Manual 5 - 2do Torneo de Programación Competitiva

Lions R.C.

Agosto 2019

Contents

1 Programación orientado a objetos

Como hemos visto anteriormente es conveniente trabajar con grupos de datos en structs en lugar de guardar los datos en pares de pares o tenerlos sueltos. Cada instancia de un struct se puede conocer como un objeto, todos los objetos tienen la caracteristica de ser conformado por distintos datos y se pueden crear diferentes instancias de un objeto con valores distintos. Un objeto puede ser visto como una base que puede ser duplicado y que mantiene sus datos de manera estructurada.

Como hemos visto con los structs cada objeto puede ser instancializado o destruido (esto tiende a ocurrir despues de que termina nuestro programa), y cada objeto puede ser declarado con un nombre unico.

1.1 Uniones

La unión es un objeto parecido a un struct con la unica diferencia que solo permite que el usuario guarde un dato en todas sus variables en un cierto tiempo para ahorrar espacio de memoria. Esta estructura siempre ocupará el tamaño de la variable más grande de la lista de todas sus variables.

Por ejemplo, si se tiene un int, un float y un long long int en un struct este ocuparía un espacio de 16 bytes para guardar los 4 bytes del int, los 4 bytes del float y los 8 bytes del long long int, pero una union con estas tres variables solo ocupará 8 bytes porque este es el tamaño del dato mas grande (el long long int).

Una desventaja de este tipo de dato es que no se puede saber cual es la variable que esta siendo guardado, asi que se debe usar otra variable o tener un contexto especifico. Debido a estas limitaciones la union es una estructura raramente usada.

Aqui se presenta el ejemplo de un código que almacena una dirección (arriba, abajo, izquierda o derecha). Como siempre se tendran dos arreglos distintos con uno para guardar direcciones horizontales y uno que guarda direcciones verticales, entonces se puede crear una union que guarda arriba o abajo en un booleano o derecha o izquierda en otra:

Listing 1: Uniones

```
#include <iostream>

using namespace std;

union Direccion {
    bool arriba;
    bool derecha;
};

int main() {
    Direccion verticales[4];
    Direccion horizontales[4];
    for(int i = 0; i < 4; i++) {
        cin >> verticales[i].arriba;
    }
    for(int i = 0; i < 4; i++) {
        cin >> horizontales[i].derecha;
    }
    for(int i = 0; i < 4; i++) {
        if(verticales[i].arriba) {
            cout << "arriba" << endl;
        } else {
            cout << "abajo" << endl;
        }
    }
    for(int i = 0; i < 4; i++) {
        if(horizontales[i].derecha) {
            cout << "derecha" << endl;
        } else {
            cout << "izquierda" << endl;
        }
    }
}
```

1.2 Clases

Las clases son objetos especiales que te permiten guardar datos como un struct, pero además pueden tener funciones internos y variables privadas. Una variable

pública es una que puede ser modificada desde afuera de la clase mientras que una privada tiene que ser modificada desde una llamada interna. Para definir la privacidad de una clase se debe escribir **public:** antes de la lista de las variables y funciones públicas y **private:** antes de la lista de funciones y variables privadas.

Por ahora solo crearemos una clase con variables públicas utilizando una estructura parecida a un struct:

Listing 2: Declarando clases

```
#include <iostream>

using namespace std;

class Cliente {
    public:
        string nombre;
        float dinero;
};

int main() {
    Cliente pablo;
    pablo.nombre = "Pablo Cesar";
    pablo.dinero = 12.1;
    cout << pablo.nombre << " tiene $" << pablo.dinero;
}
```

Algo que habiamos mencionado es que las clases pueden tener sus propios funciones, asi que crearemos uno para ver cuanto dinero tiene cada cliente y otro para sumarle una cantidad de dinero a su cuenta:

Listing 3: Funciones internas

```
#include <iostream>

using namespace std;

class Cliente {
    public:
        string nombre;
        float dinero;

        void imprimeSaldo() {
            cout << nombre << " tiene $" << dinero << endl;
        }

        void deposita(float cantidad) {
            dinero += cantidad;
```

```
        }
};

int main() {
    Cliente pablo;
    pablo.nombre = "Pablo Cesar";
    pablo.dinero = 12.1;
    Cliente jose;
    jose.nombre = "Jose Miguel";
    jose.dinero = 135.23;
    jose.imprimeSaldo();
    pablo.imprimeSaldo();
    pablo.deposita(300);
    pablo.imprimeSaldo();
}
```

Podemos ver que ambos clientes tienen las mismas funciones pero cada funcion es especifico a cada cliente, una llamada a imprimeSaldo solo imprime el saldo de ese cliente y no de todos.

La razón por la que existen las variables privadas es para asegurar que esta variable solo este modificada correctamente. Si fueramos a llamar a deposita con un número negativo se perdería dinero asi que es recomendable hacer unas pruebas dentro de esta función, hacer la variable dinero privada y crear otras funciones que manejan esta variable.

Este concepto se llama la abstracción de datos debido a que estamos protegiendolos de uso equivocado o indeseado. Si intentamos leer o modificar la variable dinero desde main se nos arrojará un error, mientras que en cualquiera de las funciones de nuestra clase no habrá problema. Nuestro código quedará de la siguiente manera:

Listing 4: Abstracción de datos

```
#include <iostream>

using namespace std;

class Cliente {
    private:
        float dinero;

    public:
        string nombre;

        void imprimeSaldo() {
            cout << nombre << " tiene $" << dinero << endl;
        }
```

```
        void deposita(float cantidad) {
            if(cantidad <= 0) {
                cout << "Deposito invalido" << endl;
            } else {
                dinero += cantidad;
            }
        }
};

int main() {
    Cliente pablo;
    pablo.nombre = "Pablo Cesar";
    pablo.deposita(12.1);
    Cliente jose;
    jose.nombre = "Jose Miguel";
    jose.deposita(135.23);
    jose.imprimeSaldo();
    pablo.imprimeSaldo();
    pablo.deposita(-300);
    pablo.imprimeSaldo();
}
```

1.3 Constructores y destructores

Si vemos el ejemplo anterior ahora hemos protegido nuestro dinero de movimientos indeseados pero no podemos darle un valor inicial a nuestro dinero sin llamar a deposita, y puede ser que no queremos usar esa función para inicializar nuestro dinero.

Existe una función especial llamada el constructor que puede recibir varios parametros y que puede ser llamada cuando se instancializa un nuevo objeto.

El constructor siempre tiene el mismo nombre que el nombre de la clase y es una función sin tipo. En esta función es recomendable inicializar todas las variables de una clase:

Listing 5: Constructores

```
#include <iostream>

using namespace std;

class Cliente {
    private:
        float dinero;

    public:
        string nombre;
```

```
        //constructor
        Cliente(string nombreInicial, float saldoInicial) {
            nombre = nombreInicial;
            if(saldoInicial < 0) {
                dinero = 0;
            } else {
                dinero = saldoInicial;
            }
        }

        void imprimeSaldo() {
            cout << nombre << " tiene $" << dinero << endl;
        }

        void deposita(float cantidad) {
            if(cantidad <= 0) {
                cout << "Deposito invalido" << endl;
            } else {
                dinero += cantidad;
            }
        }
};

int main() {
    Cliente pablo("Pablo Cesar", -22.37);
    Cliente jose("Jose Miguel", 135.23);
    jose.imprimeSaldo();
    pablo.imprimeSaldo();
    pablo.deposita(12.1);
    pablo.imprimeSaldo();
}
```

Podemos ver que también nos ayuda a simplificar el código que escribimos dentro de la función main.

Tambien existen los destructores que son funciones especiales que se corren a la hora de borrarse una clase. Una clase solo se tiende a borrar al final de su ejecución, pero más adelante veremos otros casos donde se pueden borrar manualmente.

Para declarar el destructor se debe escribir una función sin parametros y sin tipos con un tilde seguido por el nombre de la clase.

Listing 6: Destructores

```
#include <iostream>

using namespace std;
```

```
class Cliente {
    private:
        float dinero;

    public:
        string nombre;

        //constructor
        Cliente(string nombreInicial, float saldoInicial) {
            nombre = nombreInicial;
            if(saldoInicial < 0) {
                dinero = 0;
            } else {
                dinero = saldoInicial;
            }
            cout << "Cliente " << nombre << " creado" << endl;
            imprimeSaldo();
        }

        //destructor
        ~Cliente() {
            cout << "Cliente " << nombre << " eliminado" << endl;
        }

        void imprimeSaldo() {
            cout << nombre << " tiene $" << dinero << endl;
        }

        void deposita(float cantidad) {
            if(cantidad <= 0) {
                cout << "Deposito invalido" << endl;
            } else {
                dinero += cantidad;
                cout << nombre << " ha recibido " << cantidad << endl;
            }
        }
};

int main() {
    for(int i = 0; i < 3; i++) {
        string nombre;
        float saldo;
        cout << "Nombre del cliente: " << endl;
        cin >> nombre;
        cout << "Saldo inicial: " << endl;
```

```
        cin >> saldo;
        Cliente temporal(nombre, saldo);
        float deposito;
        cout << "Dinero a depositar: " << endl;
        cin >> deposito;
        temporal.deposita(deposito);
        temporal.imprimeSaldo();
        cout << endl;
    }
}
```

Liga al código

1.4 Sobrecargamiento

En tanto las funciones regulares como las funciones de clases se puede llevar a cabo el sobrecargamiento. Esto consiste en tener varias funciones con el mismo nombre pero con diferentes tipos de parámetros. Por ejemplo, si quisieramos tener una función suma que es capaz de sumar dos números o dos dígitos en forma de cáracteres ('5' + '6' nos daría 11), entonces podemos hacer dos funciones que se sobrecargan:

Listing 7: Sobrecargamiento

```
#include <iostream>

using namespace std;

int suma(int a, int b) {
    return a + b;
}

int suma(char a, char b) {
    return (a - '0') + (b - '0');
}

int main() {
    cout << suma(5, 6) << endl;
    cout << suma('5', '6') << endl;
    cout << suma('9', '7') << endl;
}
```

Aqui podemos ver que la función suma es sobrecargada y que la primera vez que se llama utiliza la función de arriba porque sabe que los parámetros son dos enteros, mientras que las otras dos veces corre la función de abajo porque los parámetros son cáracteres.

El sobrecargamiento tambien permite la llamada de la misma función, asi que se puede crear una función general que es llamada por las funciones que sobrecargan:

Listing 8: Sobrecargamiento

```
#include <iostream>

using namespace std;

int potencia(int a, int b) {
    int resultado = 1;
    for(int i = 0; i < b; i++) {
        resultado *= a;
    }
    return resultado;
}

int potencia(char a, char b) {
    return potencia(a - '0', b - '0');
}

int main() {
    cout << potencia(5, 6) << endl;
    cout << potencia('5', '6') << endl;
    cout << potencia('9', '7') << endl;
}
```

Para una clase el sobrecargamiento es exactamente igual:

Listing 9: Sobrecargamiento de clases

```
#include <iostream>

using namespace std;

class Cliente {
    public:
        string nombre;
        float dinero;

        Cliente(string nombreInicial) {
            nombre = nombreInicial;
            dinero = 0;
        }

        Cliente(string nombreInicial, float dineroInicial) {
            nombre = nombreInicial;
```

```
            dinero = dineroInicial;
        }
};

int main() {
    Cliente pablo("Pablo");
    Cliente jorge("Jorge", 121.35);
}
```

1.5 Sobrecargamiento de operadores

Se pueden sobrecargar los operadores principales (+, -, =, ...) para que puedan manipular las instancias de dos clases. Digamos que tenemos una clase que sirve como un punto bidimensional y quisieramos que se sumarán dos puntos con el operador +, entonces se puede sobrecargar este operador en nuestra clase de la siguiente manera:

Listing 10: Sobrecargamiento de operadores

```
#include <iostream>

using namespace std;

class Punto {
    public:
        int x;
        int y;

        Punto(int xInicial, int yInicial) {
            x = xInicial;
            y = yInicial;
        }

        void imprimePunto() {
            cout << "(" << x << ", " << y << ")" << endl;
        }

        Punto operator+(const Punto& p) {
            Punto suma(x + p.x, y + p.y);
            return suma;
        }
};

int main() {
    Punto a(6, 11);
    Punto b(-3, 25);
```

```
    Punto c = a + b;
    c.imprimePunto();
}
```

Se debe crear una función con nombre operator**(signo)** y este debe regresar o modificar la clase que se esta utilizando, como parámetro debe recibir un **const tipo& nombre** donde esta variable es la otra variable que se esta sumando, restando o modificando.

1.6 Clases heredadas

Suele suceder que ocupas crear varias clases especificas con diferencias pequeñas. Para evitar tener que copiar el mismo código muchas veces se puede crear una clase general que contiene variables y funciones que se comparten entre otras clases especificas que "heredan" esta clase general.

Para heredar una clase se debe escribir dos puntos, public y el nombre de la clase general despues de la clase que hereda estas propiedades. Para llamar el constructor de una clase heredada se debe hacer algo parecido donde se pone dos puntos, el nombre del constructor de la clase que hereda y las variables que se desean pasar a ese constructor original.

Aqui podemos ver un ejemplo de un código donde se tiene dos clases, la clase Miembro contiene todas las funciones y variables de la clase Cliente, pero le hacemos unas modificaciones para que solo miembros reciben ciertas ventajas. Por ejemplo, no se puede modificar la variable puntos de un cliente porque esta clase no la tiene, pero la variable nombre si es igual para las dos clases.

Listing 11: Heredación

```
#include <iostream>

using namespace std;

class Cliente {
        public:
    string nombre;
    double dinero;

    Cliente(string nombreInicial) {
        nombre = nombreInicial;
        dinero = 0;
    }

    void venderArticulo(double precio) {
        dinero += precio;
        cout << nombre << " no recibio puntos
        por vender un articulo porque no es
        miembro" << endl;
```

```
    }

    void verDatos() {
        cout << nombre << " tiene $" << dinero << endl;
    }
};

class Miembro : public Cliente {
        private:
    double puntos;

        public:
    Miembro(string nombreInicial) : Cliente(nombreInicial) {
        puntos = 0;
    }

    void venderArticulo(double precio) {
        double puntosDeVenta = precio / 10;
        puntos += puntosDeVenta;
        dinero += precio;
        cout << nombre << " obtuvo " << puntosDeVenta <<
        " puntos por vender un articulo porque es miembro" << endl;
    }

    void verDatos() {
        cout << nombre << " tiene $" << dinero <<
        " y " << puntos << " puntos" << endl;
    }
};

int main() {
        Cliente pablo("Pablo");
        Miembro jose("Jose");
        pablo.venderArticulo(653.12);
        jose.venderArticulo(653.12);
        pablo.verDatos();
        jose.verDatos();
}
```

Si observamos este código podemos ver que Miembro es lo mismo que Cliente pero con otra variable y dos funciones sobrecargadas.

1.7 Variables estáticas

Las variables estáticas son especiales debido a que solo existe una instancia de ellas a la vez. Si declaramos una clase con una variable estática, esta variable

siempre se inicializará en cero y sera igual para todas las clases.

Para crear una variable estática se debe escribir **static** antes del nombre de la variable y se debe declarar esta variable fuera de la clase usando **Clase::nombre**;

Listing 12: Variables estáticas

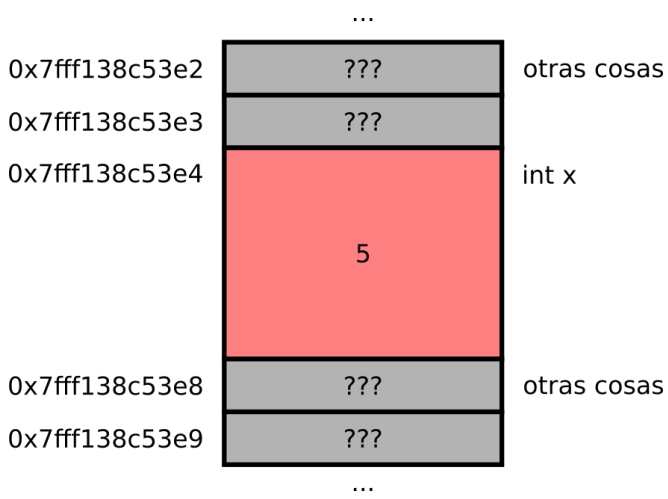
```
#include <iostream>

using namespace std;

class Punto {
    public:
        static int puntosMarcados;
        int x;
        int y;

        Punto(int xInicial, int yInicial) {
            x = xInicial;
            y = yInicial;
            puntosMarcados++;
        }
};

int Punto::puntosMarcados;

int main() {
    Punto a(3, 5);
    Punto b(5, 9);
    Punto c(2, 2);
    Punto d(1, 4);
    cout << "Hay " << Punto::puntosMarcados << " puntos" << endl;
}
```

Si corremos este ejemplo se imprimirán que hay 4 puntos porque el constructor de cada punto le sumó 1 a la variable puntosMarcados.

2 Referencias y punteros

Cada variable que se utiliza en C++ tiene un lugar o dirección en la memoria de la computadora donde se corre y estos lugares tienden a ser manejados en hexadecimal con cada lugar representando un byte de espacio. Si decidimos crear un entero este ocupará 4 bytes de espacio y ocupará una dirección que no conocemos. Por ejemplo, el entero podria ocupar las direcciones 0x7fff138c53e4, 0x7fff138c53e5, 0x7fff138c53e6 y 0x7fff138c53e7.

Abajo se muestra una figura donde se puede ver la variable señalada en rojo en la memoria ocupando 4 bytes de espacio.



Para ver la dirección de memoria de cierta variable se le debe colocar un signo de & antes de esa variable. Esto se llama la referencia de una variable y en el siguiente ejemplo desplegamos la referencia de un entero:

Listing 13: Referencias

```
#include <iostream>

using namespace std;

int main() {
        int x = 5;
        cout << &x << endl;
}
```

2.1 Punteros

Los punteros son variables especiales que guardan estas direcciones o referencias para diversas aplicaciones. Es recomendable indicarle a un puntero el tipo de variable que se encuentra en la posicion de memoria que guardará.

Para declarar un puntero se debe escribir el tipo de variable que se referenciará, una estrella y el nombre de ese puntero: **tipo∗ nombre;**

Los punteros son utiles porque te permiten modificar el contenido de la variable en ese lugar de memoria sin tener a la variable original. Para hacer esta modificación se debe de "dereferenciar" el puntero. Para dereferenciar algún puntero se debe colocar una estrella antes del nombre del puntero: **∗puntero**

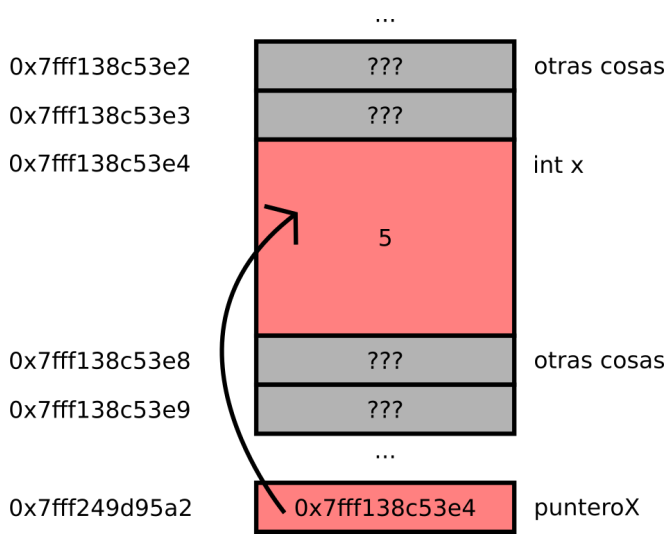
En el siguiente ejemplo se crea una variable x y se modifica su contenido para que sea 20 en lugar de 5:

Listing 14: Referencias

```
#include <iostream>
```

```
using namespace std;

int main() {
    int x = 5;

    //guarda la direccion de x
    int* punteroX = &x;

    //modifica el contenido de la direccion en el puntero
    *punteroX *= 4;

    cout << x << endl;
}
```

Como se puede ver en ningún momento modificamos x directamente. La siguiente figura muestra como el puntero guarda la dirección del entero x:



Una aplicación de un puntero es permitir que las funciones modifiquen las variables que se pasen como parámetros ya que normalmente esto no será posible porque los contenidos de las variables se copian. Por ejemplo se puede crear una función que intercambia los valores de sus dos parámetros:

Listing 15: Funciones con punteros

```
#include <iostream>

using namespace std;

void intercambiar(int x, int y) {
```

```
    int temp = x;
    x = y;
    y = temp;
}

void intercambiarConPunteros(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 5;
    int b = 7;
    intercambiar(a, b);
    cout << a << " " << b << endl;
    intercambiarConPunteros(&a, &b);
    cout << a << " " << b << endl;
}
```

Si corremos el código podemos ver que la función intercambiar no modifico las variables originales pero la función intercambiarConPunteros sí lo logró hacer.

2.2 Referencias

Otra manera de modificar los parámetros originales de una función es declarando los parámetros como referencias. Esto se puede hacer escribiendo un & en lugar de un * despues del tipo de dato. Esto automaticamente le indica a la función que quieres que ese variable sea la original y no una copia.

Listing 16: Funciones con referencias

```
#include <iostream>

using namespace std;

void intercambiarConReferencias(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5;
    int b = 7;
    intercambiarConReferencias(a, b);
```

```
    cout << a << " " << b << endl;
}
```

2.3 Punteros a objetos

Cuando se crea un puntero a un objeto como un struct o una clase se pueden acesar los miembros o variables de esas estructuras utilizando los cáracteres −> en lugar de un punto.

Digamos que tenemos un puntero a un struct que representa una comida y quisieramos guardar esta comida en un puntero. Si quisieramos ver el nombre de esta comida (un string), entonces en lugar de dereferenciar el objeto utilizamos la notación flecha:

Listing 17: Punteros a objetos

```
#include <iostream>

using namespace std;

struct Comida {
    string nombre;
    int calorias;
};

int main() {
    Comida pizza;
    pizza.nombre = "Pizza";
    pizza.calorias = 1184;
    Comida* punteroAComida;
    punteroAComida = &pizza;
    cout << punteroAComida->nombre << " tiene " <<
    punteroAComida->calorias << " calorias" << endl;
}
```

Con esta misma notación podemos modificar el contenido de la estructura.

2.4 Puntero this

El puntero this es un puntero especial que existe dentro de las clases que tiene una referencia a esa misma clase. Este puntero es util porque permite la distinción entre las variables que estan dentro de una clase y otras variables externas con el mismo nombre.

Un ejemplo sería tener una clase Punto que contiene una variable X y Y. Si creamos un constructor que tiene parametros con los mismos nombres no podemos modificar las variables internas de la clase a menos de que se usará el puntero this:

Listing 18: Puntero this

```
#include <iostream>

using namespace std;

class Punto {
    public:
    int x;
    int y;

    Punto(int x, int y) {
        this->x = x;
        this->y = y;
    }

    void imprimePunto() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Punto a(6, 3);
    a.imprimePunto();
}
```

3 Asignación de memoria

Si quisieras mayor control sobre la memoria de tu programa puedes crear variables, objetos y arreglos que después podrán ser borrados para liberar espacio utilizando la asignación de memoria. Este método busca espacio en una area compartida de la memoria de la computadora llamada el heap. El heap tiende a tener muchisimo más espacio que la memoria estandar asignada a tu programa asi que la asignación de memoria te permite crear arreglos más grandes de lo normalmente permitido (arriba de un millon de variables).

Para asignar una memoria se debe crear un puntero el cual guardará la ubicación de la variable que será asignado. Para asignar este dato se debe llamar **new tipo**. El siguiente código muestra un ejemplo de como se asigna un entero, un float y un arreglo con un millón de valores:

Listing 19: Asignando variables

```
#include <iostream>

using namespace std;

int main() {
```

```
    int enteroNormal = 19;
    int* enteroAsignado = new int;
    *enteroAsignado = 5;
    float floatNormal = 13.45;
    float* floatAsignado = new float;
    *floatAsignado = 3.1415;
    int* arregloGrande = new int[1000000];
    arregloGrande[999999] = 1234;
    cout << enteroNormal << endl;
    cout << *enteroAsignado << endl;
    cout << floatNormal << endl;
    cout << *floatAsignado << endl;
    cout << arregloGrande[999999] << endl;
}
```

Para liberar memoria que ya se haya utilizado entonces se debe llamar a **delete variable**. Es conveniente utilizar delete para cuando se quisiera ahorrar memoria, como en el siguiente código:

Listing 20: Borrando datos

```
#include <iostream>

using namespace std;

int main() {
    int* arregloLineal = new int[1000];
    for(int i = 0; i < 1000; i++) {
        arregloLineal[i] = 10 * i - 1;
    }
    cout << arregloLineal[3] << endl;
    cout << arregloLineal[999] << endl;
    cout << arregloLineal[800] << endl;
    delete arregloLineal;
    int* arregloCuadrada = new int[1000];
    for(int i = 0; i < 1000; i++) {
        arregloCuadrada[i] = i * i;
    }
    cout << arregloCuadrada[800] << endl;
}
```

Este código ahorra la mitad del espacio creando dos arreglos de tamaño 1000 y borrando uno despues de que haya sido utilizado.