Introduction to C++

Pt. 4

Emanuele Giona Department of Computer Science, Sapienza University of Rome

Internet of Things A.Y. 2022/23

Prof. Chiara Petrioli Department of Computer, Control and Management Engineering, Sapienza University of Rome
Dott. Michele Nati

Exercise 1: solution

```cpp
#include <iostream>
using namespace std;

void mergeArrays(int *array1, int *array2, int size, int *array3);

int main(){
  int a1[] = {1,2,3};
  int a2[] = {4,5,6};
  int a3[] = {0,0,0,0,0,0};
  mergeArrays(a1, a2, 3, a3);
  return 0;
}

void mergeArrays(int *array1, int *array2, int size, int *array3){
  int index, i;
  for(index=0, i=0; i<size; i++, index+=2){
    array3[index] = array1[i];
    array3[index + 1] = array2[i];
  }
}
```

Exercise 2: solution

```cpp
#include <iostream>
#include <vector>
using namespace std;

void mergeContainers(vector<int> vector1, vector<int> vector2, vector<int> &vector3);

int main(){
  vector<int> v1{1,2,3};
  vector<int> v2{4,5,6};
  vector<int> v3;
  mergeContainers(v1, v2, v3);
  return 0;
}

void mergeContainers(vector<int> vector1, vector<int> vector2, vector<int> &vector3){
  vector<int>::iterator it1;
  vector<int>::iterator it2;

  for(it1 = vector1.begin(), it2 = vector2.begin(); vector1.size() + vector2.size() != vector3.size();){
    if(it1 != vector1.end()){
      vector3.push_back(*it1);
      it1++;
    }
    if(it2 != vector2.end()){
      vector3.push_back(*it2);
      it2++;
    }
  }
}
```

Question: modifying a container during iteration on itself

```cpp
vector<int> v{0,1,2,3,4,5,6,7,8,9};
vector<int>::iterator it;
for(it = v.begin(); it != v.end();it++){
    if(it == v.begin() + 9){
        v.erase(it);
    }
}
```

Classic `for` loop with iterators

This program compiles, but will result in an infinite loop due to `erase` corrupting the iterator and resulting in an increment past the end.

```cpp
vector<int> v{0,1,2,3,4,5,6,7,8,9};
vector<int>::iterator it;
for(it = v.begin(); it != v.end();){
  if(it == v.begin() + 9){
    it = v.erase(it);
  }
  else
    it++;
}
```

Classic `for` loop with iterators and extra care

This loop instead takes advantage of the iterator returned by `erase`, which represents the updated location of the next element. This is the recommended way of implementing deletion during iteration.

```cpp
vector<int> v{0,1,2,3,4,5,6,7,8,9};
int index = 0;
for(int e : v){
  if(index == 9){
     v.erase(v.begin() + index);
  }
  index++;
}
```

Range-based `for` loop

This program compiles and does not result in infinite loop, but it a highly discouraged method to implement this feature as the underlying iterator might get corrupted, resulting in undefined behavior.

References: vector::erase, Stack Overflow #1, Techie Delight, Stack Overflow #2

1. struct

General data structures

A data structure in C++ is a group of data elements grouped together under one name. Each data element is known as member of a data structure, with its own type and length.

```
struct person {
    string name;
    float height;
    bool hasPassedIoT;
} a, b, c;

person d, e;
```

Names associated with `structs` are considered as types when declaring new variables

`struct` members are declared in the same way as normal variables.

Upon a new `struct` declaration, variables of that struct type can be declared as well by inserting their identifiers after the trailing right curly bracket (in the example: a, b, c).

Accessing `struct` members

Once a variable of a `struct` type is declared, its members can be accessed directly via the member access operator (`.`).

```
person a;
a.hasPassedIoT = true;
```

Initialization of `struct` variables can be done via curly brackets in several ways:

```
person a = {“A”, “170”, true};
person b{“B”, “165”, false};
person c = person{“C”, “153”, true};
```

Additionally, `struct` members can be provided default values:

```
struct person{
    string name = “”;
    float height = 100;
    bool hasPassedIoT = false;
}
```

Exactly how you would initialize standalone variables

Using `struct` in arrays and functions

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <limits>
using namespace std;

struct food{
  string name = "";
  float price = 0.f;
};

struct person{
  string name = "";
  food favorites[3];
};

person survey();
```

Function returning a `struct`
Parameters can also be of type `struct`

Array of `struct` elements

```cpp
int main(){
  int nPeople = 0;
  vector<person> people;
  cout << "Number of people to survey:" << endl;
  cin >> nPeople;
  cin.ignore (numeric_limits<std::streamsize>::max(), '\n');
  for(int i=0; i<nPeople; i++){
    person p = survey();
    people.push_back(p);
  }

  cout << people[0].name "'s first favorite food is '";
  cout << people[0].favorites[0].name << endl;

  return 0;
}

person survey(){
  person p;

  cout << "Name:" << endl;
  getline(cin, p.name);
  cout << "Top 3 favorite foods along with prices:" << endl;
  for(int i=0; i<3; i++){
    string fName = "";
    float price = 0.0f;
    getline(cin, fName);
    cin >> price;
    cin.ignore (numeric_limits<std::streamsize>::max(), '\n');
    p.favorites[i].name = fName;
    p.favorites[i].price = price;
  }

  return p;
}
```

`cin.ignore` required to flush the input stream from `'\n'`, when using plain `cin` and `getline` together

Pointers to `structs`

Similarly to variables of fundamental types, `struct` elements can be pointed at via pointers.

```
person p;                 ———— struct variable
person *pPtr = &p;        ———— Pointer to struct variable, initialized
```

Accessing members through pointers

Instead of using the normal access operator (`.`), in order to access members of a pointer to a struct variable, there is the arrow operator (`->`), which is an access operator made specifically for pointers to *elements that have members*:

```
pPtr->hasPassedIoT = true;
```

The arrow operator is thus equivalent to a derefenziation plus a normal member access (*e.g.* `(*pPtr).hasPassedIoT`), but it should not be mistaken for the derefenziation of a pointer member.

```
person p;                 ———— struct variable, with a pointer member foodPtr
*p.foodPtr;               ———— Value of variable pointed at by foodPtr, member of a struct variable
```

2. Classes

Classes

Classes are an expanded concept of data structures (`struct`), that can contain function members other than data members.

In OOP, an *object* is an instantiation of a `class`; for C++, an *object* is a variable and the `class` is its type.

```
class person {
    public:
        void walk(int metres);
    private:
        void sleep(float hours);
} a, b, c;
```

`class` declaration syntax is very similar to `struct`, also in the case of declaring a few variables along with it (*e.g.* a, b, c)

Access specifiers `public`, `private`, and `protected` are used to modify the access rights to `class` members

All members declared after an access specifier will apply its access rights; however, access specifiers are optional, with default access set to `private`.

More on access specifiers

- `private` (default)
  Only members of the same class can access these members
- `protected`
  Members of the same class and derived classes can access these members
- `public`
  These members can be accessed from anywhere this object is visible

```cpp
class person {
  public:
    void walk(int metres);
  private:
    void sleep(float hours);
};
```

```cpp
int main(){
  person a;
  a.walk(10000);
  a.sleep(3.5);
  return 0;
}
```

Compile-time error: the `sleep()` function member cannot be accessed from the `main()` function

Accessing `class` members and functions

Accessing members

Similarly with `structs`, `class` members can be accessed through the member access operator (`.`) as long as the correct access rights hold.

Function declaration and definition

Function members can either be defined entirely or just declared:

- Full definition within class declaration
  Class member function will be automatically considered an inline function (faster execution time)
- Function declaration only within class declaration
  Normal (not inline) class member function

```
void person::walk(int metres){
  // TODO
}

void person::sleep(float hours){
  // TODO
}
```

Class member function definitions can be placed in implementation files.

Syntax for outside definitions must be consistent with function signatures, and the `class` must be explicitly specified using the scope operator (`::`).

Initializing objects

Classes can include a special member function that is automatically called upon creation of a new object. Such function is the constructor.

A constructor function has the same name as its class, with no return type (not even `void`), and initialize class members with default values or allocate storage; it is executed exactly once, at object creation.

Overloading constructors

A constructor can be overloaded with different versions, each with different parameters. The default constructor is the one not taking any parameter, which will be the one called upon object creation.

```cpp
class Polygon{
  int nSides;
  float *sides;

  public:
    Polygon();
};
```

```cpp
class Polygon{
  int nSides;
  float *sides;

  public:
    Polygon();
    Polygon(int n, float *s);
};
```

Constructor invocation

In C++ there are multiple ways of invoking constructors:

- Functional form
  Using parentheses ( ), enclose the constructor’s parameters at the moment of variable declaration
  e.g. `Polygon p(2, sidesArray);`
- Single parameter
  For constructors with a single parameter, use the assignment operator (=) to pass the parameter
  e.g. `myClass obj = expression statement;`
- Uniform initialization^{C++11}
  Very similar to functional form, but use curly brackets { } instead of parentheses
  e.g. `Polygon p{2, sidesArray};`

No real differences among which invocation is used, just a matter of programming style.

Constructor implementation

When implementing constructors, there are several ways to initialize class data members.

```
Polygon::Polygon(int n, float *s){
  nSides = n;
  sides = s;
}
```

```
Polygon::Polygon(int n, float *s)
  : nSides(n)
{
    sides = s;
}
```

```
Polygon::Polygon(int n, float *s)
  : nSides(n),
    sides(s)
{}
```

Similarly with the invocation, there are no differences in how the constructor is implemented and its effects on members' initializations.

Destructor functions

In the opposite way a constructor function initializes class members, a destructor function takes care of performing all the operations that should be carried out at the deletion of an object.

```cpp
class Polygon {
  int nSides;
  float *sides;

  public:
    Polygon();
    Polygon(int n);
    ~Polygon();
};

Polygon::Polygon(){
  Polygon(0);
}

Polygon::Polygon(int n)
  : nSides(n)
{
  sides = new float[nSides];
}

Polygon::~Polygon(){
  delete [] sides;
}
```

Destructor function declaration

Constructor invocation within another constructor

Destructor releases memory that has previously been allocated for a dynamically-allocated array

Pointers to `classes`

Everything already applicable to `struct` variables can be used on `class` objects.

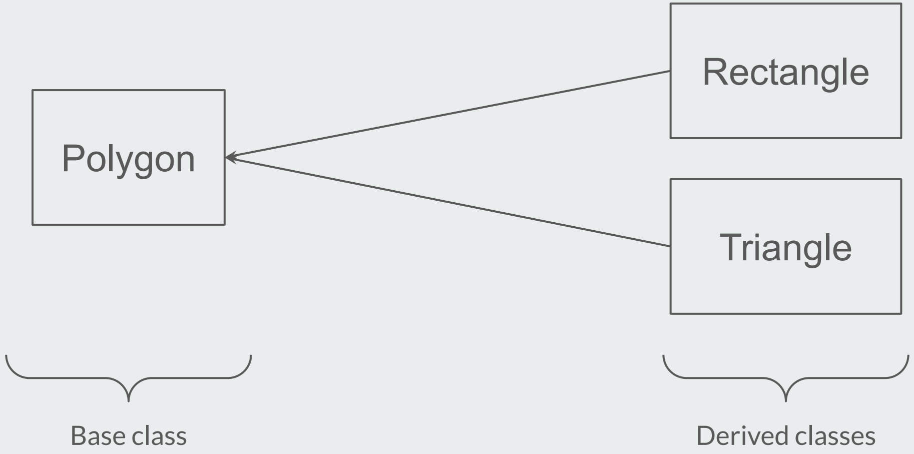
```
Polygon p;
Polygon *pPtr = &p;
delete pPtr;
```

Member access is always done via the member access operator (.) on any object, otherwise the arrow operator is used on any pointer to an object.

When using the `delete` statement on a pointer to an object, the destructor function will be invoked on that object.

Inheritance

In OOP, inheritance is the mechanism through which a class can retain some characteristics of the base class. This applies to data and function members.



Base class generally contains members that are common among its derived classes, whereas each derived class will extend with specific members or implementations for their functionality.

Inheritance in C++

In order to specify a class derives from a base class, this is the syntax to use in its declaration:

```
class Rectangle : public Polygon {
    /* rest of class declaration */
};
```

In C++, access specifiers can be used to provide more granularity to inheritance:

- `public`
  All inherited members will keep their original access rights from the base class
- `protected`
  All inherited members will become at most `protected`
- `private`
  All inherited members will become at most `private`

If no access specifier is explicitly stated, the default is `private`. However, in most cases the `public` inheritance is used.

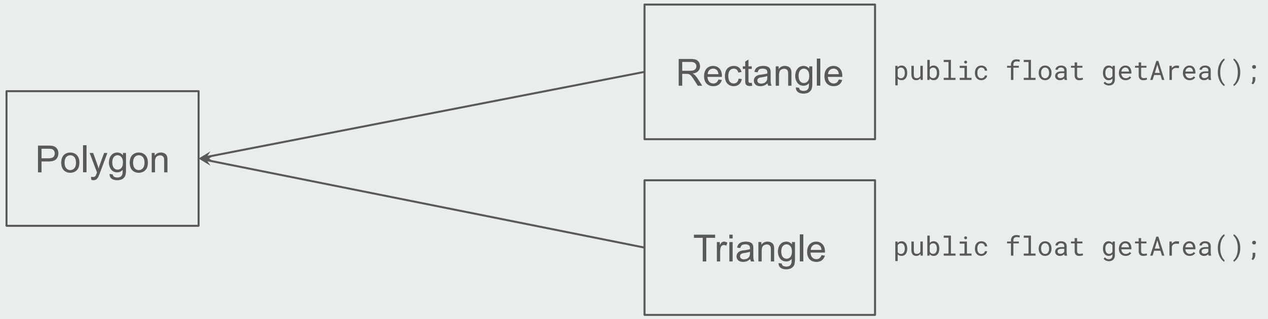
Inheritance in C++

Differently from other languages, C++ supports multiple inheritance: a derived class can extend multiple base classes, simply by adding more base classes and separating them by commas:

```
class Rectangle : public Polygon, public ConvexFigure {
    /* rest of class declaration */
};
```

Polymorphism

In OOP, polymorphism is a feature taking advantage of type-compatibility between base and derived classes. By obscuring specific implementations into the derived classes, it is possible to operate on objects of base classes only and still retain the specific behavior.



In order to leverage implementations for function `getArea()` from derived classes `Rectangle` and `Triangle`, but only accessing `Polygon` objects, we must use additional care.

Polymorphism in C++

```cpp
class Polygon {
  protected:
    float width, height;

  public:
    Polygon();
    Polygon(float w, float h);
    virtual ~Polygon();

    virtual float getArea() = 0;
};

Polygon::Polygon()
{
}

Polygon::Polygon(float w, float h)
  : width(w),
    height(h)
{
}

Polygon::~Polygon(){
}
```

Base class

- Function member providing specific implementation depending on the derived class (*e.g.* `getArea`) should be declared with `virtual` modifier
- In case the base class provides a default implementation, its declaration is exactly the same as a normal function member
- Otherwise in case the base class does not provide it, the function is declared as pure virtual: the trailing `= 0` has to be added in the declaration and no definition has to be provided

Any class containing pure virtual functions is also called an abstract class.

Polymorphism in C++

Derived classes

```cpp
class Rectangle : public Polygon {
  public:
    Rectangle(float w, float h);
    ~Rectangle();
    float getArea();
};

Rectangle::Rectangle(float w, float h)
  : Polygon(w, h)
{
}

Rectangle::~Rectangle()
{
}

float Rectangle::getArea(){
  return width * height;
}
```

```cpp
class Triangle : public Polygon {
  public:
    Triangle(float w, float h);
    ~Triangle();
    float getArea();
};

Triangle::Triangle(float w, float h)
  : Polygon(w, h)
{
}

Triangle::~Triangle()
{
}

float Triangle::getArea(){
  return width * height / 2;
}
```

They should invoke the constructor of the base class in order to initialize its members.

Additionally, if the base class has pure virtual functions, all derived classes must either declare them as pure virtual again or provide a definition.

Polymorphism in C++

Main function

```cpp
int main()
{
  Rectangle r = Rectangle(2,3);
  Triangle t = Triangle(2,3);
  Polygon *p1 = &r;
  Polygon *p2 = &t;

  cout << p1->getArea() << endl;
  cout << p2->getArea() << endl;

  return 0;
}
```

Objects of derived classes can be accessed through pointers of the base class. Beware that only members of the base class will be accessible in this way, regardless of the original object behind the pointer.

References

- https://www.cplusplus.com/doc/tutorial/structures/
- https://www.cplusplus.com/doc/tutorial/classes/
- https://www.cplusplus.com/articles/2LywvCM9/
- https://www.cplusplus.com/doc/tutorial/inheritance/
- https://www.cplusplus.com/doc/tutorial/polymorphism/