04/05/23

vlermanda2018@udec.cl

Objetivo

- Repasar punteros y memoria dinámica.
- Manejar estructuras basadas en nodos.
- Resolver ejercicios de programación competitiva aplicando estructuras de datos básicas.

```
![[Pasted image 20230504022402.png|400]]
```

Variables y tipos de datos

- "*Contenedores*" que guardan información (bytes).
- Al declarar una *variable*, se le asigna:
  1. Un nombre único para que *nosotros* la podamos *llamar*.
  2. Una dirección de memoria única para que el *computador* sepa como acceder a ella cuando sea *llamada*.
  3. Un tipo de dato, que nos indica el *dominio* donde pertenece la variable.
- Cada tipo de dato tiene cierta cantidad de bytes, y operaciones definidas para ese tipo.

```
int entero = 2;
char carac = 'b';
//A   //B   //C
```

- **A**: Tipo de dato (int: 4 bytes, char: 1 byte).
- **B**: Nombre de variable (además, el computador le asigna una dir. de memoria).
- **C**: Asignación de un valor correspondiente al dominio que representa el t.d.
- Lo más importante con lo que se tienen que quedar, es que una variable *no es* solo un valor. Si lo definimos en sus componentes sería:
  - Tipo de dato.
  - Nombre.
  - Dirección de memoria (&).
  - Valor.

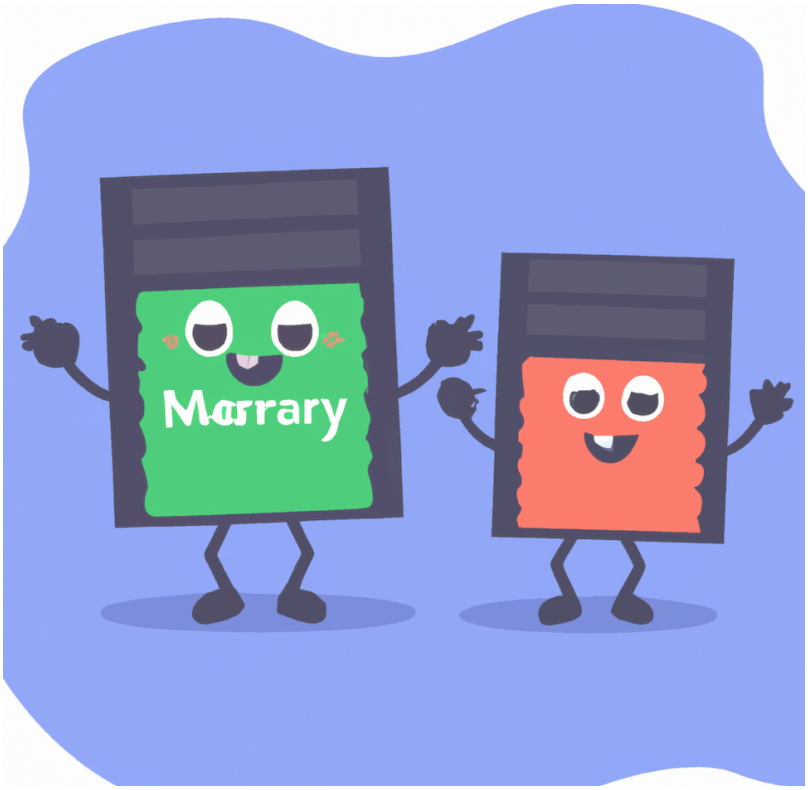
Punteros

- Corresponde a un tipo de dato, el cual siempre va *asociado* a otro t.d.
- Su función es *apuntar* la dirección de memoria de una variable.
- Sin embargo, podemos "observar" el valor que existe donde está apuntando.
- Siempre utiliza 8 bytes de memoria sin importar el tipo de dato al que esté asociado (la flechita igual pesa!).

- Ej: puntero *apuntando* la dirección de memoria de una variable.

```
double dec = 1.2;
double* d_p = &dec;
cout<<"Dir. memoria "<< d_p <<endl;
cout<<"Valor de la d.m apuntada: "<< *d_p <<endl;
```

Memoria dinámica

- Agregamos un nuevo componente! ya no tenemos una *única memoria*.
- Memoria en stack, y memoria en heap (no confundir con las EDs del mismo nombre).
- 

- Al declarar un tipo de dato (incluso un t.d puntero) de la forma tradicional, lo creamos en el *stack*. ¿Qué implica esto? 🤔

    - Se le asigna memoria, es decir, la cantidad de bytes correspondiente, cuando se compila.
    - Esta memoria es *limpiada* automáticamente una vez termina la ejecución del programa.
- La *memoria dinámica* es creada en el *heap*.
    - Como su nombre indica, su memoria es asignada dinámicamente al momento de ejecutar el programa.
    - Esto nos entrega libertad para crear, eliminar, y reasignar memoria en nuestros programas.
    - Su utilidad más directa es la de poder almacenar una cantidad de datos *no* especificada desde un comienzo.
    - Esta memoria debe ser liberada manualmente con $delete$.
- Para utilizar variables con memoria dinámica, se debe asociar esta a un puntero para que la podamos *llamar*.

```
int*  num = new int;
char* str = new char[n];
//A   //B     //C
delete num;
delete[] str;
```

- **A**: Tipo de dato puntero int y puntero char.
- **B**: Declaración de variables (asignación de memoria en el stack).
- **C**: Función para asignar memoria una vez el programa entre en ejecución.
- Recordar: *puntero != memoria dinámica* !!

Linked Lists

Nodo

- O en este contexto simplemente podemos llamarlo *elemento*.
- Corresponde a una estructura que puede tener distintos tipos de datos internamente.
- Es equivalente a una clase donde todos sus componentes son públicos.
- Guardan una referencia (un puntero) a otro nodo (dependiendo el contexto donde se use se puede llamar vecino, hermano anterior/siguiente, nodo anterior/siguiente, hijo/padre, etc).

```
struct nodo{
        int valor;
        nodo *nodo_siguiente;
        nodo(int valor){ this->valor = valor; }
};
```

Linked List

- Corresponde a una colección de nodos o elementos.
- Mantiene punteros a nodo especiales, los cuales comunmente indican el nodo inicial, y el final (head y tail).
- Si no guardara estos punteros, no habría forma de operar. ¿Cómo insertamos al comienzo / posición i / final ?

    - Obligatoriamente se debe comenzar a recorrer la lista desde el nodo apuntado por *head*.
- ¿Qué otros beneficios nos entrega mantener los punteros *head* y *tail*?
- ¿Qué complejidad tiene remover un nodo al principio y al final?
- ¿Cómo podríamos mejorar esto?
- Interfaz usual de una linked list:

```
class linked_list{
private:
        nodo* head;
        nodo* tail;
public:
        void push_front(int elemento);
        void push_back(int elemento);
        int pop_front();
        int pop_back();
        int size();
};

stack<int> st;
```

- Dependiendo de las necesidades, se agregan métodos que permitan recorrer la lista.

Deque

- Lista doblemente enlazada!

- Ahora cada nodo tiene 2 punteros, al nodo siguiente, y al anterior.
- Esto nos permite realizar todas las operaciones básicas de una linked list en tiempo constante.
- A cambio de un mayor tamaño de la estructura.
- Entonces, ¿Qué estructura es mejor usar?
    - Depende. Como regla general, "no queremos matar moscas a cañonazos".
    - Claramente no es el único factor. Si no tenemos limitantes en cuanto a memoria y procesamiento, si usar alguna opción nos ahorra tiempo al programar, es una buena idea decantarse por esa.

Ejercicios

- Juntarse en grupos para trabajar ejercicios de stacks.
- Se recomienda utilizar el stack de la STL, donde pueden encontrar todo lo referente a ella, incluido los métodos que tiene (que son varios más que solo los básicos), revisar la documentación https://cplusplus.com/reference/stack/stack/.

1. https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1012
2. https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=614

3. https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=455