Programación sobre redes

Labo 2: Memoria Dinámica (Parte I)

Basado en clase Algoritmos II dc-fcen

May 22, 2020

Instituto Industrial Luis A. Huergo

Contenidos

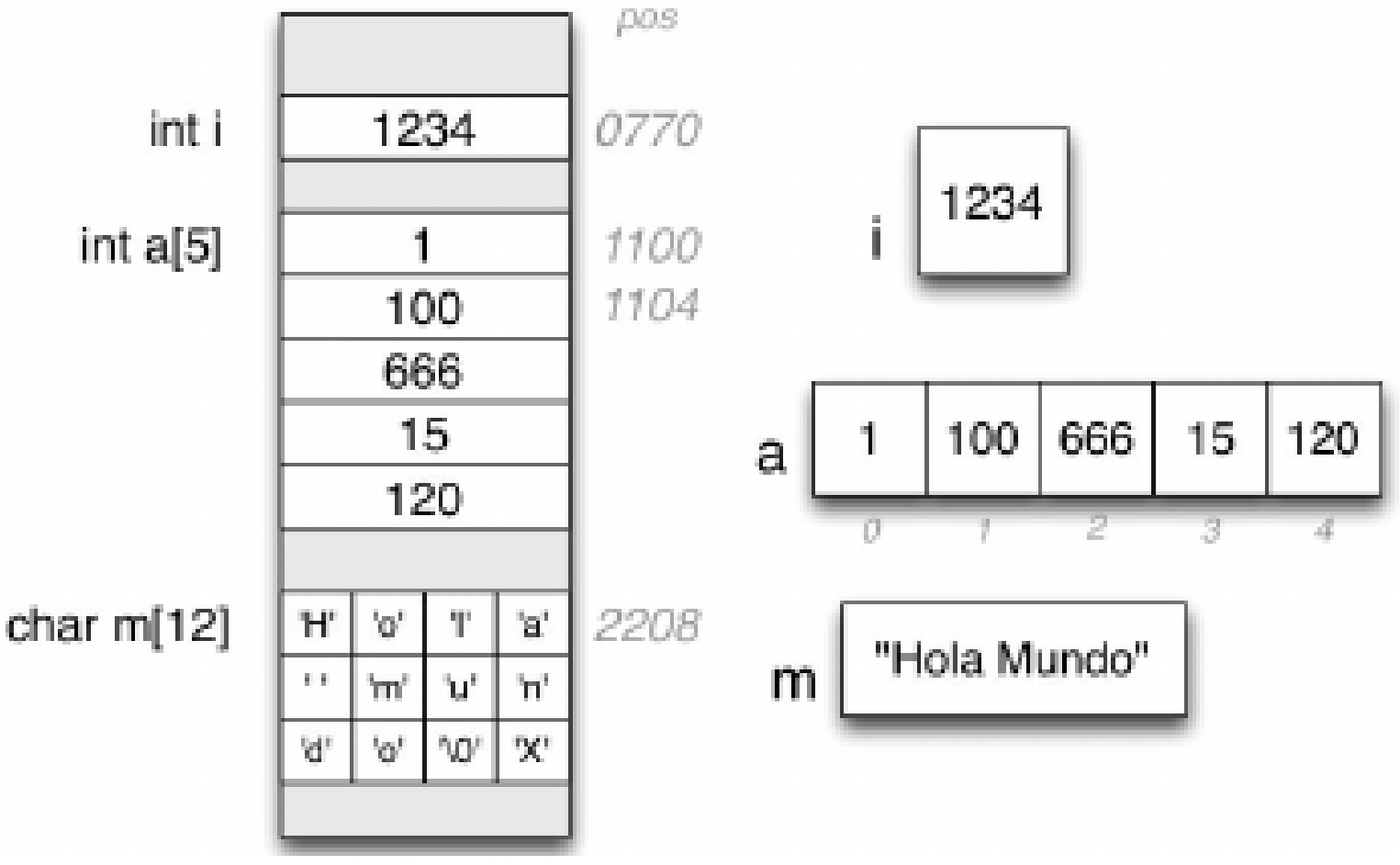
1. Modelo de memoria
2. Variables
3. Memoria dinámica
4. Punteros

Modelo de memoria

Modelo de memoria en C++

- Básicamente, podemos pensarla como un arreglo de bytes. En general, a cada uno de ellos le corresponderá una dirección
- Cada valor ”guardado” en una variable, ocupa una cierta cantidad de posiciones en el arreglo según su tamaño.
- El tamaño depende del tipo de la variable, del compilador, y de la arquitectura.

Modelo de memoria en C++



Variables

Nuestras variables

Hasta ahora, las variables que utilizabamos:

- Tenían un scope (alcance) que podía ser local o bien global
- Eran automáticas, ya que dejaban de existir cuando se abandona el scope donde fueron creadas
- ¿Podemos tener variables locales pero no automáticas? ¿Qué significa esto?

```
int main(){

    int a  = 2;
    for(int i =0;i<10;i++)
        a = 5;

    cout << a ;

    return 0;
}
```

Figure 1: ¿Qué valor se imprime?

Variables automáticas en C

- El espacio asignado para estas variables es conocido en tiempo de compilación, a partir de su tipo
- Este espacio permanece ”activo” mientras se encuentran en su scope

Pregunta: Bajo estas condiciones, ¿se podría crear un arreglo de tamaño variable? Es decir, uno cuyo tamaño pueda decidirse en tiempo de ejecución

Propuesta para lo anterior

```
int main(){
    int n;
    cin >> n;
    int arreglo[n];
    return 0;
}
```

Figure 2: partir del input de usuario, creo un arreglo de ese tamaño

Figure 3: ¿Funciona?

Memoria dinámica

Memoria dinámica

- Utilizando una función (en C, **malloc**), un proceso puede pedir cierta cantidad de memoria al sistema operativo.
- De la misma forma, cuando ese bloque de memoria deje de ser necesario, se puede invocar otra función para ”liberarlo” (en C, **free**).
- Cuando se pide memoria, la misma se aloja en el heap, que puede crecer dinámicamente (a diferencia del stack)

Reservando memoria

En C++, la biblioteca estándar, además, nos provee de **new** para pedir memoria.

- Por ejemplo, new int va a solicitar el espacio que ocupa un int en nuestra arquitectura (asumamos 4 bytes)
- ¿Donde va a alojarse ese espacio?
- Otro ejemplo:

```
int main(){
    int n;
    cin >> n;
    int * areglo = new int[n];
    return 0;
}
```

Figure 4: De esta forma tenemos la noción de arreglo de tamaño variable. ¡Ojo! La variable arreglo es local pero no **automática** ¿Qué nos devuelve el pedido de memoria?

Caso de estudio: Vector

- Representan **arreglos de tamaño dinámico**.
- Se imaginarán que en el fondo debería existir una noción de pedir memoria dinámicamente
- Como los arreglos, garantizan un rápido acceso a cualquiera de sus elementos, así también como las operaciones de agregar elementos al final o al principio.

Intuición de Vector: Creación

Supongamos que dentro del main hago:

```
VECTOR<INT> VECT1;
```

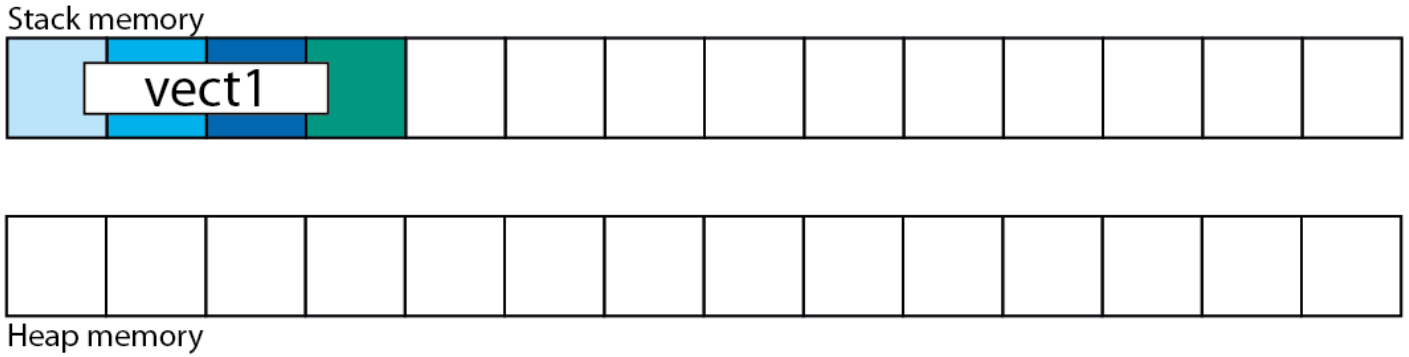


Figure 5: Parte de la representación interna del vector reside en stack. Todavía no hay elementos en el. Créditos por la imagen aquí

Intuición de Vector: Primera inserción

Supongamos que luego hago un push_back :

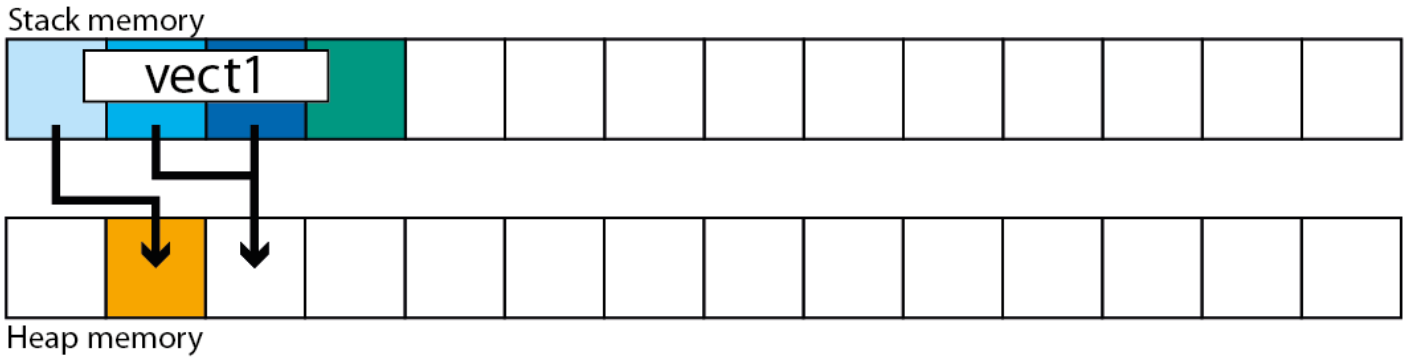


Figure 6: El primer elemento es agregado. Notar que esto ocurre en memoria dinámica. Créditos por la imagen aquí

Intuición de Vector: Próximas inserciones

¿Qué pasa si agrego un nuevo elemento?

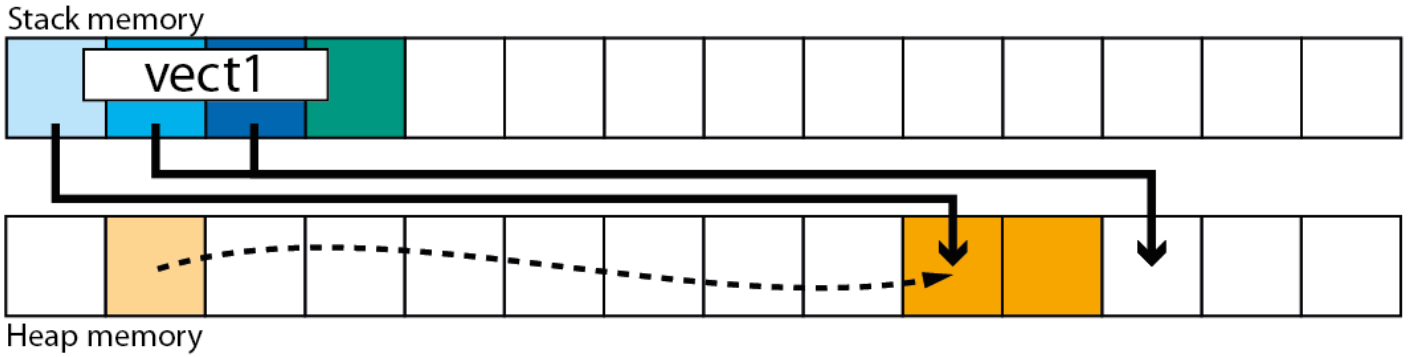


Figure 7: Cuando el tamaño es insuficiente, se pide un nuevo espacio de memoria en el heap para almacenar el doble, por ejemplo, del tamaño anterior. Créditos por la imagen aquí

Punteros

Punteros

- Una **variable** que guarda la dirección de memoria de un valor cualquiera, es llamada puntero.
- Se dice que los punteros ”apuntan” al valor cuya dirección almacenan.
- Son variables. Por lo tanto tienen un tipo y ocupan lugar en memoria.
- En general, van a ser **automáticas** (aunque el objeto apuntado no necesariamente)

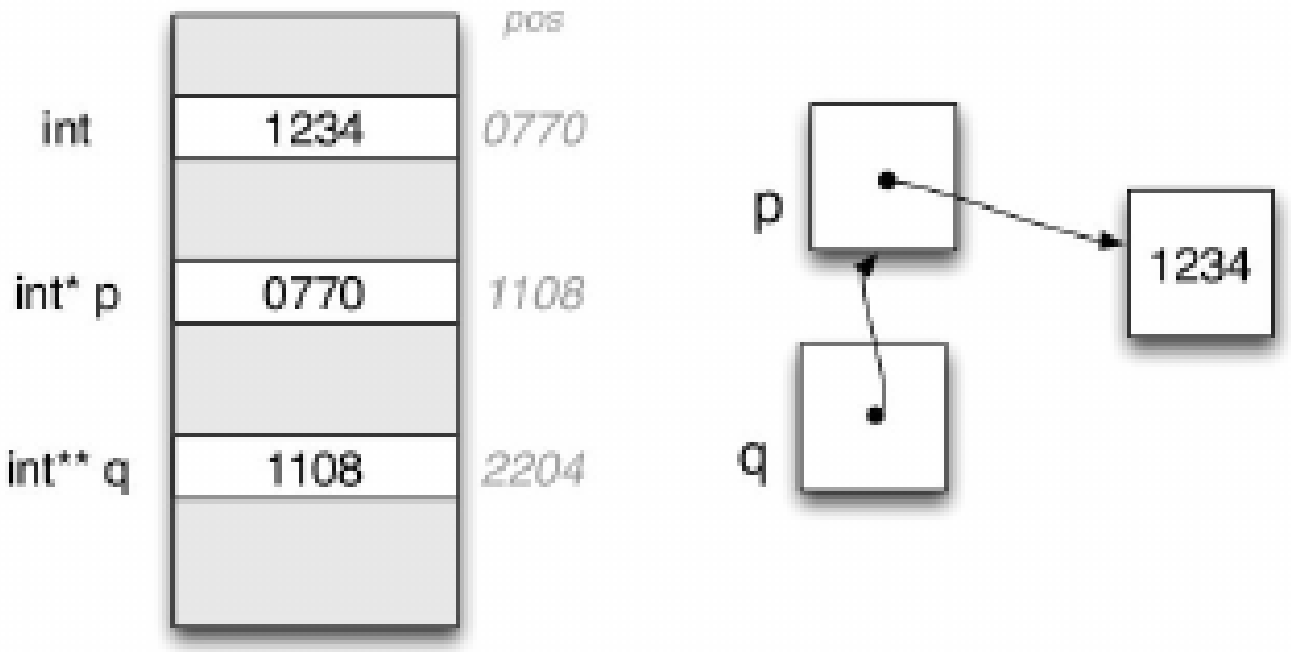
Ejemplo

```
int main(){
    int x = 5;
    int *y = &x;
    *y = 10;
    cout << x;
    return 0;
}
```

Analizando el ejemplo

- La variable y es un puntero. El mismo apunta a la dirección de memoria asociada a x.
- La dirección de memoria de una variable se obtiene utilizando el operador &
- El valor de la variable apuntada por un puntero se obtiene utilizando el operador *
- Todas las variables anteriores son locales y automáticas

Punteros en memoria



Punteros vs Referencias

- No es posible tener referencias NULL.
- Una referencia está ligada a un objeto y no puede cambiar para referenciar a otro.
- Una referencia debe ser inicializada en el momento de su creación.
- Trabajar con referencias puede resultar más sencillo

Liberando memoria

¿Qué ocurre con los bloques de memoria una vez que dejamos de usarlos?

- En C++, no se hace nada. El bloque seguirá reservado para nuestro proceso aún cuando haya terminado su ejecución

Liberando memoria

- Es responsabilidad del programador encargarse de librerar la memoria reservada por el programa.
- En C++, para liberar la memoria pedida por new, existe la función **delete** (esto no elimina el puntero, sino el objeto apuntado)
- Es importante evitar memory leaks

Garbage Collector

En otros lenguajes como Python o Java, existe un mecanismo conocido como **garbage collector**, que se encarga de períodicamente liberar la memoria de objetos que ya no son referenciados por procesos.

Memory leaks

Ejercicio 0

Para bajar un poco a tierra lo visto anteriormente, hagamos un ejercicio introductorio.

1. Crear un programa que pida un entero dinámicamente y luego modifique su valor a 5. ¡No olvidar **liberar la memoria pedida!**
2. Crear un función void incrementar que reciba un puntero a entero como parámetro y le sume uno al valor
3. Modificar 1 para que luego de asignarle el valor al entero, se utilice la función anterior para sumarle uno. Luego, imprimir el valor (desde main) para corroborar el resultado
4. ¿El comportamiento del programa sería el mismo en caso de haber creado el entero en el stack(como variable automática)?

Ejercicio 1

Determinar los valores de *p, q, *r, v and *s

```
1. int v = 8, *r, *s;
2. int *p;
3. int q = 100;
4. p = &q;
5. r = p;
6. *p = 20;
7. p = new int;
8. *r = 30;
9. q = v;
10. s = p;
11. *s = 50;
```

Spoiler: Obtenido de aquí (con solución)

Ejercicio 2

Determinar los valores de *p, *q, v y nom

```
1. int *p , *q , v , nom[5];
2. p = &v;
3. *p = 12;
4. q = p;
5. nom[0] = *q;
6. p = nom;
7. p++;
8. nom[2] = 12;
9. *p = 13;
10. *q = 10;
11. v = 11;
12. *(p+3) = 16;
13. p = &nom[3];
14. *p = 10;
15. p--;
```