Onboarding - Recap Memoria en C++

En C++, el programador tiene el poder de controlar donde y en que momento los objetos son reservados en memoria.

Otros lenguajes ofrecen un control similar como C y Rust.

Un buen manejo de la memoria permite ejecuciones más rápidas (es lo que *realmente* le permiten a C/C++/Rust ser más rápidos que Java o Python).

Pero ciertamente es más difícil y propenso a errores.

no pain, no gain

En Taller le dedicaremos varias clases a esto pero por ahora este *recap* esta para que entres en calor.

Nota: tal vez quieras hacer primero "Recap - Proceso de Building y Testing" antes de continuar.

¡Tu respuesta es exitosa! Tu calificación es 100.0%. [Tarea #66c95fd06096f1682c4593a5]

Pregunta 1:

Como se organiza los datos en memoria tiene un impacto enorme en la performance de un programa.

C++ nos da varias herramientas para ello, algunas con ayuda del compilador.

`sizeof` es un operador que determina en tiempo de compilación el tamaño de una variable o estructura.

Cual sería el resultado de `sizeof(uint16_t)` en bytes (asumir que `sizeof(char)` es 1) ?

Referencias:
- stdint
- sizeof

○ **Depende** de la arquitectura y de los flags del compilador pero en general son 2 bytes.

○ `uint16_t` es de 16 bytes.

● `uint16_t` es de 2 bytes y **no depende** de la arquitectura ni de los flags del compilador

○ **Depende** de la arquitectura y de los flags del compilador pero en general son 4 bytes.

○ `uint16_t` es un `int` por lo tanto son 4 bytes.

○ `uint16_t` es de 2 bytes pero **depende** de la arquitectura y de los flags del compilador

Pregunta 2:

Algunos tipos de variables en C++ estan garantizado tener un tamaño fijo, otros, un tamaño mínimo.

Cuales de las siguientes afirmaciones son correctas? Hay múltiples respuestas, **marcarlas todas!**

Asumir que `sizeof(char)` es 1 bytes y que 1 byte tiene 8 bits.

Referencias:
- Post sobre sizeof
- stdint
- Stardard type

☐ `int_fast8_t` tiene exactamente 1 byte.

☑ `int_least32_t` tiene al menos 4 bytes.

☑ `unsigned int` tiene al menos 2 bytes.

☑ `short` tiene al menos 2 bytes.

Pregunta 3:

Se tiene la siguiente estructura:

```
struct foo_t {
    uint8_t i;
    uint32_t j;
}
```

Cuales de las siguientes afirmaciones son correctas? Hay mas de una, **marcarlas todas!**

☑ `sizeof(struct foo_t)` tiene al menos 5 bytes.

☑ El campo `j` esta alineado a 4 bytes (asumir una arquitectura de 64 bits y configuración por default en el compilador)

☐ `sizeof(struct foo_t)` tiene exactamente 5 bytes.

☐ En una configuración por default de `gcc`, la estructura **no** tiene padding ya que `gcc` optimiza por velocidad y el padding hace que el programa sea más **lento**.

☐ La estructura **si** tiene padding y no depende ello ni de la arquitectura ni del compilador.

☐ La estructura **no** tiene padding y no depende ello ni de la arquitectura ni del compilador.

☐ El campo `j` esta alineado a 8 bytes (asumir una arquitectura de 64 bits y configuración por default en el compilador)

☑ En una configuración por default de `gcc`, la estructura **si** tiene padding ya que `gcc` optimiza por velocidad y el padding hace que el programa sea más **rápido**.

Pregunta 4:

No importa en que lenguajes programes (C++, Python, Javascript, ...), al final del día estas trabajando con bytes en memoria.

Sea cuando tenes que serializar los objetos para escribirlos en un archivo binario o enviarlos por red, la representación byte a byte cuenta.

Supone el siguiente código:

```
uint32_t i = 0x41424344;
char *p = (char*)&i;
```

Cuales de las siguientes afirmaciones es correcta? Hay mas de una, **marcarlas todas!**

Es importante q **entiendas** el por que. Si resolves el ejercicio correctamente pero te quedaron dudas, **anotalas** y llevalas a clase o preguntalas en el Discord.

Referencias:
- Post sobre en endianness
- La clase de Taller q hablamos de endianness

☑ `p[0]` es `0x41` en hexadecimal si el endianness del host es *big endian*.

☑ La variable `i` tiene el número `1094861636` en decimal y no depende del endianness del host.

☑ `p[0]` es `0x44` en hexadecimal si el endianness del host es *little endian*.

☐ La variable `i` puede tener el número `0x41424344` o `0x44434241` dependiendo del endianness del host.

Pregunta 5:

Supone el siguiente código (similar al anterior):

```
uint32_t i = 0x41424344;
char *p = (char*)&i;

cout << p[0] << p[1] << p[2] << p[3] << "\n
```

Cuales de las siguientes afirmaciones es correcta? Hay mas de una, **marcarlas todas!**

Es importante q **entiendas** el por que. Si resolves el ejercicio correctamente pero te quedaron dudas, **anotalas** y llevalas a clase o preguntalas en el Discord.

(Si, te lo repito por que aunque sea molesto, es preferible esto antes q te quedes con dudas q luego solo haran q tardes mucho más en resolver los TPs)

Referencias:
- Post sobre en endianness
- La clase de Taller q hablamos de endianness

☐ Se imprime por salida estándar `"ABCD"` independientemente del endianness del host.

☐ Para garantizar que se imprima por salida estándar `"ABCD"` independientemente del endianness del host es necesario convertir el valir `i` **antes** del casteo usando `ntohl()` (*network to host long*)

☑ Se imprime por salida estándar `"ABCD"` solo si el endianness del host es *big endian*.

☑ Para garantizar que se imprima por salida estándar `"ABCD"` independientemente del endianness del host es necesario convertir el valir `i` **antes** del casteo usando `htonl()` (*host to network long*)

☐ Se imprime por salida estándar `"ABCD"` solo si el endianness del host es *little endian*.

Pregunta 6:

En C/C++ podes ver a un mismo conjunto de bytes como uno u otro tipo.

Esto se llama casteo (cast en ingles).

Supone el siguiente código:

```
uint32_t i = 0x41424344;

uint8_t  b = (uint8_t) i;   // vemos un int 
char     c = (char) i;      // vemos un int

uint32_t q = i >> 8;        // operacion ent
uint32_t a = i &  1;        // operacion ent
uint32_t z = i %  2;        // operacion ent
```

Cuales de las siguientes afirmaciones es correcta? Hay mas de una, **marcarlas todas!**

Y no olvides q podes compilar el código y verlo con tu debugger favorito!

Referencias:
- Post sobre en endianness (si, otra vez)

☐ Tanto la variable `a` como `z` tienen el valor numérico del bit menos significativo de la variable `i`. Esto equivale a obtener el bit menos significativo del último byte que en una máquina big endian será `0x44` y en una máquina little endian será `0x41`. (O sea que el valor de `a` y `z` dependen del endianess de la máquina).

☑ La variable `q` tiene el valor numérico de la variable `i` shifteado 8 bits a la derecha. Esto equivale a perder el byte menos significativo que es `0x44` independientemente del endianess de la máquina.

☑ Tanto la variable `a` como `z` obtienen la paridad de la variable `i`.

☑ Shiftear a la derecha equivale a dividir por una potencia de 2 no importa el orden de los bytes de la variable a shiftear. En el caso de `q`, la variable `i` fue dividida por 256.

☐ La variable `b` tiene el valor numérico del último byte del número `0x41424344` si la máquina está en big endian será `0x44` mientras que si la máquina está en little endian será `0x41`.

☑ Tanto la variable `a` como `z` tienen el valor numérico del bit menos significativo de la variable `i`. Esto equivale a obtener el bit menos significativo del byte menos significativo que será `0x44` independientemente del endianess de la máquina.

☑ La variable `b` tiene el valor numérico del byte menos significativo del número `0x41424344` no importa en que endianness este la máquina y será `0x44`

☐ Tanto la variable `a` como `z` tienen el valor numérico del último bit del último byte de la variable `i`. Esto equivale a obtener el último bit de `0x44` en una máquina big endian o el último bit de `0x41` en una máquina little endian. (O sea que el valor tanto de `a` como de `z` depende del endianess de la máquina).

☐ La variable `q` tiene el valor numérico de la variable `i` shifteado 8 bits a la derecha. Esto equivale a perder el último byte que en una máquina big endian será `0x44` y en una máquina little endian será `0x41`. (O sea que el valor de `q` depende del endianess de la máquina).

☐ Shiftear a la derecha equivale a dividir por una potencia de 2 no importa el orden de los bytes de la variable a shiftear. En el caso de `q`, la variable `i` fue dividida por 8.

Pregunta 7:

Más casting en C/C++.

Presta particular atención a este ejercicio. Cuando en el TP tengas que serializar y deserializar a bytes y/o tengas que hacer cosas con shifts saber bien que sucede **te ahorrará problemas a futuro**.

Supone el siguiente código:

```
char buf[4] = {0x41, 0x42, 0x43, 0x44};

uint32_t ret = *(uint32_t*) buf;   // vemos u

uint16_t r1 = *(uint16_t*) buf;     // vemos
uint16_t r2 = *((uint16_t*) buf+1);   // vemo
```

Cuales de las siguientes afirmaciones es correcta? Hay mas de una, **marcarlas todas!**

Referencias:
- Post sobre en endianness (si, otra vez)
- Clase de Taller sobre Memoria y Endianness

☐ La variable `ret` tiene el valor numérico `0x41424344` independientemente del endianess de la máquina.

☑ La variable `ret` tiene el valor numérico `0x41424344` si se está en una máquina big endian o tendrá el valor numérico `0x44434241` si se esta en una máquina little endian.

☐ Las variables `r1` y `r2` tienen los valores numéricos `0x4142` y `0x4344` independientemente del endianess de la máquina.

☑ Para garantizar que las variables `r1` y `r2` tengan los valores numéricos `0x4142` y `0x4344` independientemente del endianess de la máquina se debe usar `ntohs`.

☐ Las variables `r1` y `r2` tienen los valores numéricos `0x4142` y `0x4344` respectivamente si se está en una máquina big endian o tendrán los valores numéricos `0x4443` y `0x4241` respectivamente si se está en una máquina little endian.

☑ Para garantizar que la variable `ret` tenga el valor numérico `0x41424344` independientemente del endianness se debe usar `ntohl`.

☑ Las variables `r1` y `r2` tienen los valores numéricos `0x4142` y `0x4344` respectivamente si se está en una máquina big endian o tendrán los valores numéricos `0x4241` y `0x4443` respectivamente si se esta en una máquina little endian.

Pregunta 8:

`cppcheck` dice que hay un posible *buffer overflow* en el siguiente código.

```
char buf[30];
strcpy(buf, otherbuf);
```

Cuales de las siguientes afirmaciones son correctas? Hay mas de una, **marcarlas todas!**

Nota como una herramienta q te lleva segundos ejecutar puede detectarte errores te salva de invertir horas en debugging. Medita sobre ello.

Referencias:
- Post sobre strncpy

☐ Al reemplazar `strcpy` por `strncpy` evitamos **todo** tipo de error.

☐ Es un falso positivo de `cppcheck`. No hay forma que `strcpy(buf, otherbuf)` tenga un *buffer overflow*.

☐ Hay que reemplazar la llamada a `strcpy` por `strncpy(buf, otherbuf, sizeof(otherbuf))` que contempla el tamaño del buffer *fuente* para evitar el overflow.

☑ Hay que reemplazar la llamada a `strcpy` por `strncpy(buf, otherbuf, sizeof(buf))` que contempla el tamaño del buffer *destino* para evitar el overflow.

☑ `cppcheck` tiene razón, si `otherbuf` tiene más de 30 bytes tendrás un *buffer overflow*.

☐ `strncpy` **siempre** deja un `\0` al final de `buf` entonces el string es seguro para ser usado luego.

☑ `strncpy` deja un `\0` al final de `buf` **pero no siempre**.

Pregunta 9:

Supone el siguiente código:

```
int foo = 42;
int bar; // sin inicializar
std::cout << foo << bar << zaz << "\n"
```

Simple eh? Imaginate que se ejecuta y termina en un *segmentation fault*.

Que se puede deducir?

Cuanto más entrenes haras mejores deducciones y te ahorraras tiempo de debugging.

No pierdas la oportunidad de entender esto!

Referencias:
- Post sobre segfaults
- Bonus track opcional

○ `std::cout` es un archivo que debe abrirse explícitamente, seguro que el programador se olvido de hacerlo.

○ `bar` no esta inicializada, por eso el *segmentation fault*.

● `foo` y `bar` están con memoria reservada, la única que no esta clara es `zaz`; debe haber quedado con memoria sin reservar y de ahí el *segfault*

Información

Autor (res)	Martin Di Paola
Fecha de entrega	03/09/2024 18:00:00
Estado	Succeeded
Calificación	100%
Promedio ponderado	1.0
Número de intentos	41
Tiempo límite de envío	Sin límite de envío

Enviado como

Santiago Jorda - 102924

Grupo: Default classroom

Para evaluación

Tu mejor envío es

24/08/2024 01:21:36 - 100.0%

Bitácora de envíos

24/08/2024 01:21:36 - 100.0%