High Level Programming

Classes

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- In C++ we define our own data structures by defining a class
  - The target is to define **class types** that behave as **built-in types** (i.e., C++ libraries)
- A class defines a **type** including
  - A set of objects
  - A collection of operations related to that objects
    - These operations are called member functions, i.e., functions defined inside a class, or methods

Introduction

Please, recall the course "Algorithms and Data Structures"

- The core ideas behind classes are
  - Data abstraction
    - Separetes the interface and the implementation
      - The **interface** specifies the operations the users can execute on the class
      - The **implementation** includes the data members and defines the body of the functions
  - Encapsulation
    - Enforces the **separation** between interface and implementation
      - Users of the class can see the interface but have no access to the implementation

Structs and unions

Structs and unions are two different ways of organizing and storing data in C++:

- C-like structures
  - Heterogenous data linked together by logical (problem based) constraints
  - There is no automatic data hiding

```
struct product {
  int weight;
  float price;
};
```

Standard C structure definition

Memory

A struct is a user-defined data type that allows you to group together related data under a single name. Each piece of data within a struct is called a member or a field. Structs are commonly used to represent records or objects that contain multiple fields of different data types.

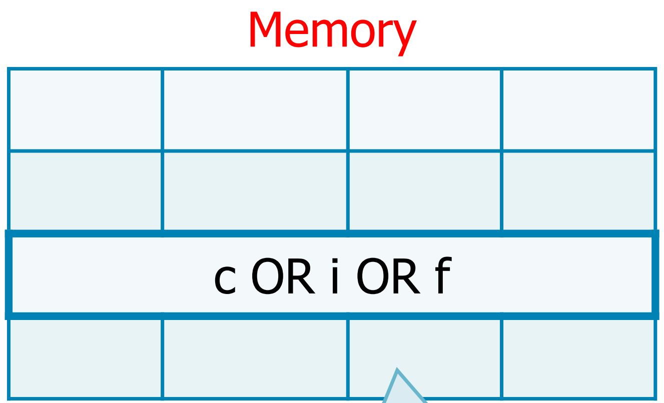
Structs and unions

A union is a special data type that allows you to store different types of data in the same memory location. Unlike structs, where each member has its own memory space, all members of a union share the same memory location.
Unions are useful when you need to represent a single piece of data in different ways, depending on the context.

- C-like unions
  - Single memory location to access the same "bit-level" configuration based on different types
    - The overall size is equal to the largest type in a union

```
union mytypes_t {
  char c;
  int i;
  float f;
};
```

All the types are merged together, and the size of the union is equal to the longer of the fields



We can do operations on integer (e.g., bit field operations) and interpreting the result as a float

Classes

- In C++ a class can be defined as a C structure or a proper C++ class
- In both cases, a class is a collection of
  - Data variables that can be
    - Public (visible from outside)
    - Private (not directly visible from outside)
  - Member functions or methods, i.e., functions
    - Must be **declared** inside the class
    - May be **defined** inside or outside the class
      - Try **not** to **mix** solutions in the same class (but in generic programming)
      - Mixed solutions makes the interface full of details and the implementation partial

Example: Version 1

A simple class

In C++ programming, it's common to split the definition and implementation of classes (or structs) into separate files:

Header File (.h or .hpp):
This file typically contains the class/struct declaration, including member variables, member functions' declarations (signatures), and any necessary includes.

Source File (.cpp):
This file contains the implementation of the member functions declared in the header file. It provides the actual code for the member functions, defining how they behave when called.
Example (my_class.cpp):

Main File:
This file typically contains the main() function and may include the header file to use the class/struct. It interacts with the class/struct by creating instances and calling its member functions.
Example (main.cpp):

Name of the class

C-like structure

```
struct my_class {
  int code;

  int get_code() {
    return (code);
  }

  void print_code();
};

void my_class::print_code () {
  cout << code << endl;
}
```

Inline method

Equivalent to:
return (this->code);
"this" is defined implicitly and refers to "this" object

External method

Method implementation:
The declaration and the definition must match

Access specifier

- One of the main issues with structures or classes is the visibility of the objects
  - We can use access specifiers to enforce encapsulation
  - Members defined after a
    - **Public** specifier are accessible to all parts of the program
    - **Private** specifier are accessible to the member functions of the class but not to the ones that use the class

Example: Version 2

Explicit access specifiers

```
struct my_class {
  private:
    int code;
  public:
    int get_code() {
      return (code);
    }
    void print_code();
};

void my_class::print_code () {
  cout << code << endl;
}
```

External method

Method implementation: The declaration and the definition must match

Encapsulation

- In C++ we often use the keyword **class** rather than **struct**
  - The only difference is the **default** access level
  - If we use the keyword
    - Struct, objects are **public** by default
      - **Struct** members defined before the first access specifiers are **public**
    - Class, objects are **private** by default
      - **Class** members defined before the first access specifiers are **private**

Encapsulation

- In a class, both data and member functions can have different access properties
  - Everything is private, by default, if no access specifier is present
- Encapsulation is one **key** concept of OOPs
  - Define objects as private as long as possible
  - Define methods as private whatever is needed
  - Define objects/methods as **protected** when they need to be inherited by sub-classes
    - Protected is like private but sub-classes can inherit objects

Example: Version 3

A stylistic choice: class and struct

Now we define a proper class

```
class my_class {
  private:
    int code;
  public:
    int get_code() {
      return (code);
    }
    void print_code();
};

void my_class::print_code () {
  cout << code << endl;
}
```

The key private can be erased (default)

Private and public objects can be interleaved, but it is cleaner to separate them

Classes end with a ";"

Sometimes, it is clearer to put the public objects before as we immediately see them

Example

Using a class type from the main

We will introduce strings in unit 04

```
class my_class {
  public:                 // Access specifier
    int myNum;            // Attribute (int variable)
    string myString;      // Attribute (string variable)
};

int main() {
  my_class myObj;
  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";
  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

Create an object of my_class

Write 15 then "Some text"

Example

Using more instantiations

```
class Car {
  public:
    string brand;
    string model;
    int year;
};
int main() {
  Car car1;
  car1.brand = "BMW";
  car1.model = "X5";
  car1.year = 1999;
  Car car2;
  car2.brand = "Ford";
  car2.model = "Mustang";
  car2.year = 1969;
  cout <<car1.brand<<" "<<car1.model<<" "<< car1.year<<"\n";
  cout <<car2.brand<<" "<<car2.model<<" "<< car2.year<<"\n";
  return 0;
}
```

Create an object

Create another object

Example

A class with an external method

```
class my_class {
  public:                    // Access specifier
    void my_method();        // Method/function declaration
};


// Method definition outside the class
void my_class::my_method() {
  cout << "Hello World!";
}


int main() {
  my_class my_obj;
  my_obj.my_method();
  return 0;
}
```

Create an object

Call the method

Example

Enforce encapsulation with getters and setters

```
#include <iostream>
using namespace std;
class Employee {
  private:
    // Private attribute
    int salary;
  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};
int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

Example

Inheritance

```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

Derivation access specifier: Control the access that users of the derived class have to the members of the base class

The derived class controls the members of the base class depending on the access specifier used to define them in the base class

Example

```
// Base class (parent)
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Derived class (child)
class MyChild: public MyClass {
};

// Derived class (grandchild)
class MyGrandChild: public MyChild {
};

int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

Multiple inheritance

Everything is public

New objects are also public

New-new objects are also public

Class initialization

- Classes are **instantiated** into objects
  - It is the only time when all data associated to the class is ever allocated in memory
  - Data is **not** shared among objects
    - For example, two instantiations of **my_class** have different **ptr** pointers
      - They may even refer to the same object, **but** they are anyway stored in different locations

```
class my_class {
  char *ptr;
  public
    int do_work();
};
```

  - Code **is** shared among objects
    - For example, two instantiations of my_class share the same **same code**

Class initialization

- Classes control what happens when we operate on objects, i.e., when we
  - Construct, copy, assign, or destroy an object of that class type
- Objects are
  - **Constructed**, when they are created
  - **Copied**, when we initialize a variable, we pass a value by variable, we return an object by value
  - **Assigned**, when we use the assignment operator
  - **Destroyed,** when they cease to exist

Class initialization

- If we do not define these operations the compiler will define them for us
  - There are cases in which the default **does not behave correctly**
  - One example of that, is when classes allocate resources that **reside outside** the class object
    - In these cases we must write those methods directly

We focus now on **constructors** and **destructors**.
More on this will follow in Unit 04

Constructors

- A constructor is a special function that initializes the object when it is created
- A constructor is characterized by
  - The same name of the class
  - No return
- Thanks to polymorphism
  - It is possible to have different constructors for the same object
  - Different constructors must have a different set of parameters (i.e., a different signature) in number and/or type

Constructors

- If the programmer does not define a constructor the compiler will implicitly define a default one
  - The default constructor initializes each data member as follows
    - If there is a in-class initializer, it runs the initializer
    - Otherwise, it initializes it with a default value
- Constructors are automatically invoked whenever an object is defined, i.e., when the class
  - Is explicitly defined
  - Receives a parameter by value
  - Returns a value
  - Is copied (a class instance)

Example

A class with two constructors

```
class my_class {
  private:
    int code;

  public:
    my_class(): code(0) {}

    my_class(int c) { code=c; }

    int get_code() {...}

    void print_code() {...}
};
```

Default constructor (no parameters)

New definition type:
After calling the constructor **code** will be defined and set to 0

Member initialization is a feature in C++ that allows you to initialize member variables of a class or struct directly in the constructor's initialization list rather than within the constructor body. It's often preferred over assigning values to members in the constructor body

Extra constructor (1 parameter)

After calling the constructor **code** will be defined and set to the parameter c

Example

An "hello world" constructor

```
class my_class {       // The class
  public:              // Access specifier
    my_class() {       // Constructor
      cout << "Hello World!";
    }
};

int main() {
  // Create an object of my_class
  // This will call the constructor
  my_class my_obj;
  return 0;
}
```

Create an object and print the message

Example

A constructor with parameters

```
class Car {
  public:              // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    // Constructor with parameters
    Car(string x, string y, int z) {
      brand = x;
      model = y;
      year = z;
    }
};
int main() {
  // Create cars and call the constructor
  Car c1("BMW", "X5", 1999);
  Car c2("Ford", "Mustang", 1969);

  // Print values
  cout <<c1.brand<< " " <<c1.model<< " " <<c1.year<< "\n";
  cout <<c2.brand<< " " <<c2.model<< " " <<c2.year<< "\n";
  return 0;
}
```

Destructors

- The destructor is a unique function that is deputed to clear any internal (dynamic) resources handled by the object before destruction
- There is only one destructor for each class
  - It has the exact name of the class with a ~ before
  - It has **no** parameters
    - Polymorphism is impossible on the destructor
  - It is **not** called directly by the user
    - Only the compiler schedules its calls, i.e., there is an automatic call, one for each abandoned object

With the constructor we have no direct call, but we decide which one to call (with the parameters)

Destructors

- There is no need of a destructor if the class handles only static resources
  - We only need a destructor to free **dynamic memory**, object descriptors, etc.
    - Same procedure used for containers (please, see unit 04)

Example

Constructor and destructor

```
#include <iostream>
using namespace std;
static int count = 0;
class Test {
  public:
    Test() {
      count++;
      cout << “#C: " << count << endl;
    }
    ~Test() {
      cout << “#D: " << count << endl;
      count--;
    }
};
int main() {
  Test t, t1, t2, t3;
  return 0;
}
```

Output

```
#C: 1
#C: 2
#C: 3
#C: 4
#D: 4
#D: 3
#D: 2
#D: 1
```

This sequence confirms that constructors are called in the order of object creation, and destructors are called in the reverse order.

Destructors are called in reverse order of object creation because of the nature of stack unwinding in C++. When objects are created, they are typically stored on the stack. As a function (such as main()) ends, the stack unwinds, and objects are destroyed in the reverse order of their creation. This ensures that objects that were created last are destroyed first, which helps maintain a consistent and logical order of resource management.

Example

```
class String {
  private:
    char* s;
    int size;
  public:
    String(char*); // constructor
    ~String();     // destructor
};
String::String(char *c) {
  size = strlen(c);
  s = new char[size + 1];
  strcpy(s, c);
}
String::~String() { delete[] s; }
int main() {
  String str = "Hello, World!";
  String myString(str);
  cout << "String: " << myString.s << endl;
  return 0;
}
```

Constructor and destructor

Strings are introduced in Unit 04

C-like string must be allocated (malloc→new) and de-allocated (free→delete)

Friend classes

- A class can allow another class (function) to access its non-public members by making that class (function) a friend
- Notes
  - Friendship is non-transitive and cannot be inherited
  - Access specifiers have no influence on friend declarations
    - They can appear in private or public sections

Reported for the sake of completeness. You may ignore it !

Friend classes

- A class makes a class (function) a friend by including a declaration for that class (function) preceded by the keyword friend
  - Friend declarations may appear only inside a class definition
  - They may appear anywhere in the class

Declares a function as a friend of the class

```
friend function_declaration;

friend function_definition;

friend class_specifier;
```

Defines a non-member function and declares it as a friend of the class

Declares another class as a friend of this class

Example

```
class A {
  int a;
  friend class B;
  friend void foo(A&);
};
```

Friend class

Friend method

```
void foo(A& a) {
    a.a=42;
}
```

Even if a is private
foo can access it

```
class B {
  friend class C;
  void foo(A& a) {
    a.a = 42;
  }
};
```

Class B is a friend of A
thus the definition of foo
is correct

```
class C {
  void foo(A& a) {
    a.a = 42;
  }
};
```

The friend attribute is not transitive
This definition of foo is invalid