Clases en C++

[1] Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

TDAs - Clases en C

```
1  struct Vector {
2      int *data;
3      int size;
4  };
```

```
15 void f() {
16     struct Vector v;
17     vector_create(&v, 5);
18     vector_get(&v, 0);
19     vector_destroy(&v);
20 }
```

```
5  void vector_create(struct Vector *v, int size) {
6      v->data = (int*)malloc(size*sizeof(int));
7      v->size = size;
8  }
9  int vector_get(struct Vector *v, int pos) {
10     return v->data[pos];
11 }
12 void vector_destroy(struct Vector *v) {
13     free(v->data);
14 }
```

TDAs - Clases en C

```c
struct Vector {
    int *data;
    int size;
};

void vector_create(struct Vector *v, int size) {
    v->data = (int*)malloc(size*sizeof(int));
    v->size = size;
}
int vector_get(struct Vector *v, int pos) {
    return v->data[pos];
}
void vector_destroy(struct Vector *v) {
    free(v->data);
}

void f() {
    struct Vector v;
    vector_create(&v, 5);
    vector_get(&v, 0);
    vector_destroy(&v);
}
```

- La forma de hacer TDAs en C aplica a C++
- El estándar de C++ garantiza que si un struct no tiene ningun feature de C++ (o sea, se parece a un struct de C) se lo llama "plain struct" y puede ser usado por libs de C desde C++ (el estándar garantiza la compatibilidad)
- Nótese como se requiere usar "struct" cada vez que se hacer referencia a Vector en C, pero no asi en C++.
- También como las funciones del TDA deben tener un nombre único para no entrar en conflicto con otros TDAs y funciones. El conflicto de nombres es un problema común en proyectos grandes en C.
- Asi también es convención pasar como primer argumento un puntero al struct

Keyword struct implícita

y adios a los typedefs de los structs

```
1  struct Vector {
2      int *data;
3      int size;
4  };
```

```
5  void vector_create(Vector *v, int size) {
6      v->data = (int*)malloc(size*sizeof(int));
7      v->size = size;
8  }
9  int vector_get(Vector *v, int pos) {
10     return v->data[pos];
11 }
12 void vector_destroy(Vector *v) {
13     free(v->data);
14 }
```

```
15 void f() {
16     Vector v;
17     vector_create(&v, 5);
18     vector_get(&v, 0);
19     vector_destroy(&v);
20 }
```

Keyword struct implícita
y adios a los typedefs de los structs

```
1  struct Vector {
2      int *data;
3      int size;
4  };
```

```
15 void f() {
16     Vector v;
17     vector_create(&v, 5);
18     vector_get(&v, 0);
19     vector_destroy(&v);
20 }
```

```
5  void vector_create(Vector *v, int size) {
6      v->data = (int*)malloc(size*sizeof(int));
7      v->size = size;
8  }
9  int vector_get(Vector *v, int pos) {
10     return v->data[pos];
11 }
12 void vector_destroy(Vector *v) {
13     free(v->data);
14 }
```

- En C++ no es necesario usar la keyword struct en todos lados.

Programación orientada a objetos

Keyword this como un parámetro implícito: un puntero a la instancia

```
1  struct Vector {
2      int *data;
3      int size;
4
5      void vector_create(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int vector_get(int pos) {
11         return this->data[pos];
12     }
13
14     void vector_destroy() {
15         free(this->data);
16     }
17 };
```

Programación orientada a objetos
Keyword this como un parámetro implícito: un puntero a la instancia

```
1  struct Vector {
2      int *data;
3      int size;
4
5      void vector_create(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int vector_get(int pos) {
11         return this->data[pos];
12     }
13
14     void vector_destroy() {
15         free(this->data);
16     }
17 };
```

- Se integran las funciones y los datos del TDA en una sola unidad.
- Los datos del TDA se lo llaman atributos
- Las funciones del TDA pasan a ser métodos del TDA y reciben como parámetro implícito un puntero a la instancia
- El puntero es un puntero constante a la instancia (Vector *const) y se lo nombra con la keyword this

Reducción de colisiones

Las clases crean sus propios namespaces

```
1  struct Vector {
2      int *data;
3      int size;
4
5      void create(int size) { // Vector::create
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) { // Vector::get
11         return this->data[pos];
12     }
13
14     void destroy() { // Vector::destroy
15         free(this->data);
16     }
17 };
```

Reducción de colisiones
Las clases crean sus propios namespaces

```
1  struct Vector {
2      int *data;
3      int size;
4
5      void create(int size) { // Vector::create
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) { // Vector::get
11         return this->data[pos];
12     }
13
14     void destroy() { // Vector::destroy
15         free(this->data);
16     }
17 };
```

- Los métodos de un TDA no entran en conflicto con otros aunque se llamen iguales. El método get de Vector no entra en conflicto con el método get de Matrix, por ejemplo
- En rigor un método de un TDA se lo llama NombreTDA::NombreMetodo, por eso Vector::get es distinto de Matrix::get.
- Veremos con mas detalle el concepto de namespace en próximas clases.

Invocación de métodos

```
17 void f() {
18     Vector v;
19     v.create(5);
20     v.get(0); // llamada a un metodo
21
22     v.data; // acceso a un atributo
23
24     v.destroy();
25 }
```

Invocación de métodos

```
17 void f() {
18     Vector v;
19     v.create(5);
20     v.get(0); // llamada a un metodo
21
22     v.data; // acceso a un atributo
23
24     v.destroy();
25 }
```

- Se accede a los atributos y/o métodos como en C

Permisos de acceso

Controlan quien puede acceder a los atributos y métodos de un objeto.

```
1  struct Vector {
2      private:
3      int *data;
4      int size;
5
6      public:
7      void create(int size) {
8          this->data = (int*)malloc(size*sizeof(int));
9          this->size = size;
10     }
11
12     int get(int pos) {
13         return this->data[pos];
14     }
15
16     void destroy() {
17         free(this->data);
18     }
19 };
```

Permisos de acceso

Controlan quien puede acceder a los atributos y métodos de un objeto.

```
1  struct Vector {
2      private:
3      int *data;
4      int size;
5
6      public:
7      void create(int size) {
8          this->data = (int*)malloc(size*sizeof(int));
9          this->size = size;
10     }
11
12     int get(int pos) {
13         return this->data[pos];
14     }
15
16     void destroy() {
17         free(this->data);
18     }
19 };
```

- Por default, un struct tiene sus atributos y métodos públicos. Esto significa que pueden accederse desde cualquier lado.
- Se puede cambiar el default forzando distintos permisos.
- private hace que solo los métodos internos puedan acceder a los métodos y atributos privados.
- Mas sobre los permisos public/protected/private y su relación con la herencia en las próximas clases.

Permisos de acceso

```
17 void f() {
18     Vector v;
19     v.create(5);
20
21     v.set(0, 1);
22     v.get(0);
23
24     v.data[0] = 1;
25 }
```

Clases en C++

Son iguales que los structs solo que el acceso es privado por default.

```
1 struct Vector {
2     int *data;  // public
3     int size;   // public
4 };
5
6 class Vector {
7     int *data;  // private
8     int size;   // private
9 };
```

Clases en C++

Son iguales que los structs solo que el acceso es privado por default.

```
1 struct Vector {
2     int *data;  // public
3     int size;   // public
4 };
5
6 class Vector {
7     int *data;  // private
8     int size;   // private
9 };
```

- Absolutamente todo lo visto con structs en C++ aplica a las clases de C++. La única diferencia es que las clases tienen sus atributos y métodos privados por default.

Clases en C++

Son iguales que los structs solo que el acceso es privado por default.

```
1  class Vector {
2      private:
3      int *data;
4      int size;
5
6      public:
7      void create(int size) {
8          this->data = (int*)malloc(size*sizeof(int));
9          this->size = size;
10     }
11
12     int get(int pos) {
13         return this->data[pos];
14     }
15
16     void destroy() {
17         free(this->data);
18     }
19 };
```

Constructor y Destructor

Código a ejecutar de forma automática cuando se construye/destruye un objeto.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) {
11         return this->data[pos];
12     }
13
14     ~Vector() {
15         free(this->data);
16     }
17 };
```

Constructor y Destructor

Código a ejecutar de forma automática cuando se construye/destruye un objeto.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) {
11         return this->data[pos];
12     }
13
14     ~Vector() {
15         free(this->data);
16     }
17 };
```

- El constructor es un código que se ejecuta al momento de crear un nuevo objeto. C++ siempre llama a algun constructor al crear un nuevo objeto.
- Todos los objetos son creados por un constructor. Si un TDA no tiene un constructor, C++ crea un constructor por default
- Un TDA puede tener múltiples constructores (que los veremos a continuación). Sin embargo solo puede haber un único destructor.
- Un destructor es un código que se ejecuta al momento de destuirse un objeto (cuando este se va de scope o es eliminado del heap con delete).
- Todos los objetos tienen un destructor. Si un TDA no tiene un destructor, C++ crea un destructor por default

Reduciendo las probabilidades de errores

```
20 // en C
21 void g() {
22     struct VectorB v;
23     // puedo usar el vector antes de inicializarlo!
24     // ...
25
26     vectorB_create(&v, 5);
27
28     // puedo olvidarme de destruirlo tambien!
29 }

30 void f() {
31     Vector v(5); // objeto inicializado
32     v.get(0);
33 } // se llama al destructor ~Vector automaticamente aqui
```

Reduciendo las probabilidades de errores

```
20 // en C
21 void g() {
22     struct VectorB v;
23     // puedo usar el vector antes de inicializarlo!
24     // ...
25
26     vectorB_create(&v, 5);
27
28     // puedo olvidarme de destruirlo tambien!
29 }

30 void f() {
31     Vector v(5); // objeto inicializado
32     v.get(0);
33 } // se llama al destructor ~Vector automaticamente aqui
```

- Con los constructores (si estan bien escritos) no se puede usar un objeto sin inicializar
- Con los destructores (si estan bien escritos, se usa RAII y usamos el stack) no vamos a tener leaks.

Manejo de errores en C (madness)

Ejemplo motivador para el buen uso de objetos en C++ con constructores y destructores bien diseñados.

```
1  int process() {
2      char *buf = (char*) malloc(sizeof(char)*20);
3
4      FILE *f = fopen("data.txt", "rt");
5      if (!f) { free(buf); return -1;}
6
7      fread(buf, sizeof(char), 20, f);
8
9      if (error) {
10         fclose(f);
11         free(buf);
12         return -1;
13     }
14
15     fclose(f);
16     free(buf);
17     return 0;
18 }
```

Manejo de errores en C (madness)

Ejemplo motivador para el buen uso de objetos en C++ con constructores y destructores bien diseñados.

```
1  int process() {
2      char *buf = (char*) malloc(sizeof(char)*20);
3
4      FILE *f = fopen("data.txt", "rt");
5      if (!f) { free(buf); return -1;}
6
7      fread(buf, sizeof(char), 20, f);
8
9      if (error) {
10         fclose(f);
11         free(buf);
12         return -1;
13     }
14
15     fclose(f);
16     free(buf);
17     return 0;
18 }
```

- En C hay que chequear los valores de retorno para ver si hubo un error o no.
- En caso de error se suele abortar la ejecución de la función actual requiriendo previamente liberar los recursos adquiridos
- El problema esta en que es muy fácil equivocarse y liberar un recurso aun no adquirido u olvidarse de liberar un recurso que si lo fue.
- No solo es una cuestión de leaks de memoria. Datos corruptos por archivos o sockets mal cerrados o leaks en el sistema operativo son otros factores que no se solucionan simplemente reiniciando el programa.

RAII - Resource Acquisition Is Initialization

El constructor adquiere el recurso mientras que el destructor lo libera.

```
1  struct Buffer {
2      Buffer(int size) {
3          this->data = (char*)malloc(size*sizeof(char));
4      }
5      ~Buffer() {
6          free(this->data);
7      }
8  };
9
10 struct File {
11     File(const char *name, const char *flags) {
12         this->f = fopen(name, flags);
13     }
14     ~File() {
15         fclose(this->f);
16     }
17 };
```

RAII - Resource Acquisition Is Initialization

El constructor adquiere el recurso mientras que el destructor lo libera.

```
1  struct Buffer {
2      Buffer(int size) {
3          this->data = (char*)malloc(size*sizeof(char));
4      }
5      ~Buffer() {
6          free(this->data);
7      }
8  };
9
10 struct File {
11     File(const char *name, const char *flags) {
12         this->f = fopen(name, flags);
13     }
14     ~File() {
15         fclose(this->f);
16     }
17 };
```

- La idea es simple, si hay un recurso (memoria en el heap, un archivo, un socket) hay que encapsular el recurso en un objeto de C++ cuyo constructor lo adquiera e inicialize y cuyo destructor lo libere.
- Nótese como la clave esta en el diseño simétrico del par constructor-destructor.
- Vamos a refinar el concepto RAII en las próximas clases

RAII + Stack: No leaks

La instanciación de objetos RAII en el stack simplifica muchísimo el trabajo en C++.

```
1  int process() {
2      Buffer buf(20);
3
4      File f("data.txt", "rt");
5      if (f.failed()) { return -1;}
6
7      f.read(buf, sizeof(char), 20, f);
8
9      if (error) {
10         return -1;
11     }
12
13     return 0;
14 }
```

RAII + Stack: No leaks

La instanciación de objetos RAII en el stack simplifica muchísimo el trabajo en C++.

```
1  int process() {
2     Buffer buf(20);
3
4     File f("data.txt", "rt");
5     if (f.failed()) { return -1;}
6
7     f.read(buf, sizeof(char), 20, f);
8
9     if (error) {
10        return -1;
11    }
12
13    return 0;
14 }
```

- Al instanciarse los objetos RAII en el stack, sus constructores adquieren los recursos automáticamente
- Al irse de scope cada objeto se les invoca su destructor automáticamente y por ende liberan sus recursos sin necesidad de hacerlo explícitamente
- El código C++ se simplifica y se hace más robusto a errores de programaciónotesen.
- RAII + Stack es uno de los conceptos claves en C++
- Veremos mas sobre RAII, manejo de errores y excepciones en C++ en las próximas clases

Pasaje por referencia

En C todo se pasa por copia. En C++ podemos usar referencias (y no emularla con punteros).

```
1  // en C
2  int f() {
3      int i = 0;
4      inc(&i);
5
6      return i; // i = 1;
7  }
8
9  void inc(int *i) {
10     *i = *i + 1;
11 }
```

```
1  // en C++
2  int f() {
3      int i = 0;
4      inc(i);
5
6      return i; // i = 1;
7  }
8
9  void inc(int &i) {
10     i = i + 1;
11 }
```

Pasaje por referencia

En C todo se pasa por copia. En C++ podemos usar referencias (y no emularla con punteros).

```c
// en C
int f() {
    int i = 0;
    inc(&i);

    return i; // i = 1;
}

void inc(int *i) {
    *i = *i + 1;
}
```

```c
// en C++
int f() {
    int i = 0;
    inc(i);

    return i; // i = 1;
}

void inc(int &i) {
    i = i + 1;
}
```

- En C todo se pasa por copia. Si queremos pasar por referencia en realidad se pasa por copia un puntero.
- En C++ podemos usar el pasaje por referencia.

Diferencias entre referencias y punteros

```
1 int i = 1;
2 int j = 2;
3
4 int *p; // Sin inicializar
5
6 p = &i;
7 p = &j; // No constante
```

```
1 int i = 1;
2 int j = 2;
3
4 int &p = i; // Inicializado
5             // y constante
```

Diferencias entre referencias y punteros

```
1 int i = 1;
2 int j = 2;
3
4 int *p; // Sin inicializar
5
6 p = &i;
7 p = &j; // No constante
```

```
1 int i = 1;
2 int j = 2;
3
4 int &p = i; // Inicializado
5             // y constante
```

- Las referencias en C++ deben ser inicializadas al construirse y una vez que referencian a algun objeto no pueden referenciar a otro.
- Las referencias funcionan como alias y el compilador en algunos casos ni siquiera reservara memoria para una referencia.
- En cambio, los punteros pueden crearse sin inicializar, cambiar de objeto al que apuntan y siempre consumen memoria.

Pasaje por movimiento (Move semantics)

Para el próximo cuatrimestre. Es un feature de C++11 muy útil. Esta por fuera del programa, pero el que este interesado puede consultar.

Pasaje por copia

```
1 Vector f(Vector v) {
2     return v;
3 }
```

Pasaje por copia

```
1 Vector f(Vector v) {
2     return v;
3 }
```

- Al igual que en C, C++ pasa todos los objetos por copia por default
- Aunque util en algunas situaciones, el pasaje por copia es uno de los cuellos de botella mas grandes en terminos de performance que tienen los programas en C/C++

Copia bit a bit naive

La implementación por default

- La copia tanto en C como en C++ es bit a bit y funciona bien para objetos simples.
- Pero cuando un objeto (A) hace referencia a otro (B) a través de un puntero, la copia bit a bit genera una copia (C) que termina apuntando a B: la copia fue superficial y no en profundidad (deep copy).
- Aunque en algunos casos es deseable, la mayoria de las veces el hecho de que dos objetos (A y C) apunten ambos a otro (B) termina con problemas.
- No solo hay que ver con detenimiento las relaciones A–>B a través de punteros sino tambien de otros tipos de referencias como los file descriptors.

Constructor por copia

Crea un nuevo objeto copiando a otro.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector(const Vector &other) {
6          this->data = (int*)malloc(other.size*sizeof(int));
7          this->size = other.size;
8
9          memcpy(this->data, other.data; this->size);
10     }
11
12 };
```

Constructor por copia

Crea un nuevo objeto copiando a otro.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector(const Vector &other) {
6          this->data = (int*)malloc(other.size*sizeof(int));
7          this->size = other.size;
8
9          memcpy(this->data, other.data; this->size);
10     }
11
12 };
```

- Para crear un objeto nuevo a partir de otro se invoca al constructor por copia.
- Como cualquier otro constructor, el constructor por copia tiene una initialization list para pasarle argumentos a los constructores de sus atributos
- Todos los objetos en C++ son copiables por default. Si un objeto no tiene un constructor por copia, C++ le creara un constructor por copia por default que implementa una copia bit a bit naive

Copia de objetos

Crear un objeto copiando otro o reasignar los datos de un objeto ya creado copiando a otro.

```
1  Vector f(Vector v) {
2      Vector a(v);
3      Vector b = v;
4
5      Vector c(5);
6
7      c = v;
8
9      return v;
10 }
```

Copia de objetos

Crear un objeto copiando otro o reasignar los datos de un objeto ya creado copiando a otro.

```
1  Vector f(Vector v) {
2      Vector a(v);
3      Vector b = v;
4
5      Vector c(5);
6
7      c = v;
8
9      return v;
10 }
```

- En la línea 1 se recibe por copia un vector al que llamaremos v.
- En la línea 2 y 3 se crean 2 vectores más copiandose de v, ambos llaman al constructor por copia.
- En la línea 9 se retorna un vector por copia también.
- En la línea 7 sucede algo distinto. El vector c copia el contenido del vector v. Pero el objeto c ya estaba creado asi que en vez de llamar al constructor por copia llama al operador asignación.

Copia por asignación

Un objeto ya creado copia el contenido de otro.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector& operator=(const Vector &other) {
6          if ( this == &other) {
7              return *this; // other is myself!
8          }
9
10         free(this->data);
11         this->data = (int*)malloc(other.size*sizeof(int));
12         this->size = other.size;
13         memcpy(this->data, other.data; this->size);
14
15         return *this;
16     }
17 };
```

Copia por asignación

Un objeto ya creado copia el contenido de otro.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector& operator=(const Vector &other) {
6          if ( this == &other) {
7              return *this; // other is myself!
8          }
9
10         free(this->data);
11         this->data = (int*)malloc(other.size*sizeof(int));
12         this->size = other.size;
13         memcpy(this->data, other.data, this->size);
14
15         return *this;
16     }
17 };
```

- Para copiar el contenido de un objeto en otro ya creado se usa el operador asignación.
- Como el objeto this ya esta creado, debemos recordar que todos sus atributos estan ya creados: no podemos cambiar ninguno de sus atributos constantes.
- Todos los objetos en C++ son copiables por asignación asi que si un objeto no implementa la sobrecarga del operador asignación, C++ le creara una implementación por default que hara una copia bit a bit naive.
- También es posible que nos asignemos a nosotros mismos (haciendo vec = vec;). Debemos programar el operador asignación de tal forma que evite copiarse a si mismo.
- El operador asignación no es el único operador que se puede sobrecargar. Ya veremos otros y en más detalle en las próximas clases.

Objetos no copiables

Hay objetos que no tiene sentido que se puedan copiar. Forzar esta prohibición.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      // declaramos privados pero no definimos
6      // si alguien quiere compiar esto tendra un
7      // error en la etapa de compilacion / linkeo.
8      private:
9      Vector(const Vector &other);
10     Vector& operator=(const Vector &other);
11
12 };
```

Objetos no copiables

Hay objetos que no tiene sentido que se puedan copiar. Forzar esta prohibición.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      // declaramos privados pero no definimos
6      // si alguien quiere compiar esto tendra un
7      // error en la etapa de compilacion / linkeo.
8      private:
9      Vector(const Vector &other);
10     Vector& operator=(const Vector &other);
11
12 };
```

- Si no queremos que un objeto sea copiable se pueden declarar y definir el constructor por copia y el operador asignación y que su implementación sea fallar (lanzar una excepción). El intento fallido de copia se detecta en runtime.
- Otra forma seria declarar pero no definir ni el constructor por copia ni el operador asignación y hacerlos privados. El intento fallido de copia se detecta en tiempo de compilación y linkeo.

Métodos constantes

Nosotros garantizamos que no modifican el estado interno del objeto.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) const {
11         return this->data[pos];
12     }
13
14     ~Vector() {
15         free(this->data);
16     }
17 };
```

Métodos constantes

Nosotros garantizamos que no modifican el estado interno del objeto.

```
1  struct Vector {
2      int *data;
3      int size;
4
5      Vector(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) const {
11         return this->data[pos];
12     }
13
14     ~Vector() {
15         free(this->data);
16     }
17 };
```

- Un método constante es un método que no modifica el estado interno del objeto. Esto es, no cambia ningún atributo ni llama a ningún método salvo que este sea también constante.
- Sirve para detectar errores en el código en tiempo de compilación: si un método no modifica el estado deberia poderse ponerle la keyword const; si el compilador falla es por que hay un bug en el código y nuestra hipotesis de que el método no cambiaba el estado interno del objeto es errónea.

Objetos constantes

Sólo se pueden invocar métodos que sean constantes.

```
17 void f() {
18     Vector v(5);
19
20     v.set(0, 1); // no const
21     v.get(0); // const
22 }

24 void f() {
25     const Vector v(5); // objeto constante
26
27     v.set(0, 1); // no const
28     v.get(0); // const
29 }
```

Const como promesa

Una función que promete no modificar el objeto pasado como parámetro

```
17 void f() {
18     Vector v(5);
19
20     g(v);
21 }
22
23 void g(const Vector &v) {
24     v.set(0, 1); // no const
25     v.get(0); // const
26 }
```

Const como promesa

Una función que promete no modificar el objeto pasado como parámetro

```
17 void f() {
18     Vector v(5);
19
20     g(v);
21 }
22
23 void g(const Vector &v) {
24     v.set(0, 1); // no const
25     v.get(0); // const
26 }
```

- Es comun recibir parámetros constantes. La función promete que no va a cambiar al objeto recibido como parámetro.

Atributos constantes

Atributos que no cambian su valor una vez instanciados.

```
1  struct Vector {
2      int *data;
3      const int size;
4
5      Vector(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) const {
11         return this->data[pos];
12     }
13
14     ~Vector() {
15         free(this->data);
16     }
17 };
```

Atributos constantes

Atributos que no cambian su valor una vez instanciados.

```
1  struct Vector {
2      int *data;
3      const int size;
4
5      Vector(int size) {
6          this->data = (int*)malloc(size*sizeof(int));
7          this->size = size;
8      }
9
10     int get(int pos) const {
11         return this->data[pos];
12     }
13
14     ~Vector() {
15         free(this->data);
16     }
17 };
```

- También podemos tener atributos constantes. Estos toman un valor cuando se crean y lo mantienen durante toda la vida del objeto.

Initialization list

Pasaje de argumentos a los constructores de los atributos.

```
1  // Mal
2  Vector(int size) {
3      // atributos "data" y "size" ya estan construidos.
4      // aca solo los re-asigno
5      this->data = (int*)malloc(size*sizeof(int));
6      this->size = size;
7  }
8
9  // Bien
10 Vector(int size) : size(size) {
11     this->data = (int*)malloc(size*sizeof(int));
12 }
```

Initialization list

Pasaje de argumentos a los constructores de los atributos.

```
1  // Mal
2  Vector(int size) {
3      // atributos "data" y "size" ya estan construidos.
4      // aca solo los re-asigno
5      this->data = (int*)malloc(size+sizeof(int));
6      this->size = size;
7  }
8
9  // Bien
10 Vector(int size) : size(size) {
11     this->data = (int*)malloc(size+sizeof(int));
12 }
```

- Al ejecutarse el cuerpo del constructor todos sus atributos ya estan creados.
- Si se necesita construir alguno o todos sus atributos con parámentros especiales hay que usar la initialization list.
- Esto es útil no solo para crear objetos que no pueden cambiar una vez construidos (como los atributos const y las referecias) sino que también es necesario si queremos construir otros objetos con parámetros custom, sean nuestros atributos o nuestros ancestros (herencia).

Initialization list

Pasaje de argumentos a los constructores de los atributos.

```
1  // Mal, no compila
2  struct Tuple {
3      Vector vec;
4
5      Tuple(int n) {
6          vec(n) /*??*/
7      }
8
9      ~Tuple() {
10     }
11
12
13 };
```

```
1  // Bien, sin problemas
2  struct Tuple {
3      Vector vec;
4
5      Tuple(int n) :
6              vec(n) {
7      }
8
9      ~Tuple() {
10     } // llama al
11       // destructor
12       // Vector::~Vector
13 };
```

Unidades de compilación

```
1  struct Vector {
2      int *data;
3      const int size;
4
5      Vector(int size);
6      int get(int pos) const;
7      ~Vector();
8  }; // en el archivo vector.h
```

```
1  #include "vector.h"
2  Vector::Vector(int size) : size(size) {
3      this->data = (int*)malloc(size*sizeof(int));
4  }
5
6  int Vector::get(int pos) const {
7      return this->data[pos];
8  }
9
10 Vector::~Vector() {
11     free(this->data);
12 } // en el archivo vector.cpp
```

Unidades de compilación

```
1  struct Vector {
2      int *data;
3      const int size;
4
5      Vector(int size);
6      int get(int pos) const;
7      ~Vector();
8  }; // en el archivo vector.h
```

```
1  #include "vector.h"
2  Vector::Vector(int size) : size(size) {
3      this->data = (int*)malloc(size*sizeof(int));
4  }
5
6  int Vector::get(int pos) const {
7      return this->data[pos];
8  }
9
10 Vector::~Vector() {
11     free(this->data);
12 } // en el archivo vector.cpp
```

- Hasta ahora se integró en un solo lugar el código de cada método. Es mas simple pero trae problemas de performance del proceso de compilación.
- Para evitar recompilar una y otra vez el código de los métodos se le define en un archivo .cpp separado de las declaraciones del .h

Operadores new y delete: uso del heap en C++

```
1 Vector *vec = (Vector*) malloc(sizeof(Vector));
2 free(vec);
```

```
1 Vector *vec = new Vector(5);
2 delete vec;
```

```
1 Vector *vecs = new Vector[10];
2 delete[] vecs;
```

Operadores new y delete: uso del heap en C++

```
1 Vector *vec = (Vector*) malloc(sizeof(Vector));
2 free(vec);
```

```
1 Vector *vec = new Vector(5);
2 delete vec;
```

```
1 Vector *vecs = new Vector[10];
2 delete[] vecs;
```

- Las funciones malloc y free reservan y liberan memoria pero no crean objetos (no llaman a los constructores ni los destruyen)
- El operador new y su contraparte delete no solo manejan la memoria del heap sino que también llaman al respectivo constructor y destructor.
- Para crear un array de objetos hay que usar los operadores new[] y delete[] y la clase a instanciar debe tener un constructor sin parámetros.

Referencias I


Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley, Third Edition.