Clases en C++

Carlos E. Alvarez[1].

[1]Dep. de Matemáticas aplicadas y Ciencias de la Computación, Universidad del Rosario

2019-II

Implementación de ADT

Los tipos de datos abstractos (ADT), como las estructuras de datos, están generalmente compuestos por variables, arreglos, operaciones, etc.

Para implementar ADTs, debemos encapsular de manera coherente sus diferentes componentes, para lo que podemos usar conceptos de programación orientada a objetos (OOP) por medio de las clases.

Estructuras

Usadas para encapsular colecciones de variables relacionadas (atributos).

Definición de una `struct`

```
struct Point { //Name of the struct
  double x;  //attribute
  double y;  //attribute
};
```

Estructuras

Declaración y uso

```
int main(){
  Point p;
  p.x = 0.;
  p.y = 1.;

  cout << "(" << p.x << "," << p.y << ")\n";

  return 0;
}
```

Note que aquí `p` es una variable tipo `Point`.

Clases

Al igual que las estructuras, son tipos definidos por el usuario que se usan para encapsular variables (atributos), pero adicionalmente encapsulan *comportamientos* (métodos).

Clases

Interfase de `class`

```
class Vector2D { //Name of the class
private:
  double x;  //attribute
  double y;  //attribute

public:
  Vector2D();     //constructor
  ~Vector2D();    //destructor

  double norm(); //method
};
```

Clases

Implementación de `class`

```
Vector2D::Vector2D(){ //constructor
  x = 0.;
  y = 0.;
}

Vector2D::~Vector2D(){}   //destructor

double Vector2D::norm(){  //method
  return sqrt(x*x + y*y);
}
```

Clases

- Palabra `private`: Los atributos y métodos declarados en esta sección sólo pueden ser accedidos por otros miembros de la clase
- Palabra `public`: Los atributos y métodos declarados en esta sección pueden ser accedidos por cualquier función

Clases

```
int main(){
  Vector2D myVec;  //Declare object of type Vector2D

  myVec.x = 2.;    //ERROR attr. x is private!
  cout << myVec.y; //ERROR attr. y is private!

  cout << myVec.norm(); //O.K. norm() is public

  return 0;
}
```

Clases

- Constructor: Este método indica como inicializar las instancias de la clase. Puede existir mas de un constructor
- Destructor: Este método indica que hacer cuando la instancia sale de rango (scope) y es destruida

Clases: Ejemplo DynArray

Interfase (dynarray.hpp)

```
class DynArray {
private:
  int *array;
  int size;

public:
  DynArray(int s);
  ~DynArray();

  void set(int pos, int val);
  int get(int pos);
  int getsize();
};
```

Clases: Ejemplo DynArray

Constructor (dynarray.cpp)

```
DynArray::DynArray(int s){
  size = s;
  array = new int[s];
  for(int i = 0; i < size; ++i)
    array[i] = 0;

  cout << "DynArray instance created\n";
}
```

Clases: Ejemplo DynArray

Destructor (dynarray.cpp)

```
DynArray::~DynArray(){
  delete[] array;

  cout << "DynArray instance destroyed\n";
}
```

Clases: Ejemplo DynArray

Otros métodos (dynarray.cpp)

```
void DynArray::set(int pos, int val){
  array[pos] = val; //no bounds checking!
}

int DynArray::get(int pos){
  return array[pos]; //no bounds checking!
}

int DynArray::getsize(){
  return size;
}
```

Clases: Ejemplo DynArray

Uso (main.cpp)

```
int main(){
  int N = 10;
  DynArray myArray(N);

  for(int i = 0; i < N; ++i)
    myArray.set(i, 10*i);

  for(int i = 0; i < N; ++i)
    cout << myArray.get(i) << " ";
  cout << "\n";

  return 0;
}
```

Sobrecarga de operadores

- Es posible redefinir el comportamiento de operadores como `+`, `*`, == y otros, cuando los operandos son instancias de clases
- Vamos a redefinir el operador == para poder comparar objetos tipo `DynArray`
- Vamos a redefinir el operador + para poder sumar objetos tipo `DynArray`

Sobrecarga de operadores: Sobrecargando (==)

En la interfase...

```
bool operator==(DynArray& a);
```

En la implementación...

```
bool DynArray::operator==(const DynArray& a){
  if(size == a.getsize()){
    for(int i = 0; i < size; ++i){
      if(array[i] != a.get(i))
        return false;
    }
  }else{
    return false;
  }
  return true;
}
```

Sobrecarga de operadores: Sobrecargando (+)

En la interfase...

```
DynArray operator+(DynArray& a);
```

En la implementación...

```
DynArray DynArray::operator+(DynArray& a){
  int minsize = size < a.getsize() ? size : a.getsize();
  DynArray result(minsize);

  for(int i = 0; i < minsize; ++i)
    result.set(i, array[i] + a.get(i));

  return result;
}
```

Sobrecarga de operadores

Ejercicios:

- Sobrecargue el op. $(\star)$ de manera que al multiplicar dos DynArray del mismo tamaño, retorne el producto punto entre ellos

Sobrecarga de operadores: Sobrecargando (<<)

- Quisiéramos tener un método como `__str__` de python para imprimir nuestro `DynArray` usando `cout`
- Sin embargo el operador << no opera solo entre objetos tipo `DynArray`, y no vamos a definirlo como miembro de la clase
- Para que una función externa tenga acceso a los miembros de una clase, usamos la palabra **`friend`**

Sobrecarga de operadores: Sobrecargando (<<)

En la interfase...

```
friend ostream& operator<<(ostream& os, DynArray& a);
```

En la implementación...

```
ostream& operator<<(ostream& os, DynArray& a){
  os << "(";
  for(int i = 0; i < a.getsize()-1; ++i)
    os << a.get(i) << ",";
  os << a.get(a.getsize()-1) << ")\n";
  return os;
}
```

Operadores miembro vs. no miembro

- Los operadores asignación (=), subscript (`[]`), llamado (`()`) y acceso a miembro (->) deben ser definidos como métodos miembro
- Los operadores de I/O deben ser definidos como funciones externas
- Es recomendado que los operadores relacionales (<,==,...) y aritméticos (+,*,...) sean funciones externas, para que sean simétricas
- Es recomendado que los operadores de cambio de estado (++,&,...) sean miembros