Unit 5

Agenda

1. Objects

Objects

DEFINITION [Object] An object is an instance of a class. It can be uniquely identified by its name, it defines a state which is represented by the values of its attributes at a particular time and it exposes a behaviour defined by the set of functions (methods) that can be applied to it.

The simplest way to create an object in C++ is by variable declaration. If a variable is an instance of a class/struct then it is also called object.

```
int main(int, char*[]) {
     Date day1; // <- 1st object
     Date day2{25, 12}; // <- 2nd object

     Date *today = new Date{15, 03, 2004}; // <- 3rd object
     cout << “Today is "
          << today->getDay() << „/” << today->getMonth() << „/” << today->getYear();
     delete today;

     return 0;
}
```

Object Instantiation - Summary

#	Type	When is constructed	When is destroyed
1	**Local variable**	Each time its declaration is encountered in the execution of the program	Each time the program exits the block in which it occurs
2	**Local static variable**	First time its declaration is encountered in the execution of the program	Termination of the program
3	**Non-local object**	Once at the start of the program	Once at the termination of the program
4	**Dynamic allocation (Free-store)**	Using new operator	Using delete operator
5	**Array of objects**	When array is created	When array is destroyed
6	**Temporary object**	When the expression is evaluated	At the end of the full expression in which it occurs
7	**Union member**		
8	**Dynamic allocation in user-supplied memory**	Using an overloaded version of operator new	Using `delete` operator
9	**Non-static member object**	When the 'enclosing' object is created	When the 'enclosing' object is destroyed

Local variable

```
int f(int a) {
     Date d1, d2{03, 03, 2007}; // creating the objects

     if(a>0) {
          Date d3=d1; // creating object using copy-constructor
     } // d3 is destroyed

     Date d4;
} // destroying objects d4, d2, d1 (in this order)
```

Objects are destroyed in reverse order of their creation.

Local static variable

```
int f(int a) {

     static Date d{03, 03, 2007}; // static object

} // d is not destroyed!
```

Static objects are created when their declaration is executed first time only.

The destructors for static objects are called in reverse order when the program terminates.

Act as ‘global’ variables visible only to blocks where they are declared.

Non-local variable

File1.cpp

```
Date gdate1;

int f(int a) {
     cout << gdate1;
}

Date gdate2{1,1,1970};
```

File2.cpp

```
class X {
     static Date referenceDate;
};

Date X::referenceDate{1,1,1970};
```

Non-local = global, namespace or class static variables

Any global object (declared outside any function) is constructed before function `main` is invoked, in the order of their definitions.

The destructors for static objects are called in reverse order when the program terminates.

No implementation-independent guarantees are made about the order of construction/destruction of non-local objects in different compilation units (files).

Free store (Dynamic allocation)

```
int f(int a) {
     Date *d1 = new Date{03, 03, 2007}; // creating the object

     if(a>0) {
          Date *d2 = new Date;
          cout << *d2;
          delete d2;
     }

     delete d1;
}
```

Operator **new** – allocates and initializes memory from free store

Operator **delete** – de-allocates and cleanup memory

DO NOT FORGET TO DESTROY ALL OBJECTS ALLOCATED USING NEW OPERATOR !

Syntax

```
X* p = new X;

X* p = new X{init_value};


delete p;
```

Array of objects

```
int f(int a) {
     Date weekend[2]{{9,4,2016}, {10,4}};

     Date *da = new Date[3] {{1}, {2}, {3}};

     Date *year = new Date[365];

     delete [] year;
     delete [] da;
}
```

Syntax

```
X array[size];

X* p = new X[size];


delete [] p;
```

Before C++11:

There was no way to specify explicit arguments for a constructor in an array declaration, the default constructor being the only option and thus mandatory.

C++11 onwards:

It is possible to specify arguments matching any user-defined constructor.

The destructor for each element of an array is invoked when the array is destroyed.

Don't forget the squared brackets in delete to free **all** the elements of the array (`delete [] array`).

Temporary object

```
class String {
    char *s;
public:
    char* c_str() { return s; }

    // TODO: add the rest
    // of required members
};
```

```
void f(String& s1, String& s2) {
     // ...
     cout << s1 + s2;
     // ...
}
```

```
String temp = s1 + s2;
cout << temp;
destroy temp;
```

- Temporary objects are results of expression evaluation (e.g. arithmetic operations)
- A temporary object can be used as initializer for a const reference or named object:
  - `const string& s = s1+s2;`
  - `string s = s1+s2;`
- A temporary object can also be created by explicitly invoke the constructor
  - `handleComplex(complex(1,2));`
- Problems may arise. See below.

```
void f(String& s1, String& s2) {
     // c_str() returns a C-style, zero-terminated array of chars
     char* pch = (s1+s2).c_str();
     cout << pch;
     // pch points to an invalid memory address that was destroyed together with
     // the temporary obj. s1+s2
}
```

Union members

```
class X {
};

union AUnion {
   int x;
   char name[12];
   X obj; // OK: No constructors or destructor defined for type X
   Date date; // ERROR: Date has constructors and destructor

   void f();
};
```

A union can have member functions.

A union can't have static members.

A union can't have members with custom constructors or destructor.

User-supplied memory

```
void* operator new(size_t, void* p) {
     return p; // memory area
} // Explicit placement operator

void* buf = reinterpret_cast<void*>(0xF00F); // nasty, nasty!
// placement syntax
X* p2 = new (buf) X; // invokes operator new(sizeof(X), buf);
```

Accessing a specific memory address as an object of type X (useful in drivers development).

```
class PersistentStorage {
    virtual void* alloc(size_t)=0;
    virtual void free(void*)=0;
};

void* operator new(size_t s, PersistentStorage* p) {
     return p->alloc(s);
}

void destroy(X* p, PersistentStorage* storage) {
    p->~X(); // explicit call of destructor
    storage->free(p); // release the memory
}

PersistentStorage* ps = new FileStorage("a.bin");
```

```
void f() {
     X* p1 = new (ps) X;
     X* p2 = new (ps) X[10];
     X* p3 = new (ps) X(3);
    destroy(p1, ps);
    destroy(p2, ps);
    destroy(p3, ps);
}
```

Should be used with caution!

Non-static members (I)

```
class Student {
     Date dob; // non-static member
     String name; // non-static member
public:
     Student(const char* n, const Date& d);
};

Student::Student(const char* n, const Date& d) {
     dob = d;
     name = String{n};
}

void f() {
     Date d{7, 8, 1988};
     Student aStudent{„Popescu”, d};
}
```

Steps to initialize *aStudent* object:

(1) Memory allocation = sizeof(Date) + sizeof(String)

(2) Call default constructor for Date to initialize dob member.

(3) Call default constructor for String to initialize name member.

(4) Call the Student::Student(n,d) constructor.

  (5) Call assignment operator in line dob=d;

  (6) Call assignment operator in line name=n;

The “real” initialization of dob and name objects is done in 2 steps: a default constructor + an assignment operator (steps 2+5 and 3+6).

Non-static members (II)

```
class Student {
     Date dob; // non-static member
     String name; // non-static member
public:
     Student(const char* n, const Date& d);
};

Student::Student(const char* n, const Date& d)
     : name{n}, dob{d} {
}

void f() {
     Date d{7, 8, 1988};
     Student aStudent{„Popescu”, d};
}
```

Steps to initialize *aStudent* object:

(1) Memory allocation = sizeof(Date) + sizeof(String)

(2) Call the copy constructor of Date to initialize dob in `dob(d)`

(3) Call the custom constructor of String to initialize name in `name(n)`

(4) Call Student::Student(n,d)

The “real” initialization of dob/name objects is done in one step: calling the appropriate constructor.

Non-static members (III)

Order of initialization:

1. Initialize members in order of their declaration in class (the order in initialization list is irrelevant).
2. Call the constructor of the class.

Order of destroy:

1. Call the destructor of the class.
2. Call the members' destructor in reverse order of declaration.

The following members can be initialized only in initialization list:

- References (X& member;)
- const (const int member;)
- Types without default constructor (X member;)

static const members can be initialized at declaration as well.

```
class Club {
     Club(); // private default constructor
public:
     Club(const char* s);
};

class X {
     // declaration + initialization
     static const int ci = 1;

     const int i;
     Date& date;
     Club club;

public:
     X(int ii, Date& dd,
       const char* clubName)
          : i(ii),
            date(dd),
            club(clubName) {
          i = ii; // error – const member
     }
};

const int X::ci; // definition
Const float X::cf = 1.2;
```

Resource management (I)

```
void use_file(const char* fn) {
    FILE* fp = fopen(fn, “w”);
    // use fp
    fclose(fp);
}
```

Resource = file, memory, lock

Ensure proper release of resource!

Certainly, not in this example ;)

Solution 1:

```
void use_file(const char* fn) {
    FILE* fp;
    try {
        fp = fopen(fn, “w”);
        // use fp
    }
    catch(...) {
        fclose(fp);
        throw;
    }
    fclose(fp);
}
```

Problems with solution 1:

- tedious
- verbose
- potentially expensive
- error-prone

Resource management (II)

Solution 2 = **Resource acquisition is initialization (RAII)** (usage of local objects to allocate/release resources in constructors/destructors).

```
void use_file(const char* fn) {
    File_ptr f(fn, “w”);
    // use f
    fwrite(“Text”, 1, 4, f); // call operator FILE* to convert File_ptr to FILE*
}
```

```
class File_ptr { // smart pointer
public:
     File_ptr(const char* fn, const char* a) {
          : p{fopen(fn, a)} {
          if (p==nullptr) throw runtime_error{"File_ptr: Can't open file"};
     }
     File_ptr(FILE* pp) : p{pp} { }
     ~File_ptr() { fclose(p); }
     operator FILE*() { return p; }
private:
     FILE* p;
};
```

Advantages of the 2nd solution:

- the destructor will be called independently of whether normal or abnormal exit
- simpler main code and less error-prone (no need to call fclose – it’s done automatically in destructor)

Resource management (III)

Ideally, a well-designed constructor doesn't leave its object in some "half-constructed" state.

Applying r**esource acquisition is initialization** to object's members ensures transactional creation of objects, i.e. either an object is completely created or not at all!

```
class X {
public:
     X(const char* x, mutex& y)
          : f(x, "rw"), // acquire f
            lck(y)     // acquire lck
     { }
private:
     File_ptr f;
     Lock_ptr lck;
};
```

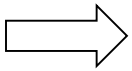
This implies that where this simple model for acquisition of resources is adhered to, the author of the constructor does not need to write explicit exception-handling code.

Standard containers of STL use RAII to implicitly manage acquisition and release.

Resource management (IV)

"Transactioning" memory allocation

```
class Y {
public:
     Y(int size) {
          p = new int [size];
          init();
     }
     ~Y() { delete [] p; }
private:
     int* p;
     void init();
};
```



```
class YY {
public:
     YY(int size) : p{size} {
          init();
     }
private:
     vector<int> p;
     void init();
};
```

Q: *Exceptions and new operator. What happens if X's constructors throws an exception? Is the memory freed?*

A: In the ordinary case, yes! But, if placement syntax is used the answer depends on the used allocator.

Example:

```
X* px = new (a) X [10]; // allocation from allocator a (deallocate it from a);
                        // depends on allocator
                        // Allocator's operator delete function is called.
```

Resource management (V)

Resource exhausting: when a resource acquisition fails (e.g. not enough memory)

Solutions:

- Resumption – ask the caller to fix the problem and carry on
- Termination – abandon the computation and return to some caller

Resumption – is implemented using function-call mechanism

Termination – is implemented using exception-handling mechanism

```
void* operator new(size_t size) {
     for(;;) {
          if(void* p = malloc(size)) return p;
          if(_new_handler==0) throw bad_alloc(); // termination
          _new_handler(); // resumption
     }
}
```

What information does the code causing a resource exhausting send to the caller? More information => increase dependency between each other. To preserve code reusability and to increase maintainability, the **coupling between different modules/components should be minimized**, but it must provide enough information so that the caller can recover from problem reliable and conveniently.

Further Reading

1. [Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 10, 11.5]

2. [Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 16.3]

Unit 6

Agenda

Class relationships

- Association
- Aggregation
- Composition
- Inheritance

Relationship types (I)

Concepts does not exist in isolation. They coexist and interact.

Association is a loose relationship in which objects of one class "*know*" about objects of another class (**has-a**)

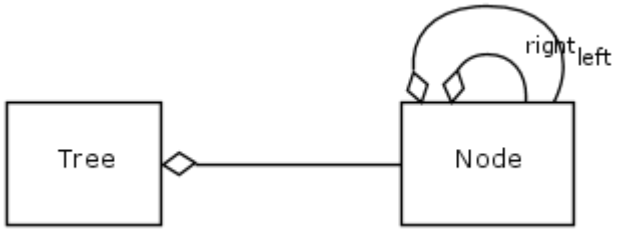
Aggregation is part-whole relationship (**is-part-of**)

Composition is similar to aggregation, but is more strict (**contains**)

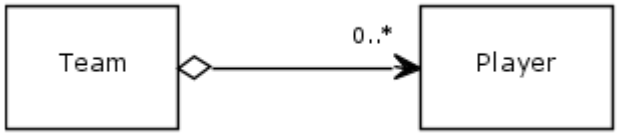
Inheritance is a generalization-specialization relationship (**is-a**, **kind-of**)

Relationship types (II)

Reflexive relationships: objects of the same class are related to each other



Directed relationship: refers to a directional relationship represented by a line with an arrowhead, which depicts a whole-part directional flow.



Associations (I)

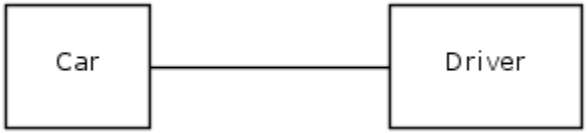
DEFINITION [Association] Association is a loose relationship in which objects of one class "*know*" about objects of the other class.

Association is identified by the phrase "**has a**".

Association is a static relationship.

Read relationships from left to right and from top to bottom.

Examples: A Car has a Driver.
A Student enrolls-in Course.



Associations (II)

Multiplicity - The multiplicity applies to the adjacent class and is independent of the multiplicity on the other side of the association.

Notation	Meaning
1	Exactly one
*	Zero or many
0..5	Zero to five
0..4,6,10	Zero to five, six or ten
	Exactly one (default)

Associations (III)

Remarks: complicated to maintain

Implementation: member variables (pointers or references) to the associated object.

```
typedef unsigned char uchar;

class Car;

class Driver {
     string name;
     Date birthDate;
     string licenseID;
     Car* car;

public:
     Driver(string name, Date& bd, string id);
};


class Car {
    string make;
    string model;
    int builtYear;
    Driver* driver;

public:
    Car(…., Driver* drv);

    Driver* getDriver() const;
    void setDriver(Driver* drv);
};
```

Aggregation and Composition

Both aggregation and composition represent a whole-part relationship.

Several definitions exist for aggregation and composition based on the following elements:

- Accessibility: The part objects are only accessible through the whole object.
- Lifetime: The part objects are destroyed when the whole object is destroyed.
- Partitioning: The whole object is completely partitioned by part objects; it does not contain any state of its own.

Aggregation (I)

DEFINITION [Aggregation] Aggregation is a relationship between part and whole in which:

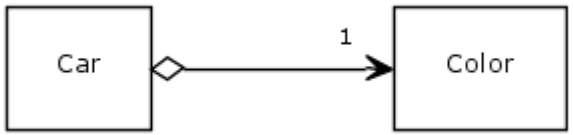
- **parts may exist independently of the whole;**
- **parts may be shared between two whole instances.**

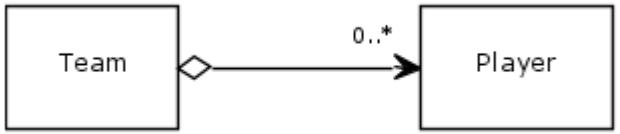
Aggregation is identified by the phrase "**is-part-of**".

Aggregation cannot be circular, i.e. an object cannot be part of itself.

Examples: Color is-part-of a Car.

A Player is-part-of a Team.





Aggregation (II)

```
typedef unsigned char uchar;

class Color {
     uchar red, green, blue;
public:
     Color(uchar r, uchar g, uchar b);
     uchar getRed() const {
          return red;
     }
     // etc.
};


class Car {
    string make;
    string model;
    int builtYear;
    Color& color; // aggregation (as reference)

public:
    Car(string& mk, string& mdl, int y,
        const Color& col);
    Color getColor() const;
    void setColor(const Color& c);
};
```

```
class Player {
     string name;
     Date dob;
public:
     Player(const char* n, const Date& d);
};


class Team {
    string name;
    string homeTown;
    int groundYear;
    PlayerNode* players; // aggregation (as linked list)

public:
    Team(string& n, string& t, int y);

    Player[] getPlayers() const;
    void addPlayer(const Player& p);
    void removePlayer(const Player& p);
};

struct PlayerNode {
     Player& player;
     PlayerNode* next;
};
```

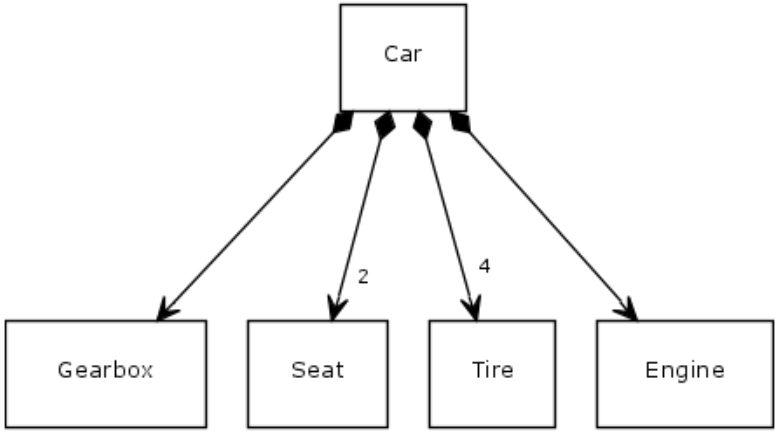
Composition

DEFINITION [Composition] Composition is a relationship between part and whole in which part may not be independent of the whole.

Composition is identified by the phrase "**contains**".

The contained object is destroyed once the container object is destroyed => No sharing between objects.

Example: A car contains an engine, four tires, two seats, and one transmission



Composition (II)

```
class Gearbox;

class Tire;

class Engine;

class Seat;


class Car {
    string make;
    string model;
    int builtYear;
    Color& color; // aggregation (as reference)

    Seat seats[2]; // composition 1 to many
    Tire tires[4]; // composition 1 to many
    Engine engine; // composition 1 to 1
    Gearbox gb;    // composition 1 to 1

public:
    Car(string& mk, string& mdl, int y,
        const Color& col);

    Color getColor() const;
    void setColor(const Color& c);

    Engine getEngine() const;
    Tire getTire(int position) const;
    // etc.
};
```

Aggregation vs. Composition

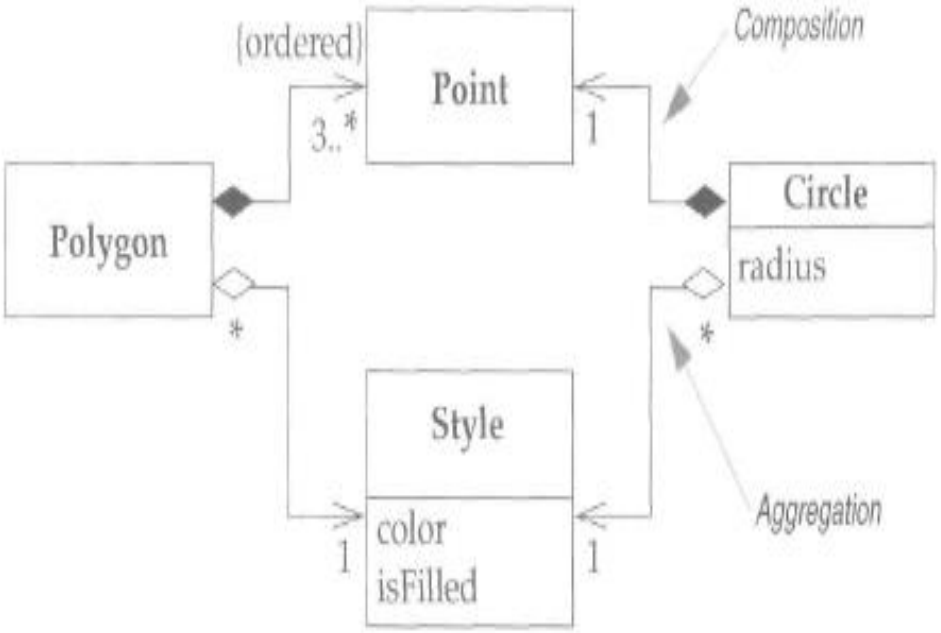


Figure 6-6: *Aggregation and Composition*

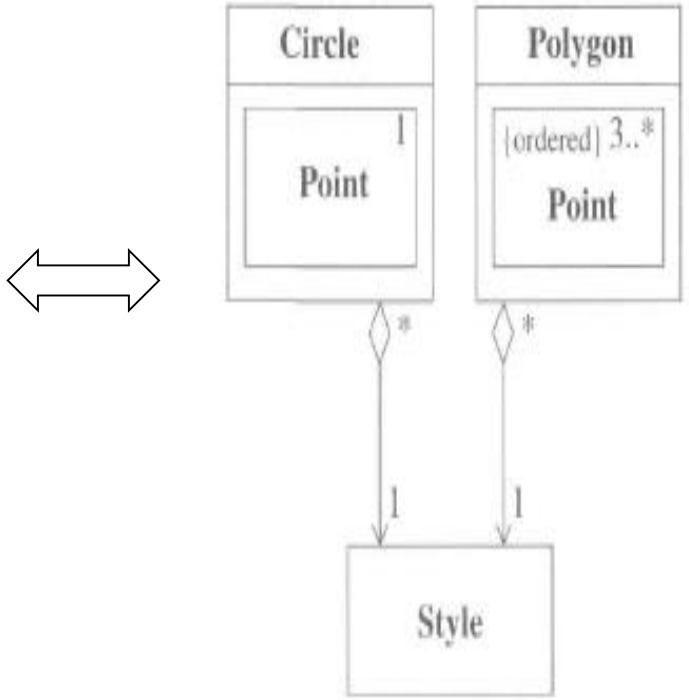


Figure 6-7: *Alternative Notation for Composition*

Source: [Fowler, 2000, Page 85-87]

Differences between aggregation, composition and associations still under debate in software design communities.

Further Reading

1. [Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 12]

2. [Sussenbach, 1999] Rick Sussenbach - Object-oriented Analysis & Design (5 Days), DDC Publishing, Inc. ISBN: 1562439820, 1999 [Chapter 5]

3. [Fowler, 2000] Martin Fowler with Kendall Scott – UML Distilled 2nd Ed, Addison-Wesley, 2000