

# Memoria dinámica

Algoritmos y Estructuras de Datos II - 2020C1

# Repaso: arreglos estáticos

## Arreglos estáticos

C++ soporta nativamente arreglos estáticos, cuyo tamaño está fijo en tiempo de compilación:

```
int main() {  
    int arreglo_estatico[10];  
    for (int i = 0; i < 10; i++) {  
        arreglo_estatico[i] = i * i;  
    }  
    for (int i = 0; i < 10; i++) {  
        cout << arreglo_estatico[i] << endl;  
    }  
}
```

Además, tampoco son parametricos, explicaremos elm motivo mas adelante.

# Memoria dinámica: motivación

Queremos implementar una versión simplificada de `std::vector`<sup>1</sup>:

```
template<class T>
class Vec<T> {
public:
    Vec();
    int size() const;
    T get(int i) const;
    void set(int i, T x);
    void push_back(T x);
private:
    ...
};
```

---

<sup>1</sup>...sin usar `std::vector`.

# Memoria dinámica: motivación

¿Qué representación elegimos?

- ▶ No alcanza con un arreglo estático (¡no es parametrico!).
- ▶ Cada vez que hacemos un `push_back` tenemos que reservar espacio para guardar el nuevo elemento.
- ▶ Necesitamos entender el modelo de memoria de C++.

# Modelo de memoria

En C++ la memoria es un arreglo de *Bytes*.

Un Byte es típicamente un entero de 8 Bits (0..255).

Cada Byte de la memoria tiene una única *dirección*.

## Representación de variables locales

```
int main() {  
    int foo = 123;  
    int bar = 1000000;  
    char baz = 'A';  
    ...  
}
```

|       | Dirección | Byte |
|-------|-----------|------|
|       | ...       | ...  |
| foo:0 | 9000      | 123  |
| foo:1 | 9001      | 0    |
| foo:2 | 9002      | 0    |
| foo:3 | 9003      | 0    |
| bar:0 | 9004      | 145  |
| bar:1 | 9005      | 96   |
| bar:2 | 9006      | 15   |
| bar:3 | 9007      | 0    |
| baz   | 9008      | 65   |
|       | ...       | ...  |

# Modelo de memoria

## Representación de estructuras

| <code>struct Par {</code>      |                           | Dirección | Byte |
|--------------------------------|---------------------------|-----------|------|
| <code>int x;</code>            |                           | ...       | ...  |
| <code>char y;</code>           | <code>pares[0].x:0</code> | 9000      | 10   |
| <code>};</code>                | <code>pares[0].x:1</code> | 9001      | 0    |
|                                | <code>pares[0].x:2</code> | 9002      | 0    |
| <code>int main() {</code>      | <code>pares[0].x:3</code> | 9003      | 0    |
| <code>Par pares[2];</code>     | <code>pares[0].y</code>   | 9004      | 65   |
| <code>pares[0].x = 10;</code>  | <code>pares[1].x:0</code> | 9005      | 20   |
| <code>pares[0].y = 'A';</code> | <code>pares[1].x:1</code> | 9006      | 0    |
| <code>pares[1].x = 20;</code>  | <code>pares[1].x:2</code> | 9007      | 0    |
| <code>pares[1].y = 'B';</code> | <code>pares[1].x:3</code> | 9008      | 0    |
| <code>...</code>               | <code>pares[1].y</code>   | 9009      | 66   |
| <code>}</code>                 |                           | ...       | ...  |

**Nota:** los detalles de representación pueden variar dependiendo de la arquitectura y del compilador.

# Punteros

El tipo  $T^*$  es el tipo de los **punteros a T**.

Un puntero a T representa una dirección de memoria en la que (presumiblemente) hay almacenado un valor de tipo T.

- ▶  $\text{int}^*$ : puntero a int
- ▶  $\text{char}^*$ : puntero a char
- ▶  $\text{vector}<\text{int}>^*$ : puntero a vector de int
- ▶  $\text{vector}<\text{int}^*>$ : vector de punteros a int
- ▶  $\text{int}^{**}$ : puntero a puntero, por ejemplo, para tener una lista dinamica de listas
- ▶ ...

## Operaciones con punteros

- ▶ Dirección de memoria de una variable.  $(\&\text{variable})$   
si  $\text{variable}$  es de tipo T  
 $\&\text{variable}$  es de tipo  $T^*$
- ▶ Valor almacenado en una dirección de memoria.  $(*\text{puntero})$   
si  $\text{puntero}$  es de tipo  $T^*$   
 $*\text{puntero}$  es de tipo T

# Punteros

## Punteros a variables locales

```
int main() {  
    int x = 10;  
    int* px = &x; // obtengo la direccion de x  
    cout << px << endl; // la direccion de apunta a x  
    cout << *p << endl; //el valor de x (10)  
    *px = *px + 1;  
    cout << x << endl; // ahora vale 11  
  
    int* q = &7;  
}
```



# Punteros

## Punteros a variables locales

```
int main() {  
    int x = 10;  
    int* px = &x; // obtengo la direccion de x  
    cout << px << endl; // la direccion de apunta a x  
    cout << *p << endl; //el valor de x (10)  
    *px = *px + 1;  
    cout << x << endl; // ahora vale 11  
  
    int* q = &7;  
}
```

co.cpp:10:13: error: lvalue required as unary '&' operand

```
    int* q = &7;
```

# Punteros

## Punteros a estructuras

```
struct Par {  
    int x;  
    char y;  
};  
  
int main() {  
    Par pares[2];  
    Par* p = &pares[1]; // puntero al 2do par(0..1)  
    (*p).x = 10;  
    p->y = 'b'; //"->" acceso directo a los miembros de par  
    cout << p->x << endl; //10  
    char* q = &p->y; //*q tiene aliasing con pares[1].y  
    *q = 'c';  
    cout << pares[1].y << endl; //'c'  
}
```

# Punteros

## La keyword `NULL`

Históricamente, la dirección de memoria 0 está reservada para representar un puntero que no referencia ninguna posición de memoria.


En C++ se puede escribir `NULL` para indicar que un puntero tiene la dirección 0. Sin embargo, esto es un mero reemplazo sintáctico.

## La keyword `nullptr`

A partir del estándar C++11, se incorporó la palabra reservada `nullptr`<sup>2</sup>, en reemplazo de `NULL`. Esto refiere, ya no al valor 0, sino al tipo de datos `nullptr_t`, que sirve para representar a los punteros nulos.

Usar `nullptr` es más prolijo que usar `NULL`, ya que se le está dando un componente semántico (un significado) al valor.

---

<sup>2</sup><https://en.cppreference.com/w/cpp/language/nullptr> 

# Regiones de memoria

La memoria en C++ se divide en tres tipos/regiones:

Global (Estática)  $\Rightarrow$  En el ejecutable

La memoria estática se encuentra incrustada en el ejecutable.

Local (Automática)  $\Rightarrow$  En la pila

La memoria en la pila se administra automáticamente.

Dinámica  $\Rightarrow$  En el *heap*

La memoria en el *heap* se administra manualmente.

# Global (Estática)

Las variables estáticas existen durante todo el programa

```
int x = 42;
```

```
int main() {  
    int* p = &x;  
    cout << *p << endl;  
}
```

# La pila

La memoria en la pila se administra **automáticamente**.

En C++ las variables locales y los parámetros se almacenan en la pila. El tiempo de vida de una variable está dado por su *scope*.

- ▶ Al declarar una variable local, se apila su valor.
- ▶ Cuando el *scope* de la variable finaliza, se desapila automáticamente su valor.

# La pila

Las variables en la pila existen sólo en su propio scope

```
void f() {  
    int x = 42;  
}
```

```
int main() {  
    f();  
    int* p = &x;  
    cout << *p << endl;  
}
```

¿Qué sucede al intentar compilar?

# La pila

Las variables en la pila existen sólo en su propio scope

```
void f() {  
    int x = 42;  
}
```

```
int main() {  
    f();  
    int* p = &x;  
    cout << *p << endl;  
}
```

¿Qué sucede al intentar compilar?

test.cpp: En la función "int main()":

test.cpp:8:13: error: 'x' no se declaró en este ambito

```
    int* p = &x;
```



# La pila

## Tiempo de vida de una variable en la pila

```
void g(int* p) {  
    cout << *p << endl; // OK  
}
```

```
int* f() {  
    int x = 42;  
    g(&x);  
    return &x;  
}
```

```
int main() {  
    int* p = f();  
    cout << *p << endl; // Segmentation fault  
}
```

## El *heap*

La memoria en el *heap* se administra **manualmente**.

C++ provee dos operaciones para administrar la memoria dinámica:

- ▶ **new** T — reserva espacio en el *heap* para almacenar un valor de tipo T. Devuelve un puntero de tipo T\* a la dirección de memoria donde comienza ese espacio.
- ▶ **delete** p — libera la memoria asociada al puntero p.

# El *heap*

## Tiempo de vida de una variable en el *heap*

```
int* f() { //Stack; Heap  
    int* p = new int; // p; basura  
    *p = 42; //p; 42  
    return p;  
}
```

```
int main() {  
    int* q = f(); //q; 42  
    cout << *q << endl; // OK  
    delete q; //q; 42 (¡pero no es mas accesible!)  
}
```

C++ sabe que puede reutilizar ese espacio, es una buena practica inicilizar variables para no depender de los valores default

```
int* p = new int(7);
```

## El *heap*

También se pueden reservar arreglos de tamaño *dinámico*, cuyo tamaño se elige en tiempo de ejecución:

- ▶ `new T[n]` — reserva espacio en el *heap* para almacenar contiguamente *n valores* de tipo T. Devuelve un puntero de tipo  $T^*$  a la dirección de memoria donde comienza ese espacio.
- ▶ `delete[] p` — libera la memoria asociada al arreglo que empieza en la dirección p.

# Implementación de Vec<T>

Podemos completar la implementación de Vec<T>:

```
template<class T>
class Vec<T> {
public:
    Vec();
    int size() const;
    T get(int i) const;
    void set(int i, T x);
    void push_back(T x);
private:
    ???
};
```

# Implementación de Vec<T>

Podemos completar la implementación de Vec<T>:

```
template<class T>
class Vec<T> {
public:
    Vec();
    int size() const;
    T get(int i) const;
    void set(int i, T x);
    void push_back(T x);
private:
    int _capacidad;
    int _tam;
    T* _valores;
};
```

## Implementación de Vec<T>

```
template<class T> Vec<T>::Vec() : _capacidad(1),  
                                _tam(0),  
                                _valores(new T[1]) { }  
  
template<class T> int Vec<T>::size() const {  
    return _tam;  
}  
  
template<class T> T Vec<T>::get(int i) const {  
    return _valores[i];  
}  
  
template<class T> void Vec<T>::set(int i, T x) {  
    _valores[i] = x;  
}
```

# Implementación de Vec<T>

La estrategia es que cada vez que tamaño llega a capacidad:

- ▶ (1) Solicito capacidad \* 2
- ▶ (2) Me copio los datos viejos en la nueva estructura
- ▶ (3) libero la memoria de la vieja estructura
- ▶ (4) actualizo las variables



## Implementación de Vec<T>

```
template<class T>
void Vec<T>::push_back(T x) {
    if (_tam == _capacidad) {
        T* nuevo = new T[2 * _capacidad]; //(1)
        for (int i = 0; i < _capacidad; i++) { //(2)
            nuevo[i] = _valores[i]; //(2)
        }

        delete[] _valores; //(3)
        _capacidad = 2 * _capacidad; //(4)
        _valores = nuevo; //(4)
    }
    _valores[_tam] = x;
    _tam++;
}
```

# Problemas con punteros

## Problema con punteros: *leaks*

- ▶ Cada vez que se hace un `new T`, se debe hacer un `delete` de esa dirección de memoria posteriormente.
- ▶ De lo contrario el programa *pierde memoria* (tiene un *leak*).

```
int main() {  
    int* p = new int;  
}
```

Nuestra implementación de `Vec<T>` tiene un *leak*.

¿Dónde?

(En breve lo arreglaremos).

# Problemas con punteros

## Otro problema con punteros: *dangling pointers*

- Una vez que hicimos `delete` de una dirección de memoria, no deberíamos acceder a su contenido.

```
int main() {  
    int* p = new int;  
    *p = 42;  
    delete p;  
    cout << *p << endl;  
}
```

# Destructores (motivación)

- ▶ Cuando termina el *scope* de una variable local  $x$  de tipo  $T$ , esa memoria se recupera automáticamente.
- ▶ ¿Qué pasa si  $x$  tiene internamente punteros a estructuras que están almacenadas en el *heap*?

Por ejemplo:

```
int main() {  
    Vec<int> v;  
    v.push_back(1);  
}
```

# Destrucción (motivación)

- ▶ Cuando termina el *scope* de una variable local  $x$  de tipo  $T$ , esa memoria se recupera automáticamente.
- ▶ ¿Qué pasa si  $x$  tiene internamente punteros a estructuras que están almacenadas en el *heap*?

Por ejemplo:

```
int main() {  
    Vec<int> v;  
    v.push_back(1);  
}
```

- ▶ **Problema:** Finaliza el scope de  $v$  pero nunca se hizo `delete[]` del arreglo privado `v._valores`.

# Destruyores

- ▶ Cada vez que se libera la memoria de un objeto de tipo T, C++ invoca implícitamente al **destructor** del tipo T.
- ▶ El destructor de una clase T se llama `T::~~T()`.
- ▶ El programador nunca debe llamar explícitamente al destructor.

```
template<class T>
class Vec {
    ...
public:
    ~Vec();
};

template<class T>
Vec<T>::~~Vec() {
    delete[] _valores;
}
```

# Referencias

## Otra forma de usar punteros: referencias

- ▶ Una variable local o parámetro se puede declarar como una referencia a un valor de tipo T, dándole tipo T&.
- ▶ Una referencia es un puntero pero que debe ser inicializado y no puede cambiar durante su ciclo de vida .

## Ejemplo: ambas funciones hacen lo mismo

```
int main() {  
    int a = 41;  
    int& b = a;  
    b = b + 1;  
    cout << a << endl;  
}
```

```
int main() {  
    int a = 41;  
    int* b = &a;  
    *b = *b + 1;  
    cout << a << endl;  
}
```

# Referencias

## Pasaje de parámetros por referencia

```
void f(int& x, int y) {  
    x++;  
    y++;  
}
```

```
int main() {  
    int a = 1;  
    int b = 1;  
    f(a, b);  
    cout << a << endl;  
    cout << b << endl;  
}
```



# Referencias

## Devolución de resultados por referencia

```
template<class T>
class Vec { ...
public:
    T& operator[](int i) ;
};

template<class T>
T& Vec<T>::operator[](int i) {
    return _valores[i];
}

int main() {
    Vec v;
    v.push_back(1);
    v[0] = 10;
    cout << v[0] << endl;
}
```

# Referencias `const` (motivación)

Consideremos la función que recibe un vector y suma sus primeros dos elementos:

```
int sumaPrimeros(vector<int> v) {  
    return v[0] + v[1];  
}
```

**Problema:** el parámetro se pasa por copia. Esto es extremadamente ineficiente.

## Referencias `const` (motivación)

Podemos arreglar el problema de eficiencia si recibimos el vector por referencia:

```
int sumaPrimeros(vector<int>& v) {  
    return v[0] + v[1];  
}
```

**Nuevo problema:** no hay ninguna garantía de que la función no modifique su parámetro.

## Referencias `const`

El tipo `const T&` representa una referencia **immutable** a un valor de tipo `T`:

```
int sumaPrimeros(const vector<int>& v) {  
    return v[0] + v[1];  
}
```

## Referencias const

Tenemos un conjunto implementado sobre un arreglo sin repetidos:

```
template<class T>
class Conj {
public:
    void agregar(const T& x);
    bool pertenece(const T& x) const;
private:
    vector<T> _elementos;
};
```

## Referencias `const`

¿Cómo agregamos un método para obtener un vector con todos los elementos del conjunto? Comparar las siguientes cuatro opciones:

1. `vector<T>& Conj<T>::elementos()`
2. `vector<T> Conj<T>::elementos() const`
3. `vector<T>& Conj<T>::elementos() const`
4. `const vector<T>& Conj<T>::elementos() const`

1. Doy una referencia modificable de elementos xxx.
2. Doy una copia modificable.
3. Intenta dar una referencia modificable, pero no compila porque el metodo es const.
4. Doy una referencia no modificable.

## Preguntas

- ▶ ¿Qué pasa si termina el *scope* del conjunto y queremos usar sus elementos?
- ▶ ¿Qué pasa si el usuario modifica el vector de elementos?

# Testing

¿Cómo comprobamos que la implementación no tiene problemas de memoria?

- ▶ *Leaks.*
- ▶ *Dangling pointers.*
- ▶ Doble `delete`.
- ▶ Desreferencia de `NULL` (`*NULL`).

Es un problema difícil en general.

- ▶ En algunos lenguajes modernos (ej. rust) el compilador puede garantizar, a través del sistema de tipos, que el programa usa la memoria de manera segura.
- ▶ En C++ tenemos que hacer *testing*. Usaremos la herramienta `valgrind`:

```
valgrind --leak-check=full ./programa
```

# Leak

## lib.cpp

```
1  int* crear() {  
2      int* p = new int;  
3      *p = 42;  
4      return p;  
5  }
```

## tests.cpp

```
6  TEST(punteros, leak) {  
7      int* ps[3];  
8      ps[0] = crear();  
9      ps[1] = crear();  
10     ps[2] = crear();  
11  
12     delete ps[0];  
13     delete ps[1];  
14 }
```



# Leak

```
==3290== HEAP SUMMARY:
==3290==    in use at exit: 72,708 bytes in 2 blocks
==3290==    total heap usage: 159 allocs, 157 frees, 108,516 bytes allocated
==3290==
==3290== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==3290==    at 0x4C2E0EF: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux)
==3290==    by 0x4055F7: crear() (lib.cpp:2)
==3290==    by 0x405681: punteros_leak_Test::TestBody() (tests.cpp:10)
==3290==    by 0x4366F2: void testing::internal::HandleSehExceptionsInMethodIfSupported<testing::Test, void>
==3290==    by 0x42FA00: void testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>
==3290==    by 0x40DB67: testing::Test::Run() (gtest-all.cc:3974)
==3290==    by 0x40E520: testing::TestInfo::Run() (gtest-all.cc:4149)
==3290==    by 0x40EC12: testing::TestCase::Run() (gtest-all.cc:4267)
==3290==    by 0x419DFF: testing::internal::UnitTestImpl::RunAllTests() (gtest-all.cc:6633)
==3290==    by 0x437CCA: bool testing::internal::HandleSehExceptionsInMethodIfSupported<testing::internal:
==3290==    by 0x430866: bool testing::internal::HandleExceptionsInMethodIfSupported<testing::internal::Un
==3290==    by 0x41873B: testing::UnitTest::Run() (gtest-all.cc:6242)
==3290==
==3290== LEAK SUMMARY:
==3290==    definitely lost: 4 bytes in 1 blocks
==3290==    indirectly lost: 0 bytes in 0 blocks
==3290==    possibly lost: 0 bytes in 0 blocks
==3290==    still reachable: 72,704 bytes in 1 blocks
==3290==    suppressed: 0 bytes in 0 blocks
==3290== Reachable blocks (those to which a pointer was found) are not shown.
==3290== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==3290==
==3290== For counts of detected and suppressed errors, rerun with: -v
==3290== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

# Invalid read

## lib.cpp

```
7  int incrementar(int* p) {  
8      return *p + 10;  
9  }
```

## tests.cpp

```
20  TEST(punteros, invalid_read) {  
21      int* x = crear();  
22      delete x;  
23      int y = incrementar(x);  
24  }
```

# Invalid read

```
==3378== Invalid read of size 4
==3378==    at 0x405618: incrementar(int*) (lib.cpp:8)
==3378==    by 0x40567C: punteros_invalid_read_Test::TestBody() (tests.cpp:23)
==3378==    by 0x4366C0: void
↳ testing::internal::HandleSehExceptionsInMethodIfSupported<testing::Test,
↳ void>(testing::Test*, void (testing::Test::*)(), char const*) (gtest-all.cc:3899)
==3378==    by 0x42F9CE: void
↳ testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>(testing::Test*,
↳ void (testing::Test::*)(), char const*) (gtest-all.cc:3935)
==3378==    by 0x40DB35: testing::Test::Run() (gtest-all.cc:3974)
==3378==    by 0x40E4EE: testing::TestInfo::Run() (gtest-all.cc:4149)
```

...

```
==3378== HEAP SUMMARY:
==3378==    in use at exit: 72,704 bytes in 1 blocks
==3378==    total heap usage: 159 allocs, 158 frees, 108,570 bytes allocated
==3378==
==3378== LEAK SUMMARY:
==3378==    definitely lost: 0 bytes in 0 blocks
==3378==    indirectly lost: 0 bytes in 0 blocks
==3378==    possibly lost: 0 bytes in 0 blocks
==3378==    still reachable: 72,704 bytes in 1 blocks
==3378==    suppressed: 0 bytes in 0 blocks
==3378== Reachable blocks (those to which a pointer was found) are not shown.
==3378== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==3378==
==3378== For counts of detected and suppressed errors, rerun with: -v
==3378== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

# Invalid write

## lib.cpp

```
11 void sobrecribir(int* p) {  
12     *p = 20;  
13 }
```

## tests.cpp

```
30 TEST(punteros, invalid_write) {  
31     int* x = crear();  
32     delete x;  
33     sobrecribir(x);  
34 }
```

# Invalid write

```
==3463== Invalid write of size 4
==3463==    at 0x40562B: sobreescribir(int*) (lib.cpp:12)
==3463==    by 0x40567C: punteros_invalid_write_Test::TestBody() (tests.cpp:33)
==3463==    by 0x4366BC: void
↳ testing::internal::HandleSehExceptionsInMethodIfSupported<testing::Test,
↳ void>(testing::Test*, void (testing::Test::*)(), char const*) (gtest-all.cc:3899)
==3463==    by 0x42F9CA: void
↳ testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>(testing::Test*,
↳ void (testing::Test::*)(), char const*) (gtest-all.cc:3935)
==3463==    by 0x40DB31: testing::Test::Run() (gtest-all.cc:3974)
==3463==    by 0x40E4EA: testing::TestInfo::Run() (gtest-all.cc:4149)

...

==3378== HEAP SUMMARY:
==3378==    in use at exit: 72,704 bytes in 1 blocks
==3378==    total heap usage: 159 allocs, 158 frees, 108,570 bytes allocated
==3378==
==3378== LEAK SUMMARY:
==3378==    definitely lost: 0 bytes in 0 blocks
==3378==    indirectly lost: 0 bytes in 0 blocks
==3378==    possibly lost: 0 bytes in 0 blocks
==3378==    still reachable: 72,704 bytes in 1 blocks
==3378==    suppressed: 0 bytes in 0 blocks
==3378== Reachable blocks (those to which a pointer was found) are not shown.
==3378== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==3378==
==3378== For counts of detected and suppressed errors, rerun with: -v
==3378== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

# Double free

## lib.cpp

```
15 void limpiar(int* p) {  
16     delete p;  
17 }
```

## tests.cpp

```
40 TEST(punteros, double_free) {  
41     int* x = crear();  
42     limpiar(x);  
43     limpiar(x);  
44 }
```

# Double Free

```
==3525== Invalid free() / delete / delete[] / realloc()
==3525==    at 0x4C2F24B: operator delete(void*) (in
↳ /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3525==    by 0x40564B: limpiar(int*) (lib.cpp:16)
==3525==    by 0x40567C: punteros_double_free_Test::TestBody() (tests.cpp:43)
==3525==    by 0x4366BC: void
↳ testing::internal::HandleSehExceptionsInMethodIfSupported<testing::Test,
↳ void>(testing::Test*, void (testing::Test::*)(), char const*) (gtest-all.cc:3899)
==3525==    by 0x42F9CA: void
↳ testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>(testing::Test*,
↳ void (testing::Test::*)(), char const*) (gtest-all.cc:3935)
==3525==    by 0x40DB31: testing::Test::Run() (gtest-all.cc:3974)

...

==3525== HEAP SUMMARY:
==3525==    in use at exit: 72,704 bytes in 1 blocks
==3525==    total heap usage: 159 allocs, 159 frees, 108,570 bytes allocated
==3525==
==3525== LEAK SUMMARY:
==3525==    definitely lost: 0 bytes in 0 blocks
==3525==    indirectly lost: 0 bytes in 0 blocks
==3525==    possibly lost: 0 bytes in 0 blocks
==3525==    still reachable: 72,704 bytes in 1 blocks
==3525==    suppressed: 0 bytes in 0 blocks
==3525== Reachable blocks (those to which a pointer was found) are not shown.
==3525== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==3525==
==3525== For counts of detected and suppressed errors, rerun with: -v
==3525== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```