
PEL



Resumen UF2

U1

Paginación de memoria: Proceso de simplificación de gestión de las memorias. Permite alojar bloques contiguos de memoria virtual en bloques desperdigados de memoria física, y también direccionar múltiples procesos a las mismas páginas de memoria física. Por motivos de seguridad, los sistemas operativos usan la técnica *Address Space Layout Randomization* para introducir de forma aleatoria los datos clave de los procesos en las diferentes áreas de ubicación de la memoria física. ASLR es una técnica de seguridad informática que se emplea en la prevención de vulnerabilidades debidas a corrupción en la memoria.

-Páginas virtuales: Los espacios de memoria virtual pueden dividirse en “páginas virtuales”, que a su vez se dividen en marcos de páginas con igual tamaño. El sistema operativo transfiere las páginas del proceso a los marcos de páginas libres de memoria según sea necesario, siguiendo un control estricto para el control de la asignación. De esta forma múltiples procesos pueden ejecutarse de forma simultánea.

(User Stack) Pila del usuario: Espacio de la memoria virtual de un equipo destinado a las operaciones ejecutadas por el usuario. (A diferencia de la pila de kernel) no es directamente accesible y presentan ubicaciones diferentes en el registro de memoria. De forma general, el tamaño de la pila es de 8MB, y normalmente permanece estático a lo largo del cuerpo de la función. Sigue un esquema lifo (last in first out), y al reutilizar la pila constantemente la mantiene activa en memoria caché, de forma que el acceso a sus datos es enormemente eficiente.

-Arq. AMD64 -> registro puntero de pila **%rsp** referencia siempre a la cabeza de la pila del usuario. Si se quiere alojar o desalojar memoria, solamente debe modificarse el valor de dicho puntero, llamar a una función, generalmente incluye un nuevo marco en la pila del usuario.



-Errores de software comunes:

1. Si el uso de la memoria sobrepasa el tamaño permitido para la pila, se produce una violación de acceso, crasheando así el proceso. (Agotar los recursos al almacenar variables de gran tamaño o Ejecutar algoritmos recursivos infinitos o demasiado profundos).
2. Sobrescribir un array más allá de su cota superior, siendo este un error difícil de detectar en el debug. Esto se puede evitar empleando la clase plantilla array y bucles for basados en rango, en lugar de utilizar los arrays estáticos.

Alineación: Consiste en alojar variables de un tipo determinado en diferentes direcciones de memoria, siendo estas múltiplos de números enteros (estos suelen ser 1, 2, 4, 8 o 16). Siendo D la dirección en memoria de una variable, la alineación L de dicha variable se define en mayo potencia de dos tal que $D \bmod L = 0$.

	Tamaño en Bytes	Alineación en Bytes
char	1	1
short	2	2
int	4	4
long int	4	4
long long int	8	8
float	4	4
double	8	8
long double	16*	16
Puntero a cualquier tipo	8	8

La alineación de datos simplifica la comunicación entre CPU y memoria. Las operaciones de lectura y escritura pueden realizarse con una única operación de una palabra en memoria.

la alineación de un objeto cualquiera coincidirá con la del dato miembro con mayor alineación. El compilador insertará n bytes después de los debidos, de forma que pueda garantizarse una alineación igual a X . (Ej: struct S con variables char a, int x, char b: ) (Ej: struct S con variables int i, char a,b: ) (El primer S ocupa 12 bytes y el segundo 8 bytes).

- **sizeof():** Permite conocer el tamaño en bytes de los tipos de datos.
- **alignof():** Muestra los datos de alineación en memoria.

- **alignas():** Permite modificar la alineación natural de una variable siempre y cuando no debilite la alineación original de la misma.

```
std::cout << "size of S: " << sizeof(S) << " bytes\n"
          << "alignment of S: " << alignof(S) << " bytes\n";

constexpr auto buffer_size = std::size_t{/* número de bytes */};
alignas(std::max_align_t) unsigned char buffer[buffer_size];
```

(Free Storage) Almacenamiento Libre: es una sección de la memoria que permite la asignación dinámica mediante expresiones de tipo `new`, al contrario que los almacenados en la pila de usuario, no se limita a ámbito en que fueron creados. a memoria debe volver a ser reclamada con una función **delete**.

Ante la siguiente expresión:

```
X* p = new X { /*Argumentos*/ }
```

el código generado es algo parecido al siguiente:

```
X* p;
{
    // alojamos sizeof(X) bytes en el free store
    // (posible lanzamiento de std::bad_alloc bajo fallo de alojamiento):
    void* loc = operator new(sizeof(X));
    try {
        // construimos el objeto X en el espacio alojado:
        ::new(loc) X(/* argumentos */);
        p = static_cast<X*>(loc); // asignamos el puntero si la construcción resultó exitosa
    }
    catch (...) { // si el constructor lanza una excepción...
        operator delete(loc); // desalojamos la memoria y
        throw; // relanzamos la excepción
    }
}
```

1. Invocamos el operador **new**, para asignar un bloque de memoria alineado, de **sizeof(X)** número de bytes.

2. En caso de no tener suficiente memoria se notifica un error de tipo **std::bad_alloc**.

3. En caso de tener éxito, devuelve un puntero no nulo, que apunta al bloque de memoria asignado.

En el código `X* p = new X[n] { /*Argumentos*/ }` ocurre lo mismo, salvo que en caso de lanzarse una excepción, todos los objetos construidos en el proceso de generación de la matriz de `n` dimensiones, serán destruidos.


La variable de retorno de una expresión **new**, es un puntero que apunta a la dirección de memoria donde se aloja el objeto construido. Este puntero

dura únicamente hasta que termina el ámbito donde fue definido. Pero a pesar de esto, el objeto referenciado se quedará en el **free store** hasta ser destruido (esto puede dar lugar a fugas indeseadas de memoria, si un puntero sale fuera del ámbito pero el objeto referenciado por él permanece en el free storage (memory leak), fuga evitable mediante punteros inteligentes).

La expresión **delete** (en `X* p = new X{}; ... delete (p);`) actúa tal que en caso de que el puntero no sea nulo, se crea un destructor para el objeto al que hace referencia y se libera la sección de memoria que esté ocupada por el objeto. En el caso de una matriz de tamaño `n`, no es necesario indicar el tamaño de la matriz. Este se almacena junto con el objeto en la memoria dinámica o con una tabla asociativa.

Fragmentación del Free Storage: Supongamos un segmento virtual de 12KB de espacio de almacenamiento. Dicho segmento de memoria, en un principio está inutilizado. Suponemos que se alojan 3 objetos de 4KB de memoria cada uno, y luego se desalojan los bloques de memoria 1 y 3, la memoria queda fragmentada. Si ocurriera en caso de necesitar alojar 8KB se produciría una excepción de imposibilidad de alojar memoria, aunque entre los fragmentos quedara una

suma de 8KB.



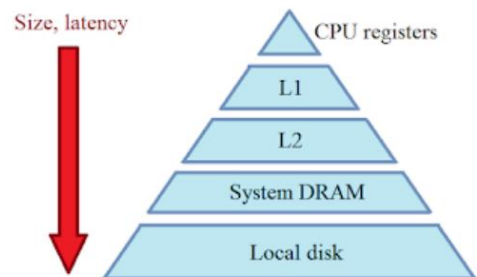
Si una operación que se va a repetir muchas veces en el tiempo necesita un buffer de tamaño conocido, el desarrollador debe alojar la memoria de una sola vez, reutilizando dicho fragmento de memoria tanto como sea necesario.

Localidad de memoria: se usa para especificar un espacio en memoria RAM que solicita algún proceso para almacenar datos. Generalmente, los algoritmos satisfacen una de las siguientes propiedades:

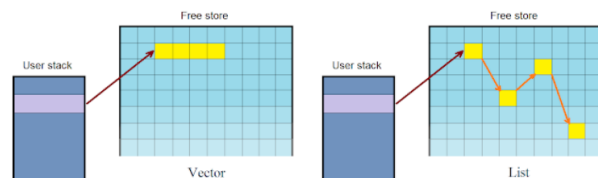
- **Localidad Espacial:** Una vez referenciada una dirección de memoria es probable que en breve se referencie una dirección contigua o próxima.
- **Localidad temporal:** Una vez referenciada una dirección de memoria, es probable que esta vuelva a ser referenciada en el futuro.

Es por esta localidad, y sus propiedades, que un ordenador puede usar las jerarquías de memoria cache para reducir los tiempos de acceso a los datos en memoria para almacenar las funciones que más se utilicen.

la memoria caché en el nivel **iv**, almacena copias de datos con instrucciones que están en el nivel inferior **iv+1**. La caché del nivel superior es más reducida que en el nivel inferior, pero más rápida a la hora de resolver las solicitudes de datos de la CPU. La memoria contenida en el nivel inferior **iv+1**, se lee y se copia en el nivel superior **iv** en bloques de tamaño fijo. En caso de que el procesador necesite un objeto del **iv+1** y este se encuentre en el **iv**, la solicitud se resuelve directamente, en caso contrario, el bloque debe copiarse al completo en el nivel **iv**.



Cuanto mayor sea la localidad de la aplicación más eficiente será. Es por ello, que se recomienda usar la plantilla de clase estándar **std::vector<std::string>** como contenedor por defecto. (como se puede ver, es mucho más eficiente su alojamiento en memoria que el de una lista).



Registros en la CPU: son espacios de memoria de acceso rápido que se encuentran integrados en el procesador, en donde se encuentran almacenadas copias de las instrucciones en ejecución, datos contenidos en la memoria física, etc.

El contador de programa contiene la dirección de memoria principal de las instrucciones en ejecución, la **CPU** completa estas operaciones sirviéndose de otros registros de datos así como de la unidad aritmético lógica (**ALU**: Circuito digital que realiza operaciones aritméticas y lógicas) del procesador.

En la arquitectura AMD64, se disponen de 16 registros con nombres especiales, este elevado número de registros permite que algunas variables locales puedan almacenarse en ellos y no en la pila de usuario. Una vez terminada la ejecución de una instrucción, el contador de programa se incrementa para acceder a la siguiente instrucción, o se modifica para acceder a cualquier otra mediante una operación de salto.

- | | |
|--------|--------|
| • %rax | • %rdi |
| • %rbx | • %rbp |
| • %rcx | • %rsp |
| • %rdx | • %r8 |
| • %rsi | • %r15 |

U2

Destruyores: cuando termina el tiempo de ejecución de un objeto, se llama automáticamente al destructor de la clase, es el encargado de liberar los recursos adquiridos en su tiempo de vida. Esta liberación se hace de forma que los recursos no se utilizan más tiempo del necesario. Sólo puede haber 1 destructor, con el mismo nombre que la clase con el signo ~ delante. No puede tener argumentos ni devolver valores y si no se declara se genera uno por defecto. Su orden es:

primero destruye los datos miembros no estáticos del objeto, luego invoca los destructores de las bases directas del objeto; es decir, primero ejecuta el cuerpo de la función y cuando acaba todos los sub-objetos son destruidos en orden inverso a su construcción mediante sus respectivos destructores.

Duración de almacenamiento:

Duración de almacenamiento	Tipo de variable afectada	Finalización del tiempo de vida
Estática	1. Declarada en un ámbito namespace (incluido el espacio de nombres global). 2. Declarada static o extern .	Al terminar la ejecución del programa.
Automática	Local, excepto si se declara static , extern o thread_local .	Cuando el flujo del programa abandona el ámbito donde la variable fue creada. En ese momento, los objetos con almacenamiento automático (variables locales) creados en dicho ámbito son destruidos en orden inverso a su construcción.
Dinámica	Alojada dinámicamente durante la ejecución del programa mediante una expresión de tipo new .	Al alcanzarse una expresión de tipo delete .
Thread-local	Declarada thread_local .	Al finalizar la ejecución del hilo donde la variable fue creada.

Como regla general, no debe llamarse directamente al destructor de una variable local. Para controlar el tiempo de vida, es suficiente si se cierra la variable en el ámbito adecuado, como se muestra en el ejemplo.

```

{
    auto x1 = X{1};
    {
        auto x2 = X{2};
        // ...
    } // x2 es destruido
    aqui
    // ...
} // x1 es destruido aqui

```

Desenredo de la pila: La emisión de excepciones es el mecanismo de información de los errores que han ocurrido durante la ejecución de un programa. Una función que sea incapaz de completar su tarea puede informar del error emitiendo una excepción mediante una expresión de tipo **throw**. El código que realiza la llamada a dicha función debe manejar los posibles errores mediante un bloque **try** y manejar las excepciones mediante cláusulas **catch**.

Al producirse una excepción dentro de un bloque **try**, se transfiere hasta la primera cláusula **catch** que pueda manejarlo. Al lanzarse el **catch**, todos los objetos que se hayan creado durante la ejecución del **try**, son destruidos en orden inverso a su creación en un proceso de desenredo de pila. A la hora de gestionar excepciones, hay que tener en cuenta:

1. Las excepciones deberían ser siempre capturadas por preferencia.
2. Los destructores no deben emitir excepciones. En caso de que sea llamado durante un desenredo de pila, se llama a una función **std::terminate**.
3. La función **std::terminate** también se invoca si no se encuentra una cláusula **catch**.
4. En caso de producirse una excepción durante la construcción de un objeto que consta de sub-objetos, se llamará a los destructores de los objetos creados en orden inverso a su creación.

Jerarquía estándar de excepciones:

```

exception
bad_alloc
bad_array_new_length
bad_cast
bad_any_cast
bad_exception
bad_function_call
bad_optional_access
bad_typeid
bad_variant_access
bad_weak_ptr
logic_error
domain_error
invalid_argument
future_error
length_error
out_of_range
<exception>
<new>
<new>
<typeinfo>
<any>
<exception>
<functional>
<optional>
<typeinfo>
<variant>
<memory>
<stdexcept>
<stdexcept>
<stdexcept>
<future>
<stdexcept>
<stdexcept>

runtime_error
ambiguous_local_time
format_error
nonexistent_local_time
overflow_error
range_error
regex_error
system_error
ios_base::failure
filesystem::filesystem_error
underflow_error
<stdexcept>
<chrono>
<format>
<chrono>
<stdexcept>
<stdexcept>
<regex>
<system_error>
<ios>
<filesystem>
<stdexcept>

namespace std {
class exception {
public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept;
    virtual ~exception();
    virtual auto what() const noexcept -> char const*;
};
}

```

El estándar del lenguaje C++ define a la clase **std::exception** donde gracias al aspecto virtual del destructor y de la función **what()** se garantiza el comportamiento polimórfico de la jerarquía.

Se distinguen dos semánticas (subclases) diferenciadas:

std::logic_error y sus derivadas informan de errores detectables antes de la ejecución del programa, mientras que **std::runtime_error** y sus derivadas informan de errores difíciles de predecir, debidos a eventos ajenos al programa que surjan durante la ejecución del mismo.

En los bloques **try-catch**, las sentencias **catch** se examinan en el mismo orden en el que se declaran, y cuando se encuentra una que permite manejar la excepción emitida, no examina las demás. El controlador con elipsis (**catch (...)** { **doSomethingElse();throw;** }) debe colocarse en último lugar, y es capaz de manejar cualquier excepción.

Garantías ante excepciones: **Básica**, **fuerte** (`vector.push_back()`, `uninitialized_copy()`) y **nothrow** (`swap()`). **Básica:** Respetan los invariantes básicos de los objetos evitando la fuga de recursos. Permite el uso de los objetos después de lanzarse la excepción. **Fuerte:** La operación concluye correctamente o emite una excepción. Si esto pasa, el programa vuelve al estado anterior a la ejecución de la operación. **Nothrow:** no se emiten ningún tipo de excepciones en ningún caso.

Técnicas RAII(Resource Acquisition In Initialization): asignación de memoria para X mediante una expresión **new**. Completadas las operaciones, se libera el bloque de memoria con la expresión **delete**. El código no es seguro ante excepciones. Al lanzarse una excepción entre la asignación de memoria y su liberación, no llegaría a liberarse pues nunca se ejecutaría la operación **delete**. Solucionable mediante:

-**bloque de memoria try-catch** usando el **catch** para operar el **delete** del objeto.

-**usando técnica RAII** donde todo recurso debe quedar representado por un objeto cuyo destructor se encarga de liberarlo.

La **técnica RAII** afecta a cualquier tipo de recurso, y no solo a la asignación dinámica de memoria. Es decir, puede emplearse las técnicas RAII para establecer un orden determinista en gestión de recursos, se desaconseja usar **new** y **delete** salvo que en las clases se implementen internamente las técnicas RAII, para ello, las bibliotecas estándar de C++ incluyen funcionalidades como punteros inteligentes, estructuras dinámicas de datos, métodos para la gestión de ficheros, métodos para la representación de cadenas de caracteres, etc...

Por lo tanto, RAII se basa en las premisas de :

- Un constructor siempre se ejecuta antes de que el objeto pueda ser usado, por lo que es un lugar seguro para reservar, inicializar, preparar los recursos a ser utilizados posteriormente.
- Los destructores son llamados implícitamente cuando el objeto se destruye, y es lo último que hace el objeto antes de liberar su propia memoria. Es el momento adecuado de liberar otros recursos usados.
- Lo único que está garantizado que se ejecutará después de una excepción son los destructores de los objetos ya creados.