Punteros

Como ya vimos en la lección de las funciones (parámetros por copia y por referencia), las variables tienen dos “elementos”.

* Dirección de memoria en la que se almacena el valor
* El valor en sí

En c++ podemos trabajar con ambos elementos por separado declarando la variable como puntero. ¿Cómo declaramos un puntero?

|  |
| --- |
| int\* a; |

En este caso hemos declarado una variable *a* tipo puntero a entero, ¿qué quiere decir esto? que a apuntará a un entero. Como no le hemos dado ningún valor, inicialmente no está apuntando a nada. Fíjate en el siguiente código.

|  |
| --- |
| int c = 32;int\* a = &c; |



Ahora el puntero *a* toma como valor la dirección de *c*. Si hicieramos

|  |  |
| --- | --- |
| 1234 | int c = 32;int\* a = &c;cout << a; |

Contrariamente a lo que se podría esperar, no se imprimiría por pantalla 32, **porque la variable a es una dirección de memoria.** Entonces, ¿cómo accedemos al valor al que apunta *a*? A través del operador indirección.

|  |  |
| --- | --- |
| 1234 | int c = 32;int\* a = &c;cout << \*a; //Muestra por pantalla 32. |

Pero, ¿cuál es la gracia de los punteros?, fíjate en el siguiente ejemplo de código:

|  |  |
| --- | --- |
| 12345678 | int c = 32;int\* a = &c;cout << \*a; // Muestra por pantalla 32.c = 25;cout << \*a; // Muestra por pantalla 25.\*a = 16;cout << c; // Muestra por pantalla 16. |

Del mismo modo, a un puntero le podriamos asignar otro puntero.

|  |  |
| --- | --- |
| 1234567891011121314 | int c = 32;int\* a = &c;int\* b = a; //a y b apuntan al valor de c cout << \*a; // Muestra por pantalla 32.c = 25;cout << \*a; // Muestra por pantalla 25.\*a = 16;cout << c; // Muestra por pantalla 16.\*b = 22;cout << c; // Muestra por pantalla 22.cout << \*a; // Muestra por pantalla 22.cout << \*b; // Muestra por pantalla 22. |

Inicializando un puntero

En el ejemplo anterior hemos visto que a un puntero le podemos asignar la dirección de otra variable del mismo tipo. Pero también podemos crear un espacio *nuevo de memoria*.

|  |  |
| --- | --- |
| 123 | float \*a = new float{3.764};  cout << \*a ; // muestra por pantalla 3.764 |

En este ejemplo creamos un puntero a, que apuntará a un número decimal, y además reservamos espacio de memoria para albergar ese número decimal. Inicializamos el puntero con el valor 3.764.



Podemos operar con el puntero del mismo modo que en los ejemplos anteriores:

|  |  |
| --- | --- |
| 12345678910 | float \*a = new float{3.764};  cout << \*a ; // muestra por pantalla 3.764float\* b = a; //a y b apuntan al valor de c\*b = 22;cout << \*a; // Muestra por pantalla 22.cout << \*b; // Muestra por pantalla 22. float c = \*a;cout << c; // Muestra por pantalla 22. |

Importancia de la memoria reservada

Cuando reservamos espacio en memoria para un puntero esa memoria permanece *ocupada* hasta que explicitamente le digamos al programa que ya no la necesitamos.

Un programa en el que se reserva continuamente memoria y no se libera corre el riesgo de quedarse sin memoria. Veamos qué pasa con el siguiente código.

|  |  |
| --- | --- |
| 1234 | float \*a;a = new float{3.4};a = new float{5.1}; // perdemos la referencia a la memoria anterior |

En la primera línea creamos un puntero a un número decimal. En la segunda línea reservamos espacio en memoria y le asignamos el valor 3.4.



El problema viene en la tercera línea, volvermos a reservar espacio, y debido a ello perdemos la referencia a 3.4.



Además, ese espacio de memoria está reservado, por lo que no puede volver a utilizarse, y si esto sucede muy a menudo nuestro programa consumirá toda la memoria disponible del ordenador.

¿Qué pasaría en el siguiente programa?

|  |  |
| --- | --- |
| 12345678 | float \*a;float \*b;a = new float{3.4};b = a;a = new float{5.1}; // perdemos la referencia a la memoria anteriorcout << \* b; // muestra por pantalla 3.4cout << \* a; // muestra por pantalla 5.1 |

Liberando la memoria reservada

El comando *new* reserva el espacio de memoria, el comando *delete* lo libera.

|  |  |
| --- | --- |
| 12345678 | float \*a;a = new float{3.4};cout << \* a; // muestra por pantalla 3.4delete a; // borramos el espacio reservado para a cout << \* a; // ERROR de ejecucion, a no apunta a ningun sitioa = new float{5.1};cout << \* a; // muestra por pantalla 5.1 |

Primero reservamos espacio para un número decimal, asignándole el valor 3.4. Cuando hacemos delete liberamos ese espacio de memoria, y por lo tanto **a no está apuntando en ese momento a ningún sitio.**. Por ello si queremos acceder a su valor con \*a nuestro programa dejará de funcionar (error de ejecución).



Esto puede ocasionar un error muy típico en los programas: **liberar un espacio de memoria al que apunta otro puntero**

|  |  |
| --- | --- |
| 123456789 | float \*a;a = new float{3.4};float \*b;b = a;cout << \* a; // muestra por pantalla 3.4cout << \* b; // muestra por pantalla 3.4delete a; // borramos el espacio reservado para a cout << \* b; // ERROR. Mierda, hemos eliminado el espacio de memoria al que apunta b. |



Listas Enlazadas Simples – Añadir Elementos

Una lista enlazada simple es una estructura de datos en los que nada elemento de la lista contiene datos y un puntero al siguiente elemento de la lista.
Las listas enlazadas simples:

* Tienen n elementos
* Los elementos están ordenados, es decir, cada elemento tiene una posición determinada
* Cada elemento sabe qué elemento viene a continuación (pero no cuál está antes)



*Lista Enlazada Simple*

Las listas se pueden programar de maneras muy distintas en C++. En este curso utilizaremos clases para los datos, punteros y funciones.

**Node.h**

|  |  |
| --- | --- |
| 1234567 | class Node{public:    Node(int \_dato = 0);private:    int dato;    Node\* next;}; |

**Node.cpp**

|  |  |
| --- | --- |
| 12345678 | #include "Node.h" Node::Node(int \_dato):    dato{\_dato},    next{nullptr}{     } |

Añadiendo elementos al final de una lista enlazada – push\_back

La función más básica para trabajar con listas enlazadas simples es la de añadir elementos al final de la misma. Esta función puede tener cualquier nombre, nosotros la llamaremos push\_back.

**Node.h**

|  |  |
| --- | --- |
| 12345678910 | using namespace std; class Node{public:    Node(int \_dato = 0);    void push\_back(int value); /// Añade el elemento value al final de la listaprivate:    int dato;    Node\* next;}; |

**Node.cpp**

|  |  |
| --- | --- |
| 12345678910111213141516 | #include "Node.h"Node::Node(int \_dato):    dato{\_dato},    next{nullptr}{ } void Node::push\_back(int value){    //si hay un elemento a continuacion    if(next){        next->push\_back(value);    }else{ // soy el ultimo elemento        next = new Node(value);    }} |

Un ejemplo de programa principal sería

|  |  |
| --- | --- |
| 123456789 | #include "Node.h" using namespace std; int main(){    Node\* lista = new Node();    lista->push\_back(3);    lista->push\_back(5);} |

De la función *push\_back* hay que destacar el comportamiento distinto dependiendo de si el elemento es el último de la lista o no.

* **El elemento es el último de la lista**. Sabemos que el elemento es el último de la lista porque *next == nullptr*. En ese caso, creamos un nuevo elemento y hacemos que *next* apunte a él
* **El elemento no es el último de la lista *if(next)***. En ese caso probamos a añadir el elemento en el siguiente elemento. Esto hará que probemos siempre con el siguiente (*next*) hasta que lleguemos al último, que será cuando se añadirá.

Vamos a ver qué pasa en cada línea de este programa



*main.cpp – línea 7*



*main.cpp – linea 8*



*main.cpp – línea 9*

**Es importante notar** que el primer elemento de la lista sólo tiene como objetivo enlazar con la lista, y no se almacena ningún valor. Por ello, una lista con dos valores (3 y 5 en el ejemplo) tendrá 3 elementos.

# Listas Enlazadas Simples: Buscar elementos

Vamos a ver el modo más sencillo de buscar elementos, que consiste en recorrer toda la lista hasta encontrar el primer elemento cuyo criterio coincide con el de búsqueda. Más adelante veremos algoritmos mejor optimizados para la búsqueda de elementos.

**Node.h**

|  |  |
| --- | --- |
| 12345678910 | class Node{public:    Node(int \_dato = 0);    void push\_back(int value);    Node\* search(int value) const;    int getDato() const;private:    int dato;    Node\* next;}; |

**Node.cpp**

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627282930313233343536 | #include "Node.h" Node::Node(int \_dato):    dato{\_dato},    next{nullptr}{ } void Node::push\_back(int value){    //si hay un elemento a continuacion    if(next){        next->push\_back(value);    }else{ // soy el ultimo elemento        next = new Node(value);    }} int Node::getDato() const{    return dato;} Node \*Node::search(int value) const{    // si hay un elemento a continuacion    if(next){        // si el siguiente es el dato buscado, lo devuelve        if(next->getDato() == value) return next;        //en caso contrario sigue buscando        else return next->search(value);    }else{        // si he llegado al ultimo sin encontrarlo devuelvo null        return nullptr;    }} |

La funcion search va buscando iterativamente, empezando por **el segundo elemento** de la lista (el primer elemento solo lo usamos como puntero inicial, pero no almacena ningún dato) hasta encontrar el valor deseado. Si lo encuentra devuelve el puntero al elemento en cuestión. Si llega al final sin econtrarlo devuelve nullptr.

Un ejemplo de main que realiza una búsqueda sería:

|  |  |
| --- | --- |
| 123456789101112 | int main(){    Node\* lista = new Node();         lista->push\_back(3);    lista->push\_back(5);    Node \* result = lista->search(3);    if(result) cout << result->getDato() << endl;    Node \* result2 = lista->search(7); // result2 seria nullptr    if(!result2) cout << "No encontrado" << endl;         delete lista;} |

Listas Enlazadas Simples: Eliminar Elementos

La siguiente figura muestra el proceso de eliminación de un elemento de una lista.

*Eliminando elemento de una lista*

* Para poder eliminar el elemento hacemos uso de un puntero auxiliar, que nos guarda la referencia al elemento posterior al que se quiere eliminar.
* Eliminamos el elemento que deseamos borrar
* Utilizamos al referencia actual para actualizar el siguiente elemento

El código quedaría del siguiente modo

**Node.h**

|  |  |
| --- | --- |
| 123456789101112 | class Node{public:    Node(int \_dato = 0);    void push\_back(int value);    Node\* search(int value) const;    bool erase(int value);    Node\* getNext() const;    int getDato() const;private:    int dato;    Node\* next;}; |

**Node.cpp**

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657 | Node::Node(int \_dato):    dato{\_dato},    next{nullptr}{ } void Node::push\_back(int value){    //si hay un elemento a continuacion    if(next){        next->push\_back(value);    }else{ // soy el ultimo elemento        next = new Node(value);    }} int Node::getDato() const{    return dato;} Node \*Node::search(int value) const{    // si hay un elemento a continuacion    if(next){        // si el siguiente es el dato buscado, lo devuelve        if(next->getDato() == value) return next;        //en caso contrario sigue buscando        else return next->search(value);    }else{        // si he llegado al ultimo sin encontrarlo devuelvo null        return nullptr;    }} bool Node::erase(int value){    if(next){        // si el siguiente es el dato buscado, lo elimino        if(next->getDato() == value){            Node\* aux = next->getNext(); //guardo referencia del elemento posterior            delete next; // borro el elemento deseado (libero su espacio en memoria)            next = aux; // actualizo next para que apunte al elemento posterior            return true; // devuelvo verdadero        }        //en caso contrario sigue buscando        else return next->erase(value);    }else{        // si he llegado al ultimo sin encontrarlo devuelvo falso        return false;    }} Node \*Node::getNext() const{    return next;} |

La funcion *erase* va buscando iterativamente, empezando por **el segundo elemento** de la lista (el primer elemento solo lo usamos como puntero inicial, pero no almacena ningún dato) hasta encontrar el valor deseado. Si lo encuentra lo eliminar y devuelve verdadero. Si llega al final sin econtrarlo devuelve *false*.

Listas Enlazadas Simples: Acceder por índice.

El acceso a los nodos de una lista por índice se puede hacer de modos muy diversos. En este ejemplo lo vamos a hacer sin utilizar funciones recursivas. En su lugar, la función *at(i)* recorrerá todos los elementos de la lista secuencialmente, y devolverá el nodo i-esimo.



*at(i) access*

Según se muestra en la figura, vamos a realizar 2 funciones distintas:

* **refAt(i)**, que devuelve un puntero al nodo i-esimo
* **at(i)**

, que devuelve la referencia al dato del nodo i-esimo.

El código quedaría del siguiente modo:

**Node.h**

|  |  |
| --- | --- |
| 1234567891011121314151617 | class Node{public:    Node(int \_dato = 0);  // ...    int & at(unsigned int i);    Node\* refAt(unsigned int i) const;    void setIndex(unsigned short value);         Node \*getNext() const;    void setNext(Node \*value); private:    void print(int i = 1) const;    int dato;    Node\* next;    unsigned int size;}; |

**Node.cpp**

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105 | #include "Node.h" #include <iostream> using namespace  std; Node::Node(int \_dato):    dato{\_dato},    next{nullptr},    size{0}{ } void Node::push\_back(int value){    //si hay un elemento a continuacion    if(next){        next->push\_back(value);    }else{ // soy el ultimo elemento        next = new Node(value);    }     // Incremento el tamano    size++;} int & Node::getDato(){    return dato;} void Node::printAll() const{    if(next) next->print();    else cout << "Lista vacia" << endl;} int &Node::at(unsigned int i){    return refAt(i)->getDato();} Node \*Node::refAt(unsigned int i) const{    if(i == 0) throw string("out of bounds (starts with 1)");    if(i > size) throw string("Index is out of bounds");    Node \* aux = next;    unsigned short index = 0;    while(aux){        index++;        if(index == i) return aux;        aux = aux->next;    }    throw string("Unexpected error");} void Node::print(int i) const{    cout << i << ": " << dato << endl;    if(next) next->print(++i);    else cout << endl;} Node \*Node::getNext() const{    return next;} void Node::setNext(Node \*value){    next = value;}  Node \*Node::search(int value) const{    // si hay un elemento a continuacion    if(next){        // si el siguiente es el dato buscado, lo devuelve        if(next->getDato() == value) return next;        //en caso contrario sigue buscando        else return next->search(value);    }else{        // si he llegado al ultimo sin encontrarlo devuelvo null        return nullptr;    }} bool Node::erase(int value){    if(next){        // si el siguiente es el dato buscado, lo elimino        if(next->getDato() == value){            Node\* aux = next->next; //guardo referencia del elemento posterior            delete next; // borro el elemento deseado (libero su espacio en memoria)            next = aux; // actualizo next para que apunte al elemento posterior            return true; // devuelvo verdadero        }        //en caso contrario sigue buscando        else return next->erase(value);    }else{        // si he llegado al ultimo sin encontrarlo devuelvo falso        return false;    }} |

Puedes consultar el código completo de este programa [aquí](https://github.com/Nebrija-Programacion/Datos-Algoritmos/tree/master/ejemplos/lista_enlazada_simple)

Si quisieramos modificar un elemnto por índice podríamos hacerlo del siguiente modo *lista.at(i) = value*, esto es así gracias a que la función at(i) devuelve la referencia al valor (*int &*), y no una copia del valor.