# INFORME DE PROGRAMACION: PARCIAL N°1

GRUPO 14:

- Vargas, Rodrigo Sebastián
- Saillen, Simón

MODALIDAD: Virtual

# ¿Cómo empezó la idea principal?

Empezamos con la idea de crear una clase *class Fruta*, cuyos elementos privados sean el tipo de fruta y la cantidad de esta, luego en la *class Cajon* la idea era ingresar un elemento privado tipo Fruta,

```
class Frutas{
    private:
        string tipo;
        long int cantidad;

    public:
        void setTipo();
        string getTipo();
        void setCantidad();
        long int getCantidad();

        void Almacenar();
        void Quitar();
        void Seccionar();
};
```

```
class Cajon {    //es la clase principal
    private:
    Frutas fruta;
    int ID;
    int pesoActual;
    bool Estado;   //True si está lleno y false si no
    int const cantMAX = 20;

    public:
    Cajon(){}; //Constructor
    ~Cajon();

};
```

Imágenes 1 y 2: “Primeros bocetos de las clases Cajon y Frutas”

Luego de pruebas e intentos, terminamos por descartar la clase Fruta y tomar la clase Cajon con un elemento tipo *string fruta* y con su respectivo peso, con sus métodos *setters* y *getters* necesarios para que funcione el mismo. También definimos una clase llamada class Secciones, que es la encargada de tener, controlar y agregar o quitar cajones y/o fruta dependiendo de su tipo. (En nuestro parcial nos centramos en 3 tipos de fruta: “Dulce”, “Acida” y “Neutra”).

```
class Cajon {
private:
  int id;
  string tipo_fruta;
  int cantidad;

public:
  Cajon(int id, string tipoF, int cant) {
    this->id = id;
    tipo_fruta = tipoF;
    cantidad = 0;
  }
  int getId() { return id; };
  void setId(int s) { id = s; };
  string getTipoFruta() { return tipo_fruta; };
  int getCantidad() { return cantidad; };
  void setCantidad(int c) { cantidad = c; };
  bool estaCompleto() { return cantidad == 20; };
  bool estavacio() { return cantidad == 0; };
};
```

```
class Seccion {
private:
  Pila<Cajon *> cajonesApilados;
  string tipo_fruta;
  int cantidadCajones;
  int cantidadFruta;

public:
  Seccion(string t) {
    this->tipo_fruta = t;
    this->cantidadCajones = 0;
    this->cantidadFruta = 0;
  }
  void agregarFruta(int c);
  void sacarFruta(int c);
  void imprimirPila();
  string getTipo() { return tipo_fruta; };
  int getCantCajones() { return cantidadCajones; };
  Pila<Cajon *> getPilaCajones() { return this->cajonesApilados;};
  void setCantFruta(int ct) { cantidadFruta = ct; }
  int getCantFruta();
};
```

Imágenes 3 y 4: “Clases Cajon y Sección”

# ¿Entonces como quedó la estructura actual?

Después de crear las clases que nos resultaron convenientes y/o funcionales, terminamos con una estructura similar a la imagen de abajo:

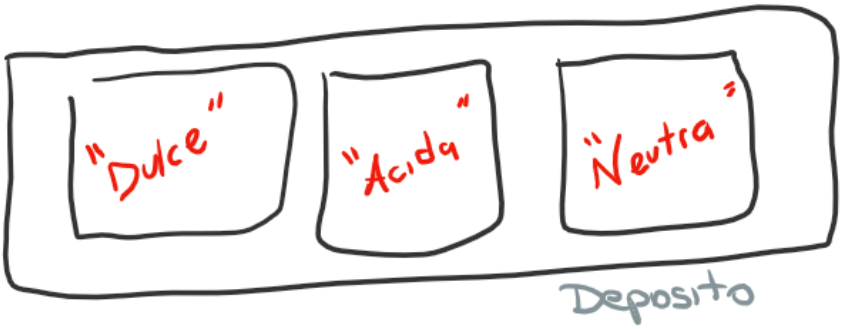


Imagen 5: “Ilustración del concepto detrás de las clases Deposito y Secciones”

Donde el Depósito contiene un arreglo de 3 elementos de tipo Sección (uno para cada tipo de fruta), y Secciones contiene la Pila de Cajones de su tipo propio. Al momento de realizar las Pilas para cada tipo de fruta nos encontramos con una dificultad, la idea principal era realizar un “Arreglo de Pilas” para poder almacenar al menos 10 pilas de cada tipo en su respectivo sector, pero desafortunadamente cada intento de agregar fruta o quitarla en distintas pilas dentro del arreglo generaba errores que no pudimos corregir, uno de los intentos es el de la siguiente imagen:

```
class Deposito { // Deposito == Stock
private:
  Seccion *sectores[3];
  int cant_dulce;
  int cant_acida;
  int cant_neutra;

public:
  Deposito() {
    sectores[0] = NULL;
    sectores[1] = NULL;
    sectores[2] = NULL;
    this->cant_dulce = 0;
    this->cant_acida = 0;
    this->cant_neutra = 0;
  };
  void agregarSeccion(Seccion *c);
  Seccion *getSeccion(int);
  int getCantDulce();
  int getCantAcida();
  int getCantNeutra();
  void setCantDulce(int d) { cant_dulce = d; }
  void setCantAcida(int a) { cant_acida = a; }
  void setCantNeutra(int n) { cant_neutra = n; }
};
```

```
void Seccion::agregarFruta(int c) {
  int Max = (pilaId+1)*200;
  if (this->getCantFruta() + c <= Max) {
    if (cajonesApilados[pilaId].pilavacia() || cajonesApilados[pilaId].tope()->estaCompleto()) {
      // si la pila está vacía o el último cajón está completo, agrega un nuevo
      // cajón a la pila
      if (cantidadCajones < 30) {
        Cajon *nuevoCajon = new Cajon(cantidadCajones + 1, tipo_fruta, 0);
        cajonesApilados[pilaId].apilar(nuevoCajon);
        cantidadCajones++;
      }
    }
    // agrega la fruta al cajón actual
    int cantidadFaltante = 20 - cajonesApilados[pilaId].tope()->getCantidad();
    if (c > cantidadFaltante) {
      cajonesApilados[pilaId].tope()->setCantidad(20);
      agregarFruta(c - cantidadFaltante);
    } else {
      cajonesApilados[pilaId].tope()->setCantidad(
          cajonesApilados[pilaId].tope()->getCantidad() + c);
      cantidadFruta += c;
    }
  } else {
    int aux = cajonesApilados[pilaId].tope()->getCantidad();
    cajonesApilados[pilaId].tope()->setCantidad(20);
    pilaId++;
    this->agregarFruta(c-(20-aux));
  }
}
```

Imagen 6 y 7: “Clase Deposito / Intento de agregar fruta en el arreglo de pilas”

```
void Seccion::sacarFruta(int cant) {
  if (cajonesApilados.pilavacia()) {

    this->setCantFruta(0);
    cout << "\nNO HAY TANTA FRUTA PARA EL PEDIDO\n";
  } else {

    if (cant >= 20) {
        int cantCj = this->cajonesApilados.tope()->getCantidad();
        this->cajonesApilados.desapilar();
        this->cantidadCajones--;
        sacarFruta(cant - cantCj);
    } else {
      if (cant < this->cajonesApilados.tope()->getCantidad()) {
        int cantCj = this->cajonesApilados.tope()->getCantidad();
        this->cajonesApilados.tope()->setCantidad(cantCj - cant);
      } else {
        int cantCj = this->cajonesApilados.tope()->getCantidad();
        this->cajonesApilados.desapilar();
        this->cantidadCajones--;
        sacarFruta(cant - cantCj);
      }
    }
  }
}
```

Imagen 8: “La versión final del método *sacarFruta()* (para 1 pila)”

```
void Seccion::agregarFruta(int aux) {
  int c;
  (aux > 200) ? c = 200 : c = aux;

  if (this->getCantFruta() + c <= 200) {
    if (cajonesApilados.pilavacia() || cajonesApilados.tope()->estaCompleto()) {
      // si la pila está vacía o el último cajón está completo, agrega un nuevo
      // cajón a la pila
      if (cantidadCajones < 10) {
        Cajon *nuevoCajon = new Cajon(cantidadCajones + 1, tipo_fruta, 0);
        cajonesApilados.apilar(nuevoCajon);
        cantidadCajones++;
      } else {
        cout << "\nNo hay mas cajones en la pila"<< endl;
      }

    }
    // agrega la fruta al cajón actual
    int cantidadFaltante = 20 - cajonesApilados.tope()->getCantidad();
    if (c > cantidadFaltante) {
      cajonesApilados.tope()->setCantidad(20);
      cantidadFruta += cantidadFaltante;
      agregarFruta(c - cantidadFaltante);
    } else {
      cajonesApilados.tope()->setCantidad(cajonesApilados.tope()->getCantidad() + c);
      cantidadFruta += c;
    }
  } else if (this->getCantFruta() + c > 200){
    cajonesApilados.tope()->setCantidad(20);
  }
}
```

Imagen 9: “La versión final del método *agregarFruta()* (para 1 pila)”

Por lo que tuvimos que limitar la cantidad de fruta que se puede agregar por sección a 200 (10 cajones llenos).

# ¿Alguna otra dificultad que encontramos?

Si, en los métodos de impresión, *imprimirPila()* e *imprimirPedidos()*, el segundo pertenece a la clase GestionPedidos (mas adelante hablaremos de ella). La problemática era que, al imprimir, en el primer caso, la pila de cajones junto a su cantidad y su ID hacia falta desencolar para mostrar el siguiente, por lo que se nos ocurrió que, a medida que desencolamos la encolamos en una pila auxiliar, el tema es que como las pilas son tipo LIFO, inconscientemente estábamos dando vuelta la pila, pero lo solucionamos con darla vuelta de nuevo del mismo modo mediante un *while*, algo parecido ocurrió en imprimirPedidos() pero a la hora de crear la cola auxiliar, no ocurrió algo como el caso anterior porque las colas son de tipo FIFO.

```
void Seccion::imprimirPila() {

  Pila<Cajon *> pilaAux;
  while (!this->cajonesApilados.pilavacia()) {
    Cajon* aux = cajonesApilados.tope();
    cout << "CAJON ID: " << cajonesApilados.tope()->getId()
         << ", FRUTA TOTAL (EN KG): " << cajonesApilados.tope()->getCantidad() << endl;
    pilaAux.apilar(aux);
    this->cajonesApilados.desapilar();
  }

  if (pilaAux.pilavacia()){
    cout << "\nPila de cajones apilados quedó vacia" << endl;
  }

  while(!pilaAux.pilavacia()) {
    cajonesApilados.apilar(pilaAux.tope());
    pilaAux.desapilar();
  }

}
```

Imagen 10: “La versión final de imprimirPila()”

```
void GestionPedidos::imprimirPedidos() {
  Cola <Pedidos*> aux;
  while (!pedidosMinorista.colavacia()) {
    string nombreCliente = pedidosMinorista.tope()->getNombreCliente();
    string tipoCliente = pedidosMinorista.tope()->getTipoCliente();
    int cantidad_fruta = pedidosMinorista.tope()->getCantidadPedido();
    string tipo_fruta = pedidosMinorista.tope()->getTipoFruta();
    cout << "\nNOMBRE CLIENTE: " << nombreCliente
         << "\nTIPO CLIENTE: " << tipoCliente << "\nCANTIDAD " << tipoCliente
         << " DE FRUTA "<< tipo_fruta<< ": " << cantidad_fruta << endl;
    aux.encolar(pedidosMinorista.tope());
    pedidosMinorista.desencolar();
  }
    pedidosMinorista = aux;

  Cola<Pedidos*> aux2;

  while (!pedidosMayorista.colavacia()) {
    string nombreCliente = pedidosMayorista.tope()->getNombreCliente();
    string tipoCliente = pedidosMayorista.tope()->getTipoCliente();
    int cantidad_fruta = pedidosMayorista.tope()->getCantidadPedido();
    string tipo_fruta = pedidosMayorista.tope()->getTipoFruta();
    cout << "\nNOMBRE CLIENTE: " << nombreCliente
         << "\nTIPO CLIENTE: " << tipoCliente << "\nCANTIDAD " << tipoCliente
         << " DE FRUTA " << tipo_fruta << ": "<< cantidad_fruta << endl;
    aux2.encolar(pedidosMayorista.tope());
    pedidosMayorista.desencolar();
   }
  pedidosMayorista = aux2;
}
```

Imagen 11: “La versión final de imprimirPedidos”

# ¿Clase Pedidos y GestionPedidos?

Similar al caso de la clase Cajon y la clase Sección, la clase Pedidos viene a ser una definición que será utilizada por GestionPedidos, esta clase tiene de elementos privados 4 colas principales de tipo Pedidos*, pedidos Normales (Mayorista y Minorista) y pedidos Pendientes (Mayorista y Minorista), un parámetro para la prioridad a elección y un elemento tipo Deposito* llamado stockDisponible, este será usado mas adelante para comprobar si es posible o no la entrega del pedido solicitado, llevándolo a la cola de pendientes en caso contrario. Los pedidos deben ser ingresados de manera manual en main mediante el método registrarPedido() el cual recibe de parámetros el nombre del cliente, el tipo de cliente, el tipo de fruta y la cantidad del pedido, agregando el pedido a la cola correspondiente al tipo de cliente.

```
void GestionPedidos::registrarPedido(string nombre, string tipo, string tipoF,
                                     int cant) {

  Pedidos *pedidoAux = new Pedidos(nombre, tipo, tipoF, cant);

  if(tipo == "Minorista"){
    pedidosMinorista.encolar(pedidoAux);
  } else {
    pedidosMayorista.encolar(pedidoAux);
  }
}
```

Imagen 12: “La versión final del método registrarPedido()”

```
void GestionPedidos::ventaPedido(Pedidos *pedido) {
  int cantPedido;
  string tipo = pedido->getTipoFruta();
  string tipoC = pedido->getTipoCliente();

  if (tipoC == "Minorista") {
    cantPedido = pedido->getCantidadPedido();
  } else {
    cantPedido = (pedido->getCantidadPedido()) * 20;
  }

  if (tipo == "Dulce") {
    this->stockDisponible->getSeccion(0)->sacarFruta(cantPedido);
  } else if (tipo == "Acida") {
    this->stockDisponible->getSeccion(1)->sacarFruta(cantPedido);
  } else if (tipo == "Neutra") {
    this->stockDisponible->getSeccion(2)->sacarFruta(cantPedido);
  }
}
```

Imagen 13: “La versión final de ventaPedido()”

El método más importante es el encargado de manejar la entrega de los pedidos en el orden que se quiera, para eso se utiliza el parámetro que recibe la función, si recibe un 0, entonces tomara la prioridad Minorista > Mayorista, primero verificando si hay pedidos pendientes a entregar.

```
void GestionPedidos ::entregarPedido(int n) {
  if (n == 0) { // prioridad minorista>mayorista
    entregarPedidosPendientes(pedidosEsperaMin, 0);
    entregarPedidosPendientes(pedidosEsperaMay, 1);
    entregarPedidosNormales(pedidosMinorista, 0);
    entregarPedidosNormales(pedidosMayorista, 1);

  } else if (n == 1) {
    entregarPedidosPendientes(pedidosEsperaMay, 1);
    entregarPedidosPendientes(pedidosEsperaMin, 0);
    entregarPedidosNormales(pedidosMayorista, 1);
    entregarPedidosNormales(pedidosMinorista, 0);
  }
}
```

Imagen 14: “Método entregarPedido()”

El método utiliza 2 métodos extras usados para 2 tipos de pedidos, los normales y los pendientes.

```
void GestionPedidos ::entregarPedidosNormales(Cola<Pedidos *> aux, int n) {
  if (!aux.colavacia()) {
    if (esPosible(aux.tope())) {
      ventaPedido(aux.tope());
      aux.desencolar();
      entregarPedidosNormales(aux, n);
    } else {
      if (n == 0) {
        pedidosEsperaMin.encolar(aux.tope());
        aux.desencolar();
        entregarPedidosNormales(aux, n);
      } else {
        pedidosEsperaMay.encolar(aux.tope());
        aux.desencolar();
        entregarPedidosNormales(aux, n);
      }
    }
  }
}
```

Imagen 15: “Método entregarPedidosNormales()”

```
void GestionPedidos ::entregarPedidosPendientes(Cola<Pedidos *> aux) {
  if (!aux.colavacia()) {
    if (esPosible(aux.tope())) {
      ventaPedido(aux.tope());
      aux.desencolar();
      entregarPedidosPendientes(aux);
    }
  }
}
```

Imagen 16: “Método entregarPedidosPendientes()”

En extra hay unos cuantos metodos más de impresión, que al no ser tan importantes no los mostramos acá, pero en el código están, estos son stockActual(), impreEstado(), etc. Luego se encuentra el main, donde se inicializan los objetos, se agrega fruta y se hacen los pedidos.

```cpp
int main() {

  Deposito *deposito = new Deposito();
  GestionPedidos *pedidos = new GestionPedidos(deposito);
  Seccion *seccion_dulce = new Seccion("Dulce");
  Seccion *seccion_acida = new Seccion("Acida");
  Seccion *seccion_neutra = new Seccion("Neutra");

  deposito->agregarSeccion(seccion_dulce);
  deposito->agregarSeccion(seccion_acida);
  deposito->agregarSeccion(seccion_neutra);

  seccion_dulce->agregarFruta(80);
  seccion_acida->agregarFruta(100);
  seccion_neutra->agregarFruta(120);
  pedidos->stockActual();
  pedidos->registrarPedido("Rodrigo", "Minorista", "Acida", 50);
  pedidos->registrarPedido("Emiliano", "Mayorista", "Dulce", 2);
  pedidos->registrarPedido("Jose", "Mayorista", "Acida", 3);
  pedidos->registrarPedido("Simon", "Minorista", "Dulce", 60);
  pedidos->registrarPedido("Anibal", "Minorista", "Neutra", 80);

  pedidos->entregarPedido(0);
  pedidos->stockActual();
  pedidos->ImpreEstado();
  seccion_dulce->agregarFruta(30);
  seccion_acida->agregarFruta(20);
  pedidos->entregarPedido(0);
  pedidos->ImpreEstado();
  pedidos->stockActual();
  return 0;
```

```
-----------------------------------------
CANTIDAD DE FRUTA DULCE (EN KG): 80
CANTIDAD DE FRUTA ACIDA (EN KG): 100
CANTIDAD DE FRUTA NEUTRA (EN KG): 120
-----------------------------------------


-----------------------------------------
CANTIDAD DE FRUTA DULCE (EN KG): 20
CANTIDAD DE FRUTA ACIDA (EN KG): 50
CANTIDAD DE FRUTA NEUTRA (EN KG): 40
-----------------------------------------

PEDIDOS EN ESPERA:
Hay 2 pedidos en espera tipo Mayorista

No hay pedidos en espera de Tipo Minorista

No hay pedidos en espera de Tipo Mayorista

No hay pedidos en espera de Tipo Minorista

-----------------------------------------
CANTIDAD DE FRUTA DULCE (EN KG): 10
CANTIDAD DE FRUTA ACIDA (EN KG): 10
CANTIDAD DE FRUTA NEUTRA (EN KG): 40
-----------------------------------------
```

Imágenes 17 y 18: “Main y ejemplo de impresión”