Memoria dinámica

Algoritmos y Estructuras de Datos II

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;
  }
}
```

Memoria dinámica: motivación

Queremos implementar una versión simplificada de `std::vector`[1]:

```
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:
    ...
};
```

[1] ...sin usar `std::vector`.

Memoria dinámica: motivación

¿Qué representación elegimos?

- No alcanza con un arreglo estático.
- 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

```
struct Par {
  int x;
  char y;
};

int main() {
  Par pares[2];
  pares[0].x = 10;
  pares[0].y = 'A';
  pares[1].x = 20;
  pares[1].y = 'B';
  ...
}
```

	Dirección	Byte
	...	...
pares[0].x:0	9000	10
pares[0].x:1	9001	0
pares[0].x:2	9002	0
pares[0].x:3	9003	0
pares[0].y	9004	65
pares[1].x:0	9005	20
pares[1].x:1	9006	0
pares[1].x:2	9007	0
pares[1].x:3	9008	0
pares[1].y	9009	66
	...	...

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`.

- `int*`
- `char*`
- `vector<int>*`
- `vector<int*>`
- `int**`
- ...

Operaciones con punteros

- Dirección de memoria de una variable. (`&variable`)

  si `variable` es de tipo `T`
  `&variable` es de tipo `T*`

- Valor almacenado en una dirección de memoria. (`*puntero`)

  si `puntero` es de tipo `T*`
  `*puntero` es de tipo `T`

Punteros

Punteros a variables locales

```
int main() {
  int x = 10;
  int* p = &x;
  cout << p << endl;
  cout << *p << endl;
  *p = *p + 1;
  cout << x << endl;

  int* q = &7;
}
```

Punteros

Punteros a variables locales

```
int main() {
  int x = 10;
  int* p = &x;
  cout << p << endl;
  cout << *p << endl;
  *p = *p + 1;
  cout << x << endl;

  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];
  (*p).x = 10;
  p->y = 'b';
  cout << p->x << endl;
  char* q = &p->y;
  *q = 'c';
  cout << pares[1].y << endl;
}
```

Punteros `NULL`

La dirección de memoria 0 está reservada para representar un puntero que no referencia ningún valor en particular.

En C++ se escribe `NULL` para el puntero a la dirección 0.

Regiones de memoria

La memoria en C++ se divide en dos regiones:

La pila

La memoria en la pila se administra automáticamente.

El *heap*

La memoria en el *heap* se administra manualmente.

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

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() {
  int* p = new int;
  *p = 42;
  return p;
}

int main() {
  int* q = f();
  cout << *q << endl; // OK
  delete q;
}
```

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>

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

- 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);
}
```

Destructores

- 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`.

Destructores

- 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 "maquillado".

```
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) const;
};

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

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

Referencias `const`

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`

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 **inmutable** 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 tres opciones:

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

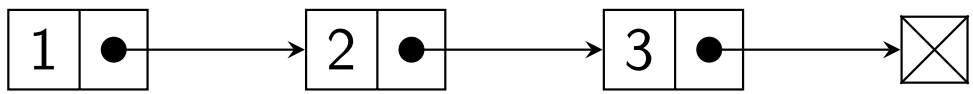
Preguntas

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

Listas simplemente enlazadas

Una *lista simplemente enlazada* es una estructura que sirve para representar secuencias.

Gráficamente



Listas simplemente enlazadas

Implementemos la clase Secuencia<T> sobre una lista simplemente enlazada, con los siguientes métodos:

```
Secuencia<T>::Secuencia();

void Secuencia<T>::agregarAdelante(const T& x);

int Secuencia<T>::longitud() const;
const T& Secuencia<T>::iesimo(int i) const;

friend ostream& operator<<(ostream&, const Secuencia<T>&);

void Secuencia<T>::agregarAtras(const T& x);
void Secuencia<T>::sacarPrimero();
void Secuencia<T>::sacarUltimo();

Secuencia<T>::~Secuencia();

Secuencia<T>::Secuencia(const Secuencia<T>& o);
Secuencia<T>& Secuencia<T>::operator=(const Secuencia<T>& o);
```

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
```