << month << "-" << day << std::endl; }
   ~Date() { std::cout << "Destructor" << std::endl; }
};

int main(void) {
   const Date today{2021, 9, 16};
   // today.temp = 10.0; // error. temp is public
   // today.print(); // error, even if printDate does not modify the variables
   return 0;
}
```

Class - Static Members

- In addition to static global and local variables, in a class we can have static variable and and function members
- Static members are not associated with class objects. They belong to the class(no this).
- Static members are declared using static
- Definition of non-const static members shall be done outside of the class. const static members can be initialized in the class
- Static function members are used to get access to static variable

```
#include <iostream>

class Product {
private:
   int id;
   static int serial;
   static const int value{100};

public:
   Product() { id = serial++; } // The next value from the serial id generator
   int getID() const { return id; }
   static const int getValue() { return value; }
};

int Product::serial{1}; // start our ID generator with 1

int main(void) {
   std::cout << Product::getValue() << std::endl;

   Product first, second, third;
   std::cout << first.getID() << std::endl;
   std::cout << second.getID() << std::endl;
   std::cout << third.getID() << std::endl;
   return 0;
}
```

Class - Singleton

- By default we can create any number of instances of a class.
- To make a singleton class we shall
  - Make the constructor(s) private
    - The client cannot create instances
  - Have a static function to create an instance and return a pointer/reference to the instance. The client can call this function without having an instance and get a handle to the instance created in the static function
  - A singleton class is a single instance module

```
#include <iostream>
class Point { int x, y; Point(const Point &) = delete;
  Point(int m, int n) : x{m}, y{n} {}
public:
  static Point *handle(int m = 0, int n = 0) {
    static Point point{m, n}; return &point;
  }
  void print(void) const {
    std::cout << "(" << x << ", " << y << ")" << std::endl;
  }
};
void func(void) {
  const Point *point = Point::handle();
  std::cout << point << std::endl; point->print();
}
int main(void) { Point *point = Point::handle(2, 3);
  std::cout << point << std::endl; point->print(); func();
  return 0;
}
```

Class - Friend Functions

- Sometimes we need to get access to private members outside of the class. C++ has friend functions and classes to achieve this
- A friend function is a function that can access the private members of a class like a func member
- It is also possible to make an entire class as a friend to another class

```
class B; // Forward declared class B
class A {
   int value;

public:
   A(int x = 0) : value{x} {};
   friend class B;
};
```

```
class B {
   int value;

public:
   B(int x = 0) : value{x} {};
   int add(const A &a) {
      return value + a.value;
   }
};
```

```
#include <iostream>

class Value {
private:
   int value;

public:
   Value(int _value) { value = _value; }
   friend bool isEqual(const Value &value1, const Value &value2);
};

bool isEqual(const Value &value1, const Value &value2) {
   return (value1.value == value2.value);
}

int main(void) {
   Value v1{15}, v2{30};
   std::cout << (isEqual(v1, v2) ? "Equal" : "Not Equal") << std::endl;
   return 0;
}
```

Class - Friend Functions

- It is possible to make a single member function as a friend

```
// An example of a member function as a friend
#include <iostream>

class A; // Forward declared class A

class B {
   int value;

public:
   B(int x = 0) : value{x} {};
   int add(const A &a);
};
```

```
class A {
   int value;

public:
   A(int x = 0) : value{x} {};
   friend int B::add(const A &a);
};
```

```
int B::add(const A &a) { return value + a.value; }

int main(void) {
   B var{20};
   std::cout << var.add(A{10}) << std::endl;
   return 0;
}
```

Operator Overloading

- In C++, it is possible to overload operators
  - In C++, operators are implemented as functions.
  - Using function overloading on the operator functions, you can define your own versions of the operators that work with different data types
  - All the operators, except **.**, **.***, **::**, **?:**, **#**, **##**, **sizeof** and **typeid** can be overloaded and we can **only** overload the operators that exist.
  - At least one of the operands in an overloaded operator must be a user-defined type.

```cpp
#include <iostream>

int main(void) {
    std::string a{"C++ "}, b{"Language"};
    std::string c = a + b; // std::operator+(a, b);
    std::cout << c << std::endl;
    return 0;
}
```

```cpp
struct Point {
  int x;
  int y;
};
Point operator+(const Point &p, const Point &q) {
    return Point{p.x + q.x, p.y + q.y};
}
int main(void) {
    Point p{1, 2}, q{3, 4}, r{p + q};
    return 0;
}
```

Operator Overloading

- It is not possible to change the number of operands which is used by an operator.
- All operators keep their default precedence and associativity and this can not be changed.
- To overload an operator, the operator keyword is used.
  - E.g. **Point operator+(const Point &point);**
- It is possible to use friend and member functions to overload operators.
- But if type of the first operand of an operator is not the class then we have to use a friend function; otherwise we can overload the operator as member function.

```
class Point {
private: int x, y;
public:
   Point(int m, int n) : x{m}, y{n} {}
   Point operator+(const Point &point) {
     return Point{x + point.x, y + point.y};
   }
};

int main(void) {
   Point p{1, 2}, q{3, 4}, r{p + q};
   return 0;
}
```

Operator Overloading

- Possible to overload operators using member functions

```
#include <iostream>
class Point {
private: int x, y;
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   friend std::ostream &operator<<(std::ostream &out, const Point &point);
   friend Point operator*(int num, const Point &p);
   Point operator*(int num) { return {num * x, num * y}; } // A member function
};

Point operator*(int num, const Point &p) { return {num * p.x, num * p.y}; }

std::ostream &operator<<(std::ostream &out, const Point &point) {
   out << "(" << point.x << ", " << point.y << ")"; return out;
}

int main(void) {
   Point p{2, 3};
   std::cout << p * 2 << 3 * p << p << std::endl;
   return 0;
}
```

```
#include <iostream>

class Point {
private: int x, y;
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   friend std::ostream &operator<<(std::ostream &out, const Point &point);
   // A member function
   Point operator*=(int num) { x *= num; y *= num; return *this; }
};

std::ostream &operator<<(std::ostream &out, const Point &point) {
   out << "(" << point.x << ", " << point.y << ")";
   return out;
}

int main(void) {
   Point p{2, 3};
   p *= 2; // *= has been overloaded using a member function
   std::cout << p << std::endl;
   return 0;
}
```

Operator Overloading

- Possible to overload the I/O operators.

```
#include <iostream>

class Point {
private: int x, y;
public:
    Point(int m, int n) : x{m}, y{n} {}
    Point operator+(const Point &p) { return Point{p.x + x, p.y + y}; }
    friend std::ostream &operator<<(std::ostream &out, const Point &point);
};

std::ostream &operator<<(std::ostream &out, const Point &point) {
    out << "(" << point.x << ", " << point.y << ")";
    return out;
}

int main(void) {
    Point p{1, 2}, q{3, 4}, a{p + q};
    std::cout << p << " + " << q << " = " << a << std::endl;
    return 0;
}
```

```
#include <iostream>

class Point {
private: int x, y;
public:
    Point(int m = 0, int n = 0) : x{m}, y{n} {}
    friend std::ostream &operator<<(std::ostream &out, const Point &point);
    friend std::istream &operator>>(std::istream &in, Point &point);
};

std::ostream &operator<<(std::ostream &out, const Point &point) {
    out << "(" << point.x << ", " << point.y << ")"; return out;
}

std::istream &operator>>(std::istream &in, Point &point) {
    in >> point.x >> point.y; return in;
}

int main(void) {
    Point point;
    std::cin >> point;
    std::cout << point << std::endl;
    return 0;
}
```

Operator Overloading

/ Overloading of the comparison operators. E.g. == and != **/**

```
#include <iostream>

class Point {
private: int x, y;
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   friend std::ostream &operator<<(std::ostream &out, const Point &point);
   bool operator==(const Point &p) { return (p.x == x && p.y == y); }
   bool operator!=(const Point &p) { return (p.x != x || p.y != y); }
};

std::ostream &operator<<(std::ostream &out, const Point &point) {
   out << "(" << point.x << ", " << point.y << ")"; return out;
}

int main(void) {
   Point p{2, 3}, q{4, 6};
   std::cout << p << " is " << ((p != q) ? "not " : "") << "equal to " << q << std::endl;
   return 0;
}
```

/* Overloading of unary operators. E.g. negative (-) ***/**

```
#include <iostream>

class Point {
private: int x, y;
public:
   Point(int m = 0, int n = 0) : x{m}, y{n} {}
   friend std::ostream &operator<<(std::ostream &out, const Point &p);
   Point operator-() const { return Point{-x, -y}; }
};

std::ostream &operator<<(std::ostream &out, const Point &p) {
   out << "(" << p.x << ", " << p.y << ")";
   return out;
}

int main(void) {
   Point p{2, 3};
   std::cout << "-" << p << " = " << -p << std::endl;
   return 0;
}
```

Operator Overloading

```
/*** Overloading the subscript operator  ***/
#include <cassert>
#include <iostream>

class IntList {
private: int list[10]{1, 2, 3, 4, 5, 6, 7, 8, 9};
public:
   int &operator[](size_t index)
   { assert(index < sizeof(list) / sizeof(list[0])); return list[index]; }

   int operator[](size_t index) const
   { assert(index < sizeof(list) / sizeof(list[0])); return list[index]; }
};
int main(void) {
   IntList a;
   a[2] = 50;
   std::cout << a[2] << std::endl;
   const IntList b;
   std::cout << b[2] << std::endl;
   return 0;
}
```

```
/*** Overloading the parenthesis operator ***/
#include <cassert>
#include <iostream>
class Matrix {
private: int data[4][4]{};
public:
   int &operator()(int row, int col)
   { assert(col >= 0 && col < 4); assert(row >= 0 && row < 4); return data[row][col]; }

   int operator()(int row, int col) const
   { assert(col >= 0 && col < 4); assert(row >= 0 && row < 4); return data[row][col]; }

   void operator()();
};
void Matrix::operator()()  // Reset all elements to 0
{ for (int row{0}; row < 4; ++row) { for (int col{0}; col < 4; ++col) { data[row][col] = 0; } } }

int main(void) {
   Matrix matrix{};  matrix(1, 2) = 4;
   std::cout << matrix(1, 2) << std::endl;
   matrix(); // reset matrix
   std::cout << matrix(1, 2) << std::endl;
   return 0;
}
```

Operator Overloading

- It is possible to enable user-defined implicit or explicit type conversions

```
/*** Overloading typecasts - user defined conversions  ***/
#include <iostream>
class Cent {
private: int m_cent;
public:
   Cent(int cent = 0) : m_cent{cent} {}
   operator int() const { return m_cent; } // Overloaded int cast. You can make it explicit
   int getCent(void) const { return m_cent; }
   void setCent(int cent) { m_cent = cent; }
};
class Dollar {
private: int m_dollar;
public:
   Dollar(int dollar = 0) : m_dollar{dollar} {}
   operator Cent() const { return Cent{m_dollar * 100}; } // Convert Dollars into Cents
};
void printCent(Cent cent) { std::cout << cent; /* cent will be implicitly cast to an int here */}
int main(void) {
   Dollar dollars{9};
   printCent(dollars); // dollars will be implicitly cast to a Cent here
   std::cout << std::endl;  return 0;
}
```

```
/*** Overloading the parenthesis operator to make functors ***/
#include <iostream>

class Accumulator {
private: int value{0};
public:
   int operator()(int i) { return (value += i); }
};

int main(void) {
   Accumulator acc{}; // acc is a functor
   std::cout << acc(10) << std::endl; // prints 10
   std::cout << acc(20) << std::endl; // prints 30
   return 0;
}
```

User-defined conversions do not take parameters, as there is no way to pass arguments to them and they also do not have a return type. C++ assumes you will return the correct type.

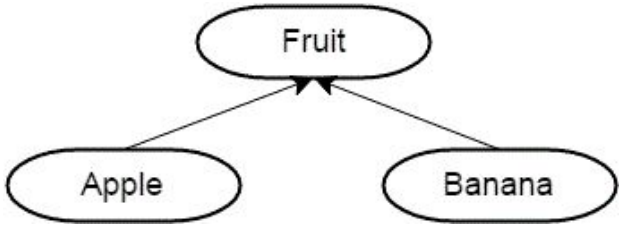
The Copy Assignment Operator

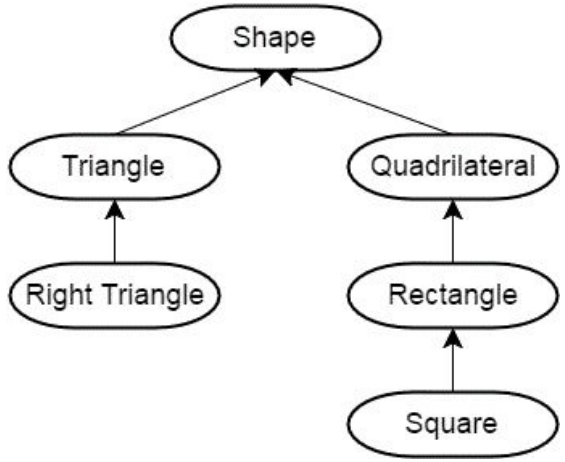
- Overloading the assignment operator(**=**)
- An ***assignment*** is used to copy an object to ***another existing object***. But copy constructors initialize **new** objects by copying another object.
- Unlike other operators, the compiler provides a default public assignment operator for your class if you do not provide one.
- Like other constructors and operators, you can prevent assignments by making the assignment operator private or delete it
  - E.g. `Point &operator=(const Point &point) = delete;`
- To make an uncopyable class you can make the copy constructor and the copy assignment of the class private or delete them.

```
#include <iostream>
class Point {
private: int x, y;
public:
   explicit Point(int m = 0, int n = 0) : x{m}, y{n} {}
   Point &operator=(const Point &point); // Overloaded assignment
   friend std::ostream &operator<<(std::ostream &out, const Point &point);
};
Point &Point::operator=(const Point &point) {
   if (this != &point) // self-assignment guard
   { x = point.x; y = point.y; }
   return *this; // return the existing object so we can chain the operator
}
std::ostream &operator<<(std::ostream &out, const Point &point)
{ out << "(" << point.x << ", " << point.y << ") "; return out; }
int main(void) {
   Point p{1, 2}, q, r;
   r = q = p; // Chained assignments call overloaded assignment;
   std::cout << p << q << r << std::endl;
   return 0;
}
```

Inheritance

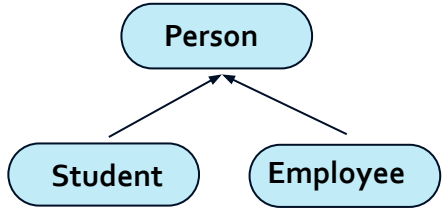
- C++ supports inheritance and classes can inherit properties and behaviours from each other. For example a child inherits some features from his/her parents, an apple is a fruit, a square is a rectangle and both of them are shapes.
- Inheritance allows a class to acquire members of another class.





Inheritance

- The class being inherited from is called the **parent**, **base**, or **superclass**, and the class doing the inheriting is called the **child**, **derived**, or **subclass**. E.g. Person is a base class; Student and Employee are two derived classes from Person




```
#include <vector>
#include <iostream>

class Person {
   int id, age; std::string name;
protected:
   Person(int _id, int _age, const std::string &_name)
     :  id{_id}, age{_age}, name{_name} {}
public:
   int getID(void) const { return id; }
   int getAge(void) const { return age; }
   const std::string &getName() const
   { return name; }
};
```

```
class Student : public Person {
   std::vector<int> grades{};
public:
   Student(int id, int age, const std::string &name, const std::vector<int> &_grades) :
            Person{id, age, name}, grades{_grades} {}
   int getGrade(unsigned int cindex) const { return (cindex < grades.size()) ? grades.at(cindex) : -1; }
};

class Employee : public Person {
   int salary{};
public:
   Employee(int id, int age, const std::string &name, int _salary) : Person{id, age, name}, salary{_salary} {}
   int getSalary(void) const { return salary; }
};
```

```
int main(void) {
  Employee e{456, 40, "Eva", 40000};
  Student s{123, 23, "Stefan", {10, 20, 30, 50, 80}};
  std::cout << s.getID() << std::endl;
  std::cout << s.getAge() << std::endl;
  std::cout << s.getName() << std::endl;
  std::cout << s.getGrade(3) << std::endl;
  std::cout << e.getSalary() << std::endl;
   return 0;
}
```

Inheritance and Constructors

- It's possible to inherit from a class that itself is derived from another class. E.g. Teacher -> Employee -> Person
- To make sure the base class is properly initialized, the proper constructor of the base class shall be called. If not, the default constructor of the base class is automatically called when an object of the derived class is created
- C++ supports multiple inheritance. The base classes are specified in a comma separated list. For example:

  class Teacher : public Employee, public Person { }
- An object can be upcast to its base class, because it contains everything that the base class contains. E.g. Base &base = derived; Base *base = &derived;

```
#include <iostream>

class Base {
public:
   int x;
   Base() : x(5) {}
};
class Derived : public Base {
public:
   int y;
   // If the base class constructor is not called,
   // Base() will be called automatically

   // using Base::Base; /* inherit all constructors */
   Derived() : Base{}, y{10} {}
};
int main(void) {
   Derived derived;
   std::cout << derived.x;
   return 0;
}
```

Types of Inheritance

- An object of a base class can be downcast to a derived class. The downcast has to be made explicitly.
  E.g. Base base; Derived &drived = static_cast<Derived &>(base);
- A protected member in a class is accessible inside the class. But outside of the class is not accessible.
- Public Inheritance: In the derived class
  - The inherited public members stay public
  - The inherited protected members stay protected
  - ***The private members in the base class are inaccessible in the derived class***
  - E.g. class Derived : public Base { };

```
#include <iostream>

class Base {

private:  int x{10};

protected:  int y{20};

public:
   int z{30};
   int addXYZ(void) const { return x + y + z; }
};

int main(void) {
   Base base;
   // base.x and base.y are not accessible
   std::cout << base.z << " " << base.addXYZ() << std::endl;
   return 0;
}
```

Types of Inheritance

- Protected inheritance: In the derived class
  - The inherited public and protected members become protected. E.g. class Derived : protected Base { };
  - ***The private members in the base class are inaccessible in the derived class***
- Private Inheritance: In the derived class
  - The inherited public and protected members become private. E.g. class Derived : private Base { };
  - ***The private members in the base class are inaccessible in the derived class***
- Best practice: When choosing an access level, it is generally best to use the most restrictive level possible.

```
#include <iostream>

class Base {
private: int x{10};
protected:  int y{20};

public:
   int z{30};
   int addXYZ(void) const { return x + y + z; }
};

class Derived : public Base {
public:
   int mulYZ(void) const { return y * z; }
};

int main(void) {
   Derived derived;
   std::cout << derived.z << std::endl;
   std::cout << derived.mulYZ() << std::endl;
   std::cout << derived.addXYZ() << std::endl;
   return 0;
}
```

Inheritance

- It is possible to override behaviors of base classes in derived classes. Look at the print functions.
  - ➢ Even it is possible to extend the existing functionality
- It is possible to hide and delete inherited members of the base class in the derived class without changing the access levels in the base class. (using delete and using)

```
#include <iostream>

class Base {
public: void print(void) const
        { std::cout << "Base" << std::endl; }
};

class Derived : public Base {
public:
   void print(void) const {
      Base::print();
      /* Adding to existing functionality */
      std::cout << "Derived " << std::endl;
   }
};

int main(void) {
   Derived derived;
   derived.print(); // calls derived::print(), which is public
   return 0;
}
```

```
class Base {
private: int value;
public:
   Base(int _value) : value{_value} {}
   int getValue(void) const { return value; }
   void print(void) const { std::cout << value; }
};

class Derived : public Base {
private: using Base::print;
public:
   Derived(int value) : Base{value} {}
   int getValue(void) = delete; // Make it inaccessible
};
```

```
int main(void) {
   Derived derived{7};
   // Error; because print() is private!
   derived.print();
   // Error; because getValue() has been deleted!
   std::cout << derived.getValue();
   return 0;
}
```

Inheritance

- To prevent inheriting from a class, the final specifier is applied after the class name

```
class A {
private: int x;
public:
   A(int _x): x{_x} {}
   int getX(void) { return x; }
};

class B final : public A { // Use of final specifier
private: int y;
public:
   B(int a, int b) : A{a}, y{b} {}
   int getY(void) { return y; }
};

class C : public B { // Error: cannot inherit from a final class
private: std::string name;
public:
   C(std::string str) : B{10, 20}, name{str} {}
   std::string getName() { return name; }
};
```

Upcasting derived classes to the base class

- Upcasting a derived class to its base class using a reference or a pointer

```
#include <string>
#include <iostream>

class Animal { /* Animal is not a real object. But it is a virtual object */
/* Making the following members  protected to prevent creating Animal
objects directly, but we want derived classes to be able to use it. */
   std::string name;
protected:  Animal(const Animal &) = default;
   Animal(std::string n) : name{n} {}
   Animal &operator=(const Animal &) = default;
public: std::string getName() const { return name; }
          std::string says() const { return "???"; }
};

class Cat : public Animal {
public: Cat(std::string name) : Animal{name} {}
          std::string says() const { return "Meow"; }
};

class Dog : public Animal {
public: Dog(std::string name) : Animal{name} {}
   std::string says() const { return "Woof"; }
};
```

```
void report(const Animal &ref)
{ std::cout << "Using a reference: " << ref.getName() << " says " << ref.says() << "\n"; }

void report(const Animal *ptr)
{ std::cout << "Using a pointer: " << ptr->getName() << " says " << ptr->says() << "\n"; }

int main(void) {
   const Cat cat{"Kiko"};
   std::cout << cat.getName() << " says " << cat.says() << std::endl;
   report(&cat); // Implicit upcasting using a pointer.
   report(cat);  // Implicit upcasting using a reference.

   const Dog dog{"Andre"};
   std::cout << dog.getName() << " says " << dog.says() << std::endl;
   report(&dog); // Implicit upcasting using a pointer.
   report(dog);  // Implicit upcasting using a reference.

   return 0;
}
```

Using ptr and ref we can only get access to the says function in Animal

Polymorphism

- A virtual function is a special type of non-static function member, when it is called
  - It resolves to the most-derived version of the function that exists between the base and derived class.
    - This capability is known as **polymorphism**
    - Polymorphism allows functions to use variables of different types at different times
- A derived function is considered as a match if it has the same signature and return type as the base version of the function. Such a function is called an **override**.

```cpp
#include <string>
#include <iostream>

class Base { // Using virtual in the base class
public: virtual std::string getName() const { return "Base"; }
};

class Derived : public Base { // Using override in the derived class
public: std::string getName() const override { return "Derived"; }
};
```

```cpp
int main(void) {
   Derived derived;

   Base &ref{derived};
   std::cout << "ref is a " << ref.getName() << '\n'; // Output: ref is a Derived

   Base *ptr{&derived};
   std::cout << "ptr is a " << ptr->getName() << '\n'; // Output: ptr is a Derived

   return 0;
}
```

Polymorphism

- Another example

```
#include <string>
#include <iostream>

class Animal { std::string name;
protected:
   // We're making these protected because
   // we don't want people creating Animal objects directly,
   // but we still want derived classes to be able to use it.
   Animal(const Animal &) = default;
   Animal(std::string _name) : name{_name} {}
   Animal &operator=(const Animal &) = default;
public:
   std::string getName() const { return name; }
   virtual std::string says() const { return "???"; }
};

class Cat : public Animal {
public:
   Cat(std::string name) : Animal{name} {}
   std::string says() const override { return "Meow"; }
};
```

```
class Dog : public Animal {
public:
   Dog(std::string name) : Animal{name} {}
   std::string says() const override { return "Woof"; }
};

void report(const Animal &ref)
{ std::cout << "Using a reference: " << ref.getName() << " says " << ref.says() << std::endl; }

void report(const Animal *ptr)
{ std::cout << "Using a pointer: " << ptr->getName() << " says " << ptr->says() << std::endl; }

int main(void) {
   const Cat cat{"Kiko"};
   std::cout << cat.getName() << " says " << cat.says() << std::endl;
   report(&cat); // Implicit upcasting using a pointer.
   report(cat);  // Implicit upcasting using a reference.

   const Dog dog{"Andre"};
   std::cout << dog.getName() << " says " << dog.says() << std::endl;
   report(&dog); // Implicit Dog upcasting using a pointer.
   report(dog);  // Implicit Dog upcasting using a reference.

   return 0;
}
```

Polymorphism

- Do not call virtual functions from constructors or destructors
  - When a Derived class is created, the Base portion is constructed first. If you call a virtual function from the Base constructor, it will be unable to call the Derived version of the function because there's no Derived object for the Derived function to work on.
  - If you call a virtual function in the Base class destructor, it will always resolve to the Base class version of the function, because the Derived portion of the class will already have been destroyed.

```cpp
#include <iostream>

class Base {
public:
  Base() { func(); } // Calling a virtual function in the constructor

  virtual void func() { std::cout << "Base" << std::endl; }

  ~Base() { func(); } // Calling a virtual function in the destructor
};

class Derived : public Base {
public:
  Derived() : Base() {}

  void func() override { std::cout << "Derived" << std::endl; }
};

int main(void) {
  Derived derived;
  // Normally we expect Derived::func() gets called.
  // But it does not. Instead Base::func() is called.
  return 0;
}
```

Polymorphism

- Use final to make a function unoverridable
  - E.g. void func() override final { }
- Make the base class destructor virtual if you're dealing with polymorphism.
- Because **base** is a Base pointer, when base is deleted, the program looks to see if the Base destructor is virtual. If it's not, it assumes that it only needs to call the Base destructor.
- If you want to make the default destructor virtual you can use default. E.g.

  virtual ~Base() = default;

```
#include <iostream>

class Base {
public:
   virtual ~Base() { std::cout << "Calling ~Base()" << std::endl;  }
};

class Derived : public Base {
private:  int num;
public:
   Derived(int x) : num{x} {}
   ~Derived() { std::cout << "Calling ~Derived()\n"; }
};

int main(void) {
   Derived *derived{new Derived{5}};
   Base *base{derived};

   // If destructors are not virtual, only the the Base destructor is called
   delete base;

   return 0;
}
```

Polymorphism

- The return type of a virtual function and its override must match.
  - Exception: If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class

```
class Base {
public:
   virtual int getValue() const { return 5; }
};

class Derived : public Base {
public:
   // Compilation error!
   // Conflicting return type specified for the virtual function
   double getValue() const override { return 6.78; }
};
```

```
#include <string>
#include <iostream>

class Base {
public:
   virtual Base *getThis() // Returns a pointer to a Base class
   { std::cout << "Called Base::getThis()"<< std::endl; return this; }
   void printType() { std::cout << "Returned a Base" << std::endl; }
};

class Derived : public Base {
public:
   // Because Derived is derived from Base, it's okay to return Derived* instead of Base*
   Derived *getThis() override
   { std::cout << "Called Derived::getThis()" << std::endl;  return this; }
   void printType() { std::cout << "Returned a Derived" << std::endl;  }
};

int main(void) {
   Derived derived{};
   derived.getThis()->printType(); // Calls Derived::getThis(), returns a Derived*, calls Derived::printType
   Base *ptr{&derived};
   ptr->getThis()->printType(); // Calls Derived::getThis(), returns a Base*, calls Base::printType
   return 0;
}
```

Polymorphism

- It is possible to ignore virtualization
- The downside of virtual functions - They are inefficient
  - Resolving a virtual function call takes longer than resolving a regular one.
  - The compiler shall allocate an extra pointer for class objects that have virtual functions
- To implement virtual functions, c++ uses a table of function pointers which is called virtual table.
- The compiler at compile time sets up a virtual table(a static array of function pointers) for each class which uses virtual functions or derived from a class that uses virtual functions.

```cpp
#include <iostream>

class Base {
public:
   virtual ~Base() = default;
   virtual const char *getName() const { return "Base"; }
};

class Derived : public Base {
public: const char *getName() const override { return "Derived"; }
};

int main(void) {
   Derived derived;
   const Base &base{derived};
   // Calls Base::GetName() instead of the virtualized Derived::GetName()
   std::cout << base.Base::getName() << std::endl;
   return 0;
}
```

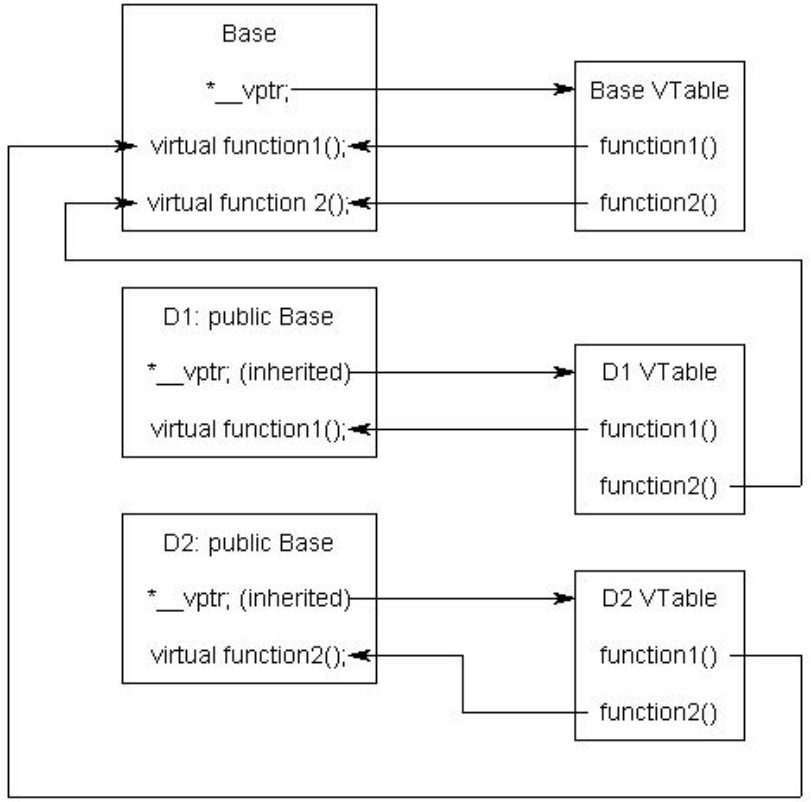
Polymorphism

- Each entry in a virtual table is simply a function pointer that points to the most-derived function accessible by that class.
- The compiler also adds a hidden pointer, called as **vpointer**, that is a member of the base class. **vpointer** is set automatically when a class instance is created so that it points to the **vtable** of the class.
- **vpointer** is inherited by derived classes
- Size of each instance is increased by size of one pointer.

```
class Base {
public:
   virtual void function1(){};
   virtual void function2(){};
};

class D1 : public Base {
public:
    void function1() override {};
};

class D2 : public Base {
public:
   void function2() override {};
};
```



Akademin

Polymorphism

- Virtualization causes a run-time overhead
  - Calling a virtual function is slower than calling a non-virtual function
  - Virtual tables and virtual pointers increase the required memory
- A **pure virtual(abstract) function** is a special kind of virtual function that has no body and it acts as a placeholder which shall be defined by derived classes.
- To create a pure virtual function, instead of defining a body for the function, assign **0** to the function.
- Any class which has one or more pure virtual functions is an **abstract base class**, and it cannot be instantiated
- Any derived class must define pure functions, or it will be an abstract base class as well.

```
class Base {
public:
   const char *sayHi() const { return "Hi"; } // A normal non-virtual function
   virtual const char *getName() const { return "Base"; } // A normal virtual function
   virtual int getValue() const = 0; // A pure virtual function
   int doSomething() = 0; // Compilation error: can not set non-virtual functions to 0
};
int main(void) {
   Base b; // Compilation error: Base cannot be instantiated
   return 0;
}
```

Polymorphism

- It is possible to define a pure virtual function. The body shall be outside of the class.
- The class is still an abstract base class and derived classes shall define the pure function

```
class Circle : public Shape {
private: double radius;
public:
   Circle(const std::string &name, double _radius) : Shape{name}, radius{_radius} {}
   double getArea() const override { return (3.1415 * radius * radius); }
   virtual ~Circle() = default;
};

class Square : public Shape {
private: double length;
public:
   Square(const std::string &name, double _length) : Shape{name}, length{_length} {}
   double getArea() const override { return (length * length); }
   virtual ~Square() = default;
};

void print(const Shape &shape) {
   std::cout << shape.getName() << " area is = "  << shape.getArea() << std::endl;
}
```

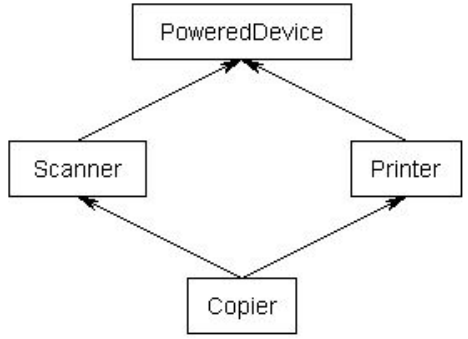
```
#include <string>
#include <iostream>
class Shape { // An abstract base class
protected: std::string name;
public:
   Shape(const std::string &_name) : name{_name} {}
   const std::string &getName() const { return name; }
   virtual double getArea() const = 0;
   virtual ~Shape() = default;
};

// Default implementation of a pure function outside of the class
double Shape::getArea() const { return 0.0; }

int main(void) {
   Circle c{"Circle", 10.2}; Square s{"Square", 10.2};
   print(c); print(s);
   return 0;
}
```

Polymorphism

- An interface class is a class has no variable members, and all of the functions are pure virtual
- Interfaces are useful when you want to declare functionalities that derived classes must implement
- Virtual base classes
  - A virtual base class is used to share a base class and ensure that only one instance of the base class is shared by derived classes. Means that the base object is shared between all objects in the inheritance tree and it is only constructed once.
  - In the example, the Copier constructor is responsible to call the constructor of PoweredDevice; not Printer or Scanner.




```
// An interface class
class IShape {
public:
   virtual void move_x(double dx) = 0;
   virtual void move_y(double dy) = 0;
   virtual void rotate(double angle) = 0;
   //...
};
```

```
class PoweredDevice { /* ... */ };
class Scanner : virtual public PoweredDevice { /* ... */ };
class Printer : virtual public PoweredDevice { /* ... */ };
class Copier : public Scanner, public Printer { /* ... */ };
```

Polymorphism

- dynamic_cast is used to safely convert a pointer/reference to an instance of a class to a pointer/reference of another class in the inheritance hierarchy during run-time.
- If a dynamic_cast fails and the target type is a
  - Pointer, then result of the conversion will be a null pointer.
  - Reference then an exception of type std::bad_cast will be thrown.

```
Base *createObject(bool derived) {
  return derived ? new Derived{1, "Apple"} : new Base{2};
}
int main(void) {
  bool b;
  std::cin >> b;
  Base *object{createObject(b)};
  Derived *ptr{dynamic_cast<Derived *>(object)};
  if (ptr != nullptr)
  {
     std::cout << "Name of the Derived is: " << ptr->getName() << std::endl;
  }
  delete ptr;
  return 0;
}
```

```
class Base {
protected: int value;
public:
   Base(int _value) : value{_value} {}
   virtual ~Base() = default;
};

class Derived : public Base {
protected: std::string name;
public:
   Derived(int value, const std::string &_name) :
                  Base{value}, name{_name} {}
   const std::string &getName() const { return name; }
   virtual ~Derived() = default;
};
```

Polymorphism

- Printing inherited classes using the stream insertion operator (<<)

```
#include <iostream>

class Base {
public:
   // The overloaded operator<<. Friend functions cannot be virtualized. Only member functions can be virtualized
   friend std::ostream &operator<<(std::ostream &out, const Base &base)
   { base.print(out); return out; /* Using print as a virtual member function to do the actual printing */ }
   virtual void print(std::ostream &out) const { out << "Base"; } // print is a normal member function, it can be virtualized
};

class Derived : public Base {
public:
   void print(std::ostream &out) const override { out << "Derived"; } // The override print function to handle the Derived case
};

int main(void) {
   Base b; std::cout << b << std::endl;
   Derived d; std::cout << d << std::endl;
   Base &r{d}; std::cout << r << std::endl;
   return 0;
}
```

C++ Language

- Some useful links
  - C++ Reference
  - Standard C++ Library Reference
  - C++ Tutorial
  - C++ Language
  - C++ Tutorial
  - C++ Full Course For Beginners (Learn C++ in 10 hours)
  - C++ Tutorial for Beginners - Full Course
  - C++ Tutorial 2021
  - C++ Programming Tutorials Playlist