Taller de Programación, Resumen para el Final.

Cátedra Veiga, 2C 2020.

Este resumen junta información relevante dictada durante las clases para la preparación del final. Ante algún error que se pueda detectar en este archivo, por favor mencionarlo.

Clase 1

Memoria

- Llamamos ***alineación*** al hecho de que las variables siempre deben arrancar a estar en posiciones de memoria que sean múltiplos del tamaño de las mismas. Esto se hace por una cuestión de eficiencia y por lo tanto hace que muchos arreglos de datos desperdicien a próposito bytes para mantener esta estructura.
- Aquellos pedazos de memoria que dejamos sin usar que nos ayuda a alinear la memoria se conoce como ***padding***.
- Recordamos que el tamaño de los datos está definido por la ISA de la arquitectura que utiliza nuestro computador.

Structs & Unions

Un ejemplo donde vemos como se manifiesta el padding, es con los ***structs & unions***. Sea el siguiente ejemplo en una arquitectura x86 de 64b:

```
struct S {
    int a;         |A|A|A|A|
    char b;        |B| | | |
    int c;         |C|C|C|C| | | | |
    short d;       |D|D|D|D|D|D|D|D|
};
```

Donde `a` ocupa los bytes 0,1,2 y 3 del mismo y luego `b` está en el byte 4 (siempre relativo al inicio de la estructura). Pero `c` no comienza en el byte 5, comienza en el 8, dejando tres bytes de padding pues necesita que el byte donde arranca sea un múltiplo de su tamaño (es decir 4). Por lo tanto antes de poner a `d` tenemos que dejar 4 bytes más de padding así puede arrancar en la dirección 16.

Esto es diferente con los unions donde el tamaño de la estructura general es el tamaño de su miembro más pesado y donde todos los datos viven en el mismo espacio de memoria.

Podemos agregar `__atribute__((packed));` antes de terminar de definir el struct para que esté no agregué padding al mismo.

Si tenemos:

```
union U {
    long l;
    int* p;
};
```

La estructura general pesará 8 bytes pues si accedo a `l` estaré leyendo el número que luego `p` interpreta como una dirección de memoria.

Ajuste de Endianness

Para ajustar el endianness de nuestros datos cuando estos pasan por sockets por ejemplo, es con este conjunto de funciones:

```
short int htons(short int);
short int ntohs(short int);
int htonl(int);
int ntohl(int);
```

Siendo `hton` host-to-network, `ntoh` network-to-host y el último caracter indica si es un short o int el dato con el que trabaja.

Si recibo un dato debería pasarlo por `ntoh` y si lo mando `hton` para ajustar el endianness.

Segmentos de Memoria

Code Segment: De lectura y ejecución. Para código y constantes. No redefinible.

Data Segment: Variables creadas al inicio del programa y son válidas hasta que termina la ejecución del programa. De acceso global o local. No redefinible.

Stack: Variables creadas al inicio del scope en el que este existe. Los datos se destruten al salir de este.

Heap: Variables compartidas por todos los programas de la máquina.

Lifetime y Scope

```
int g = 1;         // Data segment, scope global.
static int l = 2;  // Data segment, scope local.
extern int e;      // No asigna memoria, simplemente es una definición.

void Fg() {}       // Code segment, scope global.
void Fl() {}       // Code segment, scope local.
void Fe();         // No asigna memoria, simplemente es una definición.

void foo() {
    int localStack = 1;
    static int localDataSegment = -1;
    char* pointingToTheCodeSegment = "ABC";
    char stackArray[] = "ABC";
}
```

- Para utirlizar a `e`, el mismo tiene que en alguna parte del programa haber sido definido para no leer algo ni inicializado.
- Los punteros a función apuntan al code segment.

Compilación

Decimos que C/C++ son lenguajes portables pues estos pueden crear código ejecutable hacia cualquier tipo de arquitectura (siempre tendiendo en cuenta el tipo de compilador que parsea y genera el ejecutable).

El proceso de compilación tiene varias etapas:

Precompilación: Se realizan tareas relacionadas con el reemplazo de variables, búsqueda de código fuente y un parseo de texto general. Todo lo que tiene que ver con manejo de archivos, headers, etc se resuelve acá. Si compilamos en C, solo le llegarán `.c` a la etapa de compilación. Reemplaza en el código los defines correspondientes, en función de las reglas definidas por los rótulos dados (`include`, `ifndef`, `define`, `ifdef`, `endif`, `...`, todas instrucciones del precompilador).

Compilación: Verificación del código suministrado. Genera un código objeto. Los `.c` que reciben se transforman en `.o` & `.lib`.

Link-Edición: Combina los distintos códigos objetos, librerías externas, resuelve problemas de referencias cruzadas y genera el ejecutable correspondiente a la plataforma en la que se realiza esto. Para generar un ejecutable más pequeño se puede realizar un *linkeo dinámico*. Esto consiste de linkear código objeto ya pregenerado que es usado por varios programas. Estas librerias dinámicas se encuentran una vez en la memoria del sistema y permite ahorrar memoria. Puede correr más rápido si este código ya está cacheado por otro programa que lo utilizó hace relativamente poco.

Clase 2

Sockets

Siempre que se mande un mensaje por red, al destinatario del mismo se lo apoda `host`. Todo mensaje necesita conocer la ip y puerto de quien va a recibirlo. Para modelar la ip existen los protocolos iPv4 (codifica la ip con 4 bytes) e iPv6 (codifica la ip con 16 bytes).

El server hace un bind de su socket a su ip local (resultado de getaddrinfo) y le podemos decir al socket que escuche a una cierta cantidad de conexiones entrantes, a través de `listen` por el sistema operativo. A este socket le pueden llegar conexiones que puede aceptar o no, y hasta entonces bloqueará la ejecución del programa.

Del lado del cliente creamos un socket que utilizamos para conectarnos con el servidor utilizando `connect` hacia la dirección dada (obtenida con el `getaddrinfo`). Se manda el mensaje de conexión y se acepta, estableciendo la misma.

El servidor crea un segundo socket asociado al puerto y es el que acepta conexiones. Necesito uno de estos sockets extras por cada cliente con el que trabajo. Si llegan más pedidos que espacios en la pila, estos se pierden.

Para cerrar un canal de comunicación, lectura y/o escritura, se utiliza el `shutdown` .

Protocolo TPC

Envia bytes, nunca streams. Además asegura que siempre llega la información sin bytes repetidos y respetando el orden de salida.

Para satisfacer el protocolo correctamente debemos utilizar sockets, los cuales están diferenciados por su file descriptor.

Existen muchos más protocolos relacionados con la comunicación web, como el DNS, HTTP(S), etc.

Manejo de Archivos

Para manejar archivos en C podemos hacer principalmente dos cosas. Operamos con file descriptors o utilizamos la interfaz que nos provee el tipo FILE. En la materia usaremos el segundo.

Funciones importantes:

- `FILE* fopen(const char* filepath, const char* mode);`
- `size_t fread(void* ptr, size_t size, size_t count, FILE* fp );`
- `size_t fwrite(const void* ptr, size_t size, size_t count, FILE* fp );`
- `int fseek(FILE* fp, long int offset, int origin);`
- `int ftell(FILE* fp);`
- `void rewind(FILE* fp);`
- `int getc(FILE* fp);`
- `int putc(FILE* fp);`
- `char* fgets(const char* str, int n, FILE* fp);`
- `int fputs(const char* str, FILE* fp);`
- `int feof(FILE* fp);`
- `int ftruncate(FILE* fp, long int offset);`
- `int fclose(FILE* fp);`

Los modos de apertura de archivos en C son los siguientes:

- `r` : Lectura. El archivo debe existir.
- `w` : Escritura. Crea el archivo si no existe, de lo contrario lo sobreescribe.
- `a` : Escritura al final. Crea el archivo si no existe.
- `r+` : Lectura y escritura al principio. El archivo debe existir.
- `w+` : Lectura y escritura. Sobreescribe el archivo si existe.
- `a+` : Lectura y escritura al final del mismo.
- `b` : Para que se trabajen sobre archivos binarios.
- `t` : Para archivos de texto.

Clase 3

RAII

Resource Acquisition is Initialization (RAII) es un patrón de programación donde se acostumbra a pedir los recursos necesarios al sistema operativo en el constructor de un objeto y devolverlos en su destructor. Para hacer esto correctamente debemos asegurarnos que los constructores y destructores se llamen siempre para cada objeto que queremos crear, para así evitar inconsistencias con la forma de manejar recursos y evitar errores.

Pasajes de Objetos en C++

Tenemos cuatro formas de pasar objetos como parámetros en C++:

```
void A(Class obj);
void B(Class* obj);
void C(Class& obj);
void D(Class&& obj);
```

- `A` recibe por copia, es decir, al scope de A entra como obj una copia la instancia de Class que haya sido pasada por parámetro. Esto es ***peligroso*** pues si pasamos un objeto por copia que respeta RAII a un scope y este se destruye al salir del mismo (pues si un objeto sale de un scope en C++ se destruye automáticamente), podemos perder algún dato dinámico que podríamos querer usar en un futuro. Otro ejemplo peligroso es el de recibir una matriz por copia; esto podría afectar fuertemente a la performance del programa.
- `B` recibe por puntero, es decir, al scope entra la dirección de memoria donde existe el objeto. Siempre pesa que lo que pesa un puntero y evita llamadas innecesarias al destructor.
- `C` pasa por referencia. Las referencias son como punteros que *siempre deben estar inicializados y no pueden cambiar el lugar a donde apuntan*. No puedo asignar memoria para una referencia sin inicializarla.
- `D` pasa por movimiento. El método toma el ownership del objeto que recibe. En los constructores se tiende a deshabilitar el objeto entrante para desplazar el ownership del mismo a nuestro nuevo objeto.

POO en C++

Keywords

La keyword const al final de un método implica que el mismo es solo de lectura. No modifica los datos del objeto. Por ejemplo resulta útil para llamar a los métodos de un objeto `const`.

Para deshabilitar un método de una clase debemos introducir `delete` al final del mismo. Esto es útil, por ejemplo, para evitar que se llame a algún constructor por copia "sin querer".

La keyword `virtual` nos deja hacer que un método sea abstracto. Se escribe al principio de la firma del método. Agrega el método a la *vtable* (que redirecciona internamente a los métodos virtuales a las posiciones de memoria donde se encuentran los otros métodos que la implementan, siempre en función del tipo al que apuntamos).

Si un método debe si o si implementarse sobre alguno abstracto, podemos agregar `override` al final de la firma del mismo para que el compilador se fije si implementa alguna función de la vtable.

Para que un método sea de clase añadimos `static` al principio de la firma del mismo.

Herencia, Clases Abstractas e Interfaces

- No se pueden heredar constructores, sobrecargas de y friends.

- Para que un método sea puramente abstracto debemos hacer que tenga `virtual` al inicio de la firma (así tiene una entrada en la *vtable*) y lo igualamos a cero.
- Las interfaces deben tener todos sus métodos puramente abstractos. No son instanciables.
- Es muy importante tener en cuenta que *si el destructor tiene que ser abstracto,* ***debe*** *tener* `virtual`. Esto resulta ser crítico pues si heredamos de una clase que no hace nada en el destructor y nosotros como hijos hacemos RAII para nuestros propios recursos, puede pasar que no se liberen los mismos.
- Si heredamos de una clase, instanciamos al padre y en el lugar del padre metemos a una variable hija, es decir la heredera, puede ocurrir que el tamaño de la hija supere al del padre. Esto hace que las llamadas polimórficas no se activen, pues a la hija se le deben quitar los accesorios que implementó sobre el padre para que pueda encastrar en el pedazo de memoria del mismo. A este problema se lo conoce como *object slicing*. Una buena forma de solucionarlo es usando referencias o punteros, que siempre ocupan el mismo espacio. También se puede manifestar este valor en el pasaje por parámetros.
- Se conoce como *problema del diamante* al caso donde una clase A es heredada por B y C, quienes a su vez son heredadas por D (si, C++ soporta herencia múltiple y solo debería hacerse si es contra interfaces). Esto puede hacer que si A tiene algún atributo e instanciamos a D, este no sabrá si usar al atributo de B o C. El compilador se encargará de levantar el error correspondiente.

Polimorfismo

Ejemplo de polimorfismo con herencia:

```
class Father {
  public:
    void method() {
      std::cout << "Base" << std::endl;
    }
};

class Son : public Father {
  public:
    void method() {
      std::cout << "Son" << std::endl;
      // Podría decorar al método del hijo con el del padre:
      // Father::method();
    }
};

class VirtualFather {
  public:
    void method() {
      std::cout << "Base" << std::endl;
    }
};

class VirtualSon : public VirtualFather {
```

```
  public:
    virtual void method() override {
      std::cout << "Son" << std::endl;
    }
};

Father* f;
Son s;
VirtualFather* vf;
VirtualSon vs;
f = &s;
vf = &vs;
f->method();   // Prints "Base"
vf->method();  // Prints "Son"
```

La resolución de problemas dinámicos de la vtable se la conoce como `Late Binding` o `Dynamic Binding`. De lo contrario, si se resuelve en la compilación, se lo conoce como `Static Binding`.

Clase 4

Programación Multihilo

En las arquitecturas de las máquinas que utilizamos podemos darnos cuenta que la ejecución de un proceso que dura un tiempo determinado, hace que la máquina tenga que trabajar durante ese tiempo completo. Pero eso no necesariamente implica que la CPU va a estar computando valores durante todo ese tiempo; es más, puede ocurrir que ciertas operaciones lentas representen la mayoría del tiempo requerido por estos procesos. Esto hace que desperdiciemos tiempo de cómputo de la CPU. Para evitar esto se hace multitasking, donde el kernel, con su *Scheduler* corran de forma concurrente.

Y concurrente no significa paralelo. Es concurrente, pues el Scheduler selecciona constantemente que proceso tiene acceso a la CPU, haciendo que la ejecución de cada uno de estos se transforme en una ejecución de a ráfagas más rápidas. Solo cambiamos la forma en que procesamos datos y esto no implica tener una ventaja temporal, hasta que pensamos en lo dicho en el parrafo anterior. Muchas veces estos cambios que realiza el Scheduler, hace que mientras un proceso que tarda (por ejemplo con la espera a una búsqueda en memoria) y está ocupando la CPU haciendo nada, pueda ser desalojado un rato por otro que aproveche y compute datos.

Estos saltos entre actores que ocupan el CPU por el sistema operativo se lo conoce como *context switching*.

Un proceso puede estar en los siguientes estados:

- Inicialización
- Espera a la CPU
- Utilizacón de CPU
- Bloqueados (por I/O, etc)
- Completados

El paralelismo real se realiza con los cores del procesador. A estos hilos se los conoce como *hilos pesados, heavy o del kernel*. Los *hilos verdes o de usuario* a

diferencia del resto, está controlado por un pseudo-scheduler y es este mismo el que decide que proceso entra o no en el hilo del kernel, el real. El context switching de estos pseudo-scheduler está delegado por el context switching del OS; le sacamos peso de encima, pero no le permite realizar tareas realmente paralelas.

Diferencias entre Procesos e Hilos

Cada proceso puede compartir simplemente su code segment.

Los hilos en cambio, comparten el code segment, el heap, el data segment y los file descriptors. Todos estos están separados para los procesos.

Los dos no tienen acceso al stack y registros de sus compañeros.

Mutex & Locks

Como diferentes hilos pueden acceder al mismo grupo de variables, esto puede introducir condiciones de carrera, es decir, se pierde el orden de instrucciones ejecutadas. Para mantener al mismo y asegurar la atomicidad de la ejecución, se introducen los mutex. Estos permiten realizar un lock sobre un segmento de código; si un hilo bloquea cierto segmento de código, ningún otro podrá entrar hasta que el primero lo desbloquee. Si esto último no ocurre, estaremos en presencia de un *deadlock*.

En C++ podemos bloquear un hilo con `std::mutex`, el cual requiere un lock y unlock (debemos minimizar las llamadas a estos dentro de los métodos y funciones para evitar deadlocks inesperados). También se puede utilizar `std::unique_lock` (el cual brinda la ventaja de ser RAII pues, al salir del scope donde hace el lock, termina haciendo el unlock automaticamente).

Clase 5

Templates

En C/C++ podemos hacer algo parecido a los templates aprovechando la magia del precompilador:

```
#define MAKE_ARRAY_STRUCT(TYPE) \\
  class Array_##TYPE { \\
    TYPE data[64]; \\
    void set(int p, TYPE v) { data[p] = v; } \\
    TYPE get(int p) { return data[p]; } \\
  }
```

Y con templates:

```
template<class T>
class Array {
  T data[64];
  void set(int p, T t) { data[p] = t; }
  T get(int p) { return data[p];}
}
```

Podemos crear objetos personalizados. Acepta tanto tipos primitivos como clases propias.

También acepta más de un token y puede tener un valor por default:

```
template<class T, class U>
class Dupla {
  T t;
  U u;
}

template<class T=char, int size=64>
class ArrayVariable {
  T data[size];
}
```

Si quiero que cuando el tipo del template sea uno en particular, se le pueda hardocodear la implementación de la siguiente forma:

```
template<class T>
class Array { ... }

template<>
class Array<bool> { ... }
// Si T == bool → definí a la clase Array booleana de la forma que te hardcodeo acá.
```

Este estilo de programación es conocida como programación genérica.

Sobrecarga de Operadores

Supongamos que tenemos las instancias `x1`, `x2` y queremos operar sobre ellas de la siguiente forma `x1 <OP> x2`. Esto en C++ se puede representar así:

```
R resultado = x1 <OP> x2;
R resultado = x1.operator<OP>(x2);
```

Y definir así:

```
R operator<OP>(const X& other) const {...}
```

Otros ejemplos particulares son:

```
class Complex {

  public:

  float re, im;

  Complex(float re, float im) : re(re), im(im) {}

  Complex& operator=(const Complex& other) {
    if (this != &other) {
      this->re = other.re;
```

```
      this->im = other.im;
    }
    return *this;
  }

  Complex operator+(const Complex& other) const {
    Complex resultado(this->re + other.re, this->im + other.im);
    return resultado;
  }

  // Sobrecargar un casteo implica que no es necesario decir el tipo que devuelve pues
es obvio.
  operator float() const {
    return sqrt(this->re * this->re + this->im - this->im);
  }

};
```

Se conoce como *functor* a la llamada del `operator()`.

STL

Esta librería, haciendo uso de templates, implementa, entre otras cosas, un conjunto importantísimo y extenso de estructuras de datos. Es muy recomendable trabajar con estas y no con cualquier implementación personal la cual no está sujeta a las optimizaciones que la STL pueda realizar.

Clase 6

Manejo de Errores

La keyword `throw` permite lanzar cualquier cosa como una excepción. Esto, o la aparición natural de una excepción para la ejecución del programa para hacer *bubble up*. Una forma óptima de atrapar excepciones es con un bloque *try-catch*:

```
try {
  doStuffWithSockets();
} catch (SocketException& e) {
  return;
} catch (std::exception& e) {
  std::cerr << e.what() << std::endl;
} catch (...) {
  std::cerr << "Unkown error" << std::endl;
}
```

En el ejemplo vemos que pueden pasar varias cosas. Si se levanta una `SocketException` (que por ejemplo represente que me cerraron un socket que acepté) no hago nada y termino la ejecución. Si atrapo una `std::exception` o algo que herede de esto imprimo la descripción del mismo. Finalmente mantengo un catch genérico para cualquier cosa que escape a eso.

Es útil meter uno de estos bloques por cada hilo que lanzo y en las "capas superiores" de nuestro hilo principal.

Para evitar tener errores con la devolución de recursos en un bloque catch, hacemos uso del patrón RAII. Wrappeamos todo pedido de recursos en el constructor de una clase y la liberación en el destructor para asegurar que este último se llamará automaticamente al salir del try. Este mismo truco se puede realizar para wrappear objetos que pueden evitar tener estas fallas en el constructor de un objeto (pues ahí no se llamaría al destructor).

Una api es *exception safe strong* si es resistente a excepciones, es decir, está bien preparado para lidiar con estas. De lo contrario es *exception safe weak*.

Modelo Cliente-Servidor

Este patrón se utiliza para establecer comunicaciones desde el servidor hacia varios clientes.

Digamos que para cierto momento n clientes conectados a nuestro server, por lo tanto nuestro modelo propone lanzar n+2 hilos.

 Servidor manejando varios clientes

El primero será el hilo principal. Este tendrá algún objeto que pueda lanzar al segundo hilo. Este será el hilo con el socket aceptador por cada cliente que recibe lanza un hilo para lidiar con ellos. El destructor debe tener un recolector de basura que elimina los recursos y cierra los hilos y sockets individuales. El hilo principal, cuando se de cuenta de que debe cerrar al aceptador, se llama a su destructor, cierra las conexiones y libera todos los recursos.

Programación Orientada a Eventos

Podríamos definir a cada evento como un suceso cuya aparición ante el sistema es aleatorio. Esto hace que, si nuestro código trabaja con estos eventos, tengamos que estar esperando constantemente a que alguno se manifieste. Esto intuitivamente se puede hacer utilizando mezclas entre concurrencia y estructuras de datos.

Un buen ejemplo es el de la cola de eventos. La misma apila objetos que representen eventos a medida que van entrando y con un mutex los sacamos manteniendo el orden.

Es además muy útil con el controlador del patrón MVC y hacer las verificaciones de eventos lo más rápido y sencillo posible.

Clase 7

SDL

Un programa que inicializa una ventana es la siguiente:

```
int main(int argc, char* argv[]) {
  SDL_INIT(SDL_INIT_VIDEO);
  SDL_Window* window;
```

```
  SDL_Renderer* renderer;
  SDL_CreateWindowAndRenderer(WIDTH, HEIGHT, SDL_RENDERER_ACCELERATED, &window,
&renderer);
  SDL_Event event;

  bool running = true;
  while (running) {
    while (SDL_PollEvent(&event)) {
      if (event.type == SDL_QUIT) running = false;
    }
    doCoolStuff(window, renderer);
  }

  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);
  SDL_Quit();

  return 0;
}
```

Podemos utilizar funciones como `SDL_GetWindowSize`, `SDL_SetRenderDrawColor`, `SDL_DrawLine`, `SDL_RenderClear` y `SDL_RenderPresent`.

Clase 8

Namespaces

Nos permite diferenciar instancias y variables. Por ejemplo:

```
namespace Black {
  class Cat { ... }
}

namespace Orange {
  class Cat { ... }
}
```

Nos permite diferenciar entre `Black::Cat` y `Orange::Cat`.

Si un namespace es anónimo, cualquier instancia de lo que se encuentre adentro del mismo podrá existir en el módulo en que se lo definió. Eso permite generar una funcionalidad similar a la de *static*.

La keyword `using`, al utilizarla con namespaces nos permite obviar la mención de estos en nustros módulo de código.

Friendship

Nos permite acceder a atributos privados de nuestros "amigos". No se hereda. Un buen ejemplo está en el siguiente código, donde implementamos un método que nos permite imprimir los valores de nuestra clase Stack, sin tener getters.

```
class Stack {
  ...
  friend std::ostream& operator<< (std::ostream& stream, const Stack& stack) {
    stream << "[";
    int len = stack.buffer.size();
    for (int i = 0; i < len; i++) {
      stream << stack.buffer[i];
    }
    stream << "]";
    return stream;
  }
}
```