General

- Documentá más el código. Ayuda al corrector a entender lo que hiciste, y cuando estés en un ambiente laboral, tus compañeros te van a odiar menos.

main

- La función `std::getline(std::basic_stream &file, std::string &)` te devuelve una referencia a `file`. Internamente y desde el estandar 11, esta clase tiene implementado el `operator bool()`. Este operador te devuelve `true` si todavía no se llegó al final del archivo, o false en caso contrario. El ciclo de parseo lo podrías haber hecho de la siguiente forma:

```
std::string key;
std::string value;
while (std::getline(config_file, key, DELIMITADOR) &&
       std::getline(config_file, value)){
    if (key == "cache type") {
        cache_type = value;
    }
    map_config_file.insert(std::pair<string, string>(key, value));
}
```

Observar la simplicidad de cargar el archivo de configuración. No está mal lo que hiciste, solo sugiero una alternativa más legible y con menos variables en el medio. Otra cosa que sugiero es usar `std::map::emplace`. Insert realiza una copia de los elementos, mientras que emplace puede realizar un movimiento, revisar la documentación al respecto de esta función para corroborar lo que digo.

- linea 62: La cadena de `if` que hay para determinar cual hijo de Cache debe instanciarse podrías tenerlo en el constructor de ProtectedCache. Le pasarías por referencia al constructor el `std::string cache_type` y el constructor creá dinámicamente el tipo de cache. De esta forma podrías lograr que esta parte sea RAII, ya que construirías el recurso en el constructor y lo destruirías en el destructor. Con la implementación que elegiste tenés que tener mucho cuidado de liberar los recursos cada vez que creás un cache (en este TP lo vas a crear una única vez, pero en próximos TPs te va a ser muy útil).

- Línea 76: El mapa lo estás copiando 2 veces ahí. Eso es muy feo. CacheProtected debería tener una referencia constante del mapa. No debe copiarse. Y para que te compile el programa deberías pasar la referencia del `map_config_file` en el constructor de CacheProtected. El constructor se vería de la siguiente forma:

```
CacheProtected(string cache_type, const map<string, string> &config){
   // la logica para crear el cache que te mencione anteriormente
   // ahora solo crearé  un cache directo
   cache = new Cache_Direct(config);
}
```

Internamente la clase CacheDirect debería estar definida de la siguiente forma:

```
class Cache_Direct: public Cache{
 public:
    Cache_Direct(const map<string,string> &config): Cache(config){
      //Logica extra si la tiene. La definición deberia estar en el cpp
  }
};
```

No generes copias innecesarias que pueden enlentecer tu programa. Revisar las diapositivas de clase sobre *move semantics* y *RAII*.

- linea 81: podrías haber usado `emplace_back` y no crear una clase en el heap, sinó simplemente moverías una clase creada en el stack.

```
threads.emplace_back(FunctorCache(cache_protected, argv[i]));
```

De esta forma no tendrías que hacer ningún delete, simplemente se destruirían cuando se destruya el vector contenedor.

- Linea 101: fstream es una clase que implementa la ideología RAII, no es necesario cerrarlo ya que su destructor se encarga de hacerlo.

cache

- ¿Era necesario recuperar todos los datos del mapa? ¿no podías simplemente guardar una referencia del mapa (sin copiarlo), y cuando te pidan algun dato del cache realizar un `this->config_map.find("vendor_id")` por ejemplo? Revisar que devuelve el método find para resolver los casos en los que no esté cargado el dato. Si no queres gastar tiempo buscando el elemento, está bien, tenelos en atributos distintos, pero no los copies, movelos.

- Otra cosa fea en el constructor es que estás copiando todos los atributos. Si no los va a usar nadie mas, movelos. Incluso mejor, move el mapa directamente y guardá esa referencia.
- No usar `atoi`. Esta función no chequea errores. En C usá strotol, en C++ podés crear un `std::istringstream` y utilizar las bondades del `operator>>`.
- Respetá el idioma, si estás codeando en inglés no metas un spanglish (`print_informe`)

Cache_Associative_Fifo

set_data

- Debería estar en el constructor para cumplir con RAII.
- No copies el mapa de addresses_in_cache_map.

```
// de esta forma lo estarías moviendo al mapa, creado en el stack
this->addresses_in_cache = map<string, bool>();
```

print_initialization_data

- No era necesario re implementar este método si no le vas a agregar ninguna funcionalidad extra. Estas creando un llamado en el medio, haciendo un backup en el stack para guardar el contexto, para llamar a un método que se iba a llamar solo si no definías esto. Como este método no le agregan nada ninguno de sus hijos, no lo hagas virtual.

procces_memory_address

- No copies todos los parámetros, esto no es Java. En C++ el pasaje por default es por copia. Agregá un `&` al principio del nombre de la variable para que sea una referencia, o doble ampersand si lo vas a mover. Leer las diapositivas de *move semantics*.

print_informe

- No redefinas un método que su única función es llamar al método del padre. Es ineficiente.

Cache_Associative_Lru

constructor

- Evitá copiar parámetros.
- Es peligroso lo que estás planteando con el mapa de iterators y la cola. Un iterator se puede pensar como un puntero. Cuando modifiques la cola deberías revisar todos los iterators porque los elementos pudieron ser movidos

  Un ejemplo puede ser el siguiente:

Tenés una pila llena, y surge un hit del último elemento de la pila. Al eliminar este elemento, todos los elementos pueden reordenarse. Como generalmente, el deque está implementado como un arreglo dinámico, al insertar o eliminar un elemento puede ocurrir una reallocación de los elementos, haciendo que tus iterator en el mapa sean obsoletos.

Vos podrías replicarme “¿pero una queue no está implementada con una lista doblemente enlazada, haciendo que el elemento sea inalterado en su lifetime?” y la respuesta te la da cplusplus (http://www.cplusplus.com/reference/deque/deque/):

deque (usually pronounced like “deck”) is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Specific libraries may implement deques in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes undefined behavior.

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists and forward lists.

FunctorCache

- Tecnicamente hablando, un functor es aquel que define el método `operator()`
- Podrías abrir el archivo en el constructor directamente.

run

- Acá volcaste, toda la función atómica es process_if_valid, donde internamente se valida que sea valido, y si no lo es printea el mensaje de error. Imaginate el caso en el que haya dos hilos que quieran ingresar el tag 0x000001. El primer hilo (h1), valida la memoria, y le devuelven `true`. Imaginate que justo que obtiene el result y libera el lock, el hilo de ejecución cambia. h2 pasa a tomar el control, valida que la memoria sea válida, y como el lock se encuentra liberado, y como h1 todavía no

lo insertó al tag, esta función le devuelve `true` nuevamente. Vas a insertar dos veces el mismo tag en estos casos.

- El enunciado dice claramente que el archivo de acceso a memoria es un archivo de **TEXTO**.