

Memoria Dinámica en C++

Algoritmos y Estructuras de Datos II

DC-FCEyN-UBA

30 de Marzo de 2016

Repaso: ¿Qué es una variable?

- Matemática: una "etiqueta" que representa a un valor en una expresión: $f(y) = y^2 + 2y$
- Programación: Nombre simbólico dado a un *valor* residente en la memoria.

```
int x = 0;  
x += 10;      // equivale a x=x+10;  
x++;          // equivale a x=x+1;
```

- Nombre: "x"
- Tipo: **int**
- Valor: para este ejemplo, **11**

Repaso: ¿Qué es una variable?

- Matemática: una "etiqueta" que representa a un valor en una expresión: $f(y) = y^2 + 2y$
- Programación: Nombre simbólico dado a un *valor* residente en la memoria.

```
int x = 0;  
x += 10;           // equivale a x=x+10;  
x++;              // equivale a x=x+1;
```

- Nombre: "x"
- Tipo: **int**
- Valor: para este ejemplo, **11**
- Su ubicación en la memoria (?)

Tiempo de vida de una variable

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del `scope` que la declaró.

Tiempo de vida de una variable

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del `scope` que la declaró.

```
int h = 4;
for(int i=0;i<10;i++){
    h = 25;
}
cout << h << endl;
```

```
int h = 4;
for(int i=0;i<10;i++){
    int g = 75;
}
cout << g << endl; //que ocurre aca??
```

El compilador se encarga de usar la pila o *stack* que el SO provee para almacenar las variables locales (y los parámetros de las funciones).

Variables en el stack (1/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

a = ?
r = ?

Stack

Variables en el stack (2/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

cin >> a;

a = 3
r = ?

Stack

Variables en el stack (3/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

cuad(a)

x = 3 c = ?
a = 3 r = ?
Stack

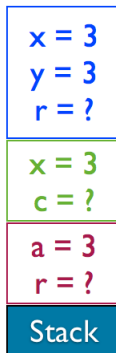
Variables en el stack (4/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

sum(x, x);



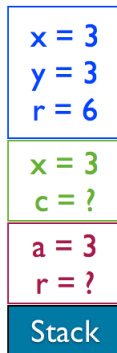
Variables en el stack (5/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

$r = x + y;$



Variables en el stack (6/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
return r;  
int c = ...
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

x = 3 c = 6
a = 3 r = ?
Stack

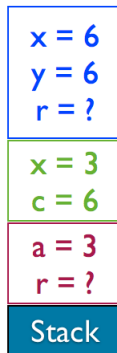
Variables en el stack (7/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

sum(c,c);



Variables en el stack (8/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

$r = x + y;$

$x = 6$
 $y = 6$
 $r = 12$

$x = 3$
 $c = 6$

$a = 3$
 $r = ?$

Stack

Variables en el stack (9/10)

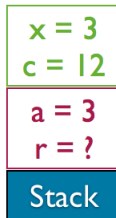
```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

return r;

c = ...



Variables en el stack (10/10)

```
int sum(int x, int y) {  
    int r = x + y;  
    return r;  
}
```

```
int cuad(int x) {  
    int c = sum(x, x);  
    c = sum(c, c);  
    return c;  
}
```

```
int main() {  
    int a;  
    cin >> a;  
    int r = cuad(a);  
    // ...  
    return 0;  
}
```

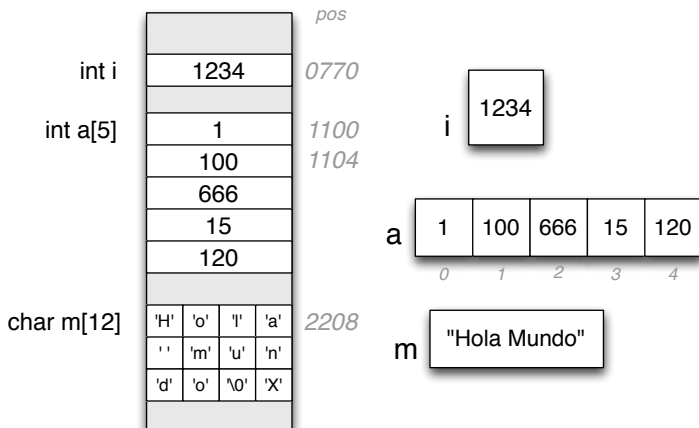
```
return c;  
r = ..
```

a = 3 r = 12

Stack

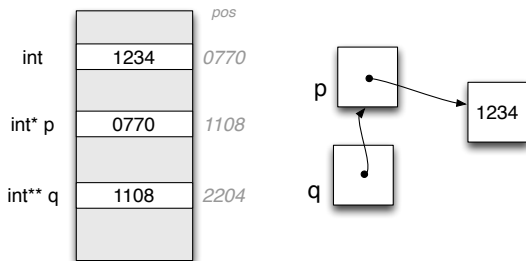
Modelo de memoria: Abstrayendo

- Para C++, la memoria es simplemente un *array de bytes*
- Cada variable ocupa una o más posiciones del array según su tamaño
- El tamaño depende del tipo, del compilador, y de la arquitectura



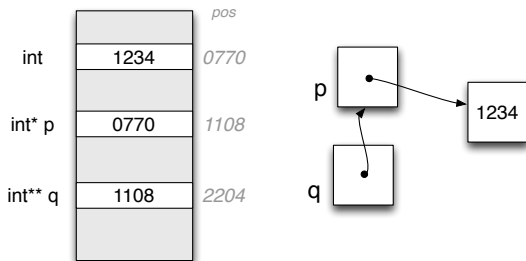
Punteros

- Un puntero se ve más claramente como una flecha (\rightarrow) que apunta a un objeto en algún lugar de la memoria



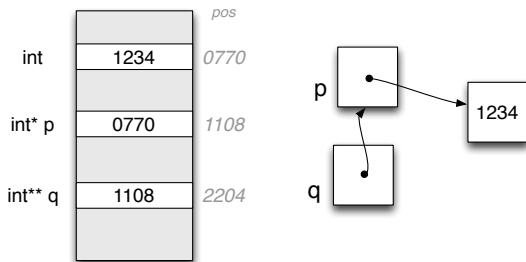
Punteros

- Un puntero se ve más claramente como una flecha (\rightarrow) que apunta a un objeto en algún lugar de la memoria
- Varios punteros pueden apuntar a lo mismo generando *aliasing* de punteros. (¡¡no borres más de una vez lo mismo!!)



Punteros

- Un puntero se ve más claramente como una flecha (\rightarrow) que apunta a un objeto en algún lugar de la memoria
- Varios punteros pueden apuntar a lo mismo generando *aliasing* de punteros. (¡¡no borres más de una vez lo mismo!!)
- Un puntero declarado y no inicializado apunta a cualquier cosa (peligro!)



Punteros en C++

- Dado el tipo T , el tipo T^* se denomina “puntero a T ”. Su tipo nos dice a qué tipo de objeto apunta.
- Valor (numérico): *una dirección de memoria*
- El valor de memoria “0” (NULL) está reservado como “invalido”
- Se puede tener T^{**} , T^{***} , ... (puntero a puntero a T , puntero a puntero a puntero a T , ...)

Punteros en C++

- Dado el tipo T , el tipo T^* se denomina “puntero a T ”. Su tipo nos dice a qué tipo de objeto apunta.
- Valor (numérico): *una dirección de memoria*
- El valor de memoria “0” (NULL) está reservado como “invalido”
- Se puede tener T^{**} , T^{***} , ... (puntero a puntero a T , puntero a puntero a puntero a T , ...)
- $\text{new } T$: devuelve un T^* apuntando a un nuevo objeto de tipo T (alojado en el *heap*) Reserva el espacio en memoria.
- $\text{delete } p$: borra lo que está siendo apuntado por el puntero p (¡sólo si lo que apunta fue creado con new !).
- $*p$: devuelve el **valor** apuntado por p
- $p \rightarrow (...)$: equivalente a $((*p).(...))$. Para usar con clases/structs.
- $\&v$: Sea v **variable** de tipo T , $\&v$ es un T^* con la dirección de memoria donde se aloja v

Ejemplos

```
// p: en el stack, pk lo va a apuntar
void ejemploPuntero(int N) {
    long k = 1234;
    long * pk, * pk2;           // pk, pk2 -> ???
    pk = &k;                    // pk -> k
    cout << k << "==" << *pk << "==" << *&k << endl;
    pk = new long;             // nuevo long en heap
    *pk = 10;
    (*pk)++;                    // ese long ahora vale 11
}
```

```
struct punto {int x; int y;};

void ejemploPunteroStruct() {
    punto * p = new punto;
    cout << p->x << '==' (*p).x << endl;
}
```

Referencias

- Dado el tipo T, T& es el tipo referencia a un objeto de tipo T.
- También “flecha”, pero que se usa exactamente igual que el objeto original
- Más seguros: la dirección de memoria está “escondida”
- Una referencia debe inicializarse con un objeto existente al declararse (o no compila): no pueden haber referencias apuntando a algo inválido o a NULL

Referencias

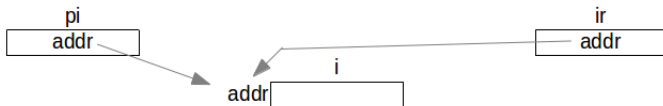
- Dado el tipo T, T& es el tipo referencia a un objeto de tipo T.
- También “flecha”, pero que se usa exactamente igual que el objeto original
- Más seguros: la dirección de memoria está “escondida”
- Una referencia debe inicializarse con un objeto existente al declararse (o no compila): no pueden haber referencias apuntando a algo inválido o a NULL

Punteros

```
int i;  
int *pi = &i;
```

Referencias

```
int i;  
int &ri = i
```




```
void funcionReferencias() {  
    int a = 10;  
    int & loco = a;  
    int & errorDeCompilacion; //No inicializado!  
    a += 10;  
    loco += 10; // es una ref, pero se usa igual  
    // cuanto valen ahora?  
}  
  
long & funcMortal() {  
    long a = 10;  
    return a;      // boom... por que?  
}
```

Tiempo de vida de una variable (bis)

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del `scope` que la declaró.

Memoria dinámica

La memoria pedida con *new* tiene un tiempo de vida que **comienza** cuando se llama a *new* y **termina** cuando se le hace *delete* al puntero (o termina el programa en ejecución (proceso) correspondiente).

Tiempo de vida de una variable (bis)

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del `scope` que la declaró.

Memoria dinámica

La memoria pedida con *new* tiene un tiempo de vida que **comienza** cuando se llama a *new* y **termina** cuando se le hace *delete* al puntero (o termina el programa en ejecución (proceso) correspondiente).

Atención

Notar que una variable de tipo puntero muere cuando finaliza su scope pero no así la memoria a la que apunta. Cuidado con los leaks!

Tiempo de vida de una variable (bis)

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del `scope` que la declaró.

Memoria dinámica

La memoria pedida con *new* tiene un tiempo de vida que **comienza** cuando se llama a *new* y **termina** cuando se le hace *delete* al puntero (o termina el programa en ejecución (proceso) correspondiente).

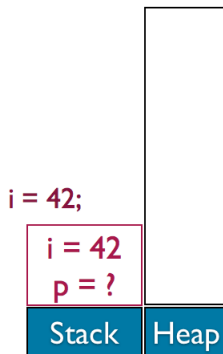
Atención

Notar que una variable de tipo puntero muere cuando finaliza su scope pero no así la memoria a la que apunta. Cuidado con los leaks!

Veamos un ejemplo!

Variables en el heap (1/9)

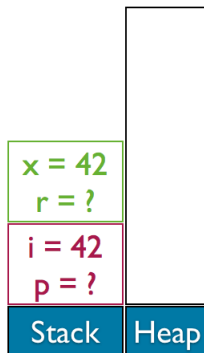
```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```



Variables en el heap (2/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

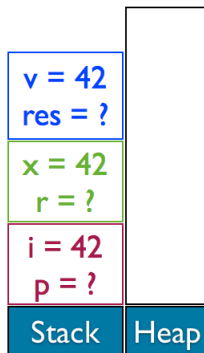
pasaManos(i);



Variables en el heap (3/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

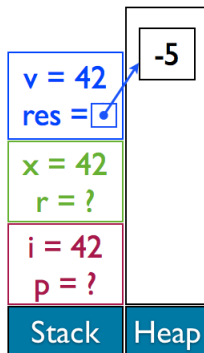
crearOtro(x);



Variables en el heap (4/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

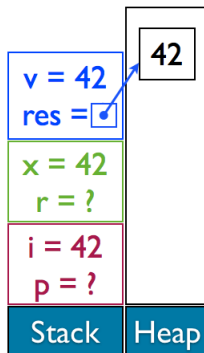
res = new int;



Variables en el heap (5/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

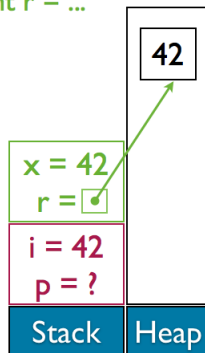
*res = v;



Variables en el heap (6/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

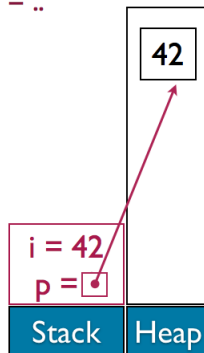
return res;
int r = ...



Variables en el heap (7/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

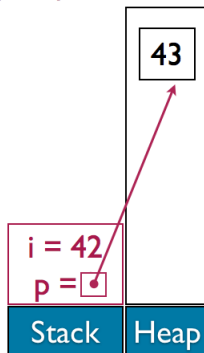
return r;
p = ..



Variables en el heap (8/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

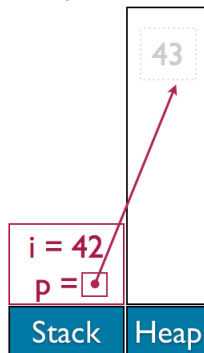
$*p = *p + 1;$



Variables en el heap (9/9)

```
int* crearOtro(int v) {  
    int* res = new int;  
    *res = v  
    return res;  
}  
int* pasaManos(int x) {  
    int* r = crearOtro(x);  
    return r;  
}  
int main() {  
    int i = 42;  
    int* p;  
    p = pasaManos(i);  
    *p = *p + 1;  
    delete p;  
    return 0;  
}
```

delete p;



Stack Vs. Heap: conclusiones y diferencias

- Dentro de un programa, el stack es un área de memoria de tamaño fijo y limitado, mientras que el heap es variable e “ilimitado”.

Stack Vs. Heap: conclusiones y diferencias

- Dentro de un programa, el stack es un área de memoria de tamaño fijo y limitado, mientras que el heap es variable e “ilimitado” .
- El stack lo usa y administra el compilador mediante el pasaje a assembler y no se puede controlar desde adentro del programa.

Stack Vs. Heap: conclusiones y diferencias

- Dentro de un programa, el stack es un área de memoria de tamaño fijo y limitado, mientras que el heap es variable e “ilimitado” .
- El stack lo usa y administra el compilador mediante el pasaje a assembler y no se puede controlar desde adentro del programa.
- El heap lo administra el programador mediante llamadas a funciones de la librería standar de C/C++ (malloc/free, new/delete).

Stack Vs. Heap: conclusiones y diferencias

- Dentro de un programa, el stack es un área de memoria de tamaño fijo y limitado, mientras que el heap es variable e “ilimitado” .
- El stack lo usa y administra el compilador mediante el pasaje a assembler y no se puede controlar desde adentro del programa.
- El heap lo administra el programador mediante llamadas a funciones de la librería standar de C/C++ (malloc/free, new/delete).
- El stack y el heap son espacios de memoria disjuntos. No comparten direcciones de memoria. **Atención al guardar variables del stack EN el heap y viceversa.**

Valgrind para chequear memory leaks:

- Compilar con información de debug:
\$ g++ -g <archivos> -o binario

Valgrind para chequear memory leaks:

- Compilar con información de debug:
`$ g++ -g <archivos> -o binario`
- Ejecutar por consola:
`$ valgrind --leak-check=full -v ./binario`



C++

***UN GRAN PODER CONLLEVA UNA
GRAN RESPONSABILIDAD...***

Veremos...

- El const
- Constructor por defecto
- Constructor con parámetros
- Constructor por copia
- Listas de inicialización
- Destructor
- Operador de asignación

Ejemplo: secuencia

- Queremos una secuencia de tamaño dinámico, con operaciones: nueva, agregarAtras/Adelante, iésimo, longitud, etc... ¿Cómo hacemos?

Ejemplo: secuencia

- Queremos una secuencia de tamaño dinámico, con operaciones: nueva, agregarAtras/Adelante, iésimo, longitud, etc... ¿Cómo hacemos?
- ¿Cómo podemos implementar la secuencia con arreglos?

Ejemplo: secuencia

- Queremos una secuencia de tamaño dinámico, con operaciones: nueva, agregarAtras/Adelante, iésimo, longitud, etc... ¿Cómo hacemos?
- ¿Cómo podemos implementar la secuencia con arreglos?
- Solución: cadena de nodos de longitud variable.

Ejemplo: secuencia

- Queremos una secuencia de tamaño dinámico, con operaciones: nueva, agregarAtras/Adelante, iésimo, longitud, etc... ¿Cómo hacemos?
- ¿Cómo podemos implementar la secuencia con arreglos?
- Solución: cadena de nodos de longitud variable.
- Sin usar memoria dinámica no alcanza... ¿por qué?

Ejemplo de estructuras

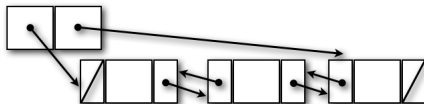
Listas simple y doblemente encadenadas.

¿Cómo es una lista vacía? ¿Cómo se hace para agregar adelante? ¿y atrás?

ListaSimple



ListaDoble



Ejemplo de estructuras

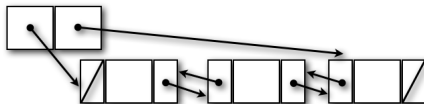
Listas simple y doblemente encadenadas.

¿Cómo es una lista vacía? ¿Cómo se hace para agregar adelante? ¿y atrás?

ListaSimple



ListaDoble



¿Por qué es indispensable la memoria dinámica? (Concretamente)

- Sin punteros y memoria dinámica no podemos crear más cosas que las que declaramos como variables.

Reflexiones sobre punteros y memoria dinámica

- Sin punteros y memoria dinámica no podemos crear más cosas que las que declaramos como variables.
- En C++, sin punteros son imposibles los tipos de datos recursivos

Reflexiones sobre punteros y memoria dinámica

- Sin punteros y memoria dinámica no podemos crear más cosas que las que declaramos como variables.
- En C++, sin punteros son imposibles los tipos de datos recursivos
- ¿Qué otras estructuras no se podrían implementar sin punteros?

El const se usa para indicar que...

- Una variable local o de clase es una constante.

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de “sólo lectura” para esa función/método de clase.

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de “sólo lectura” para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado (“solo lectura” de nuevo);

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de “sólo lectura” para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado (“solo lectura” de nuevo);
- Un valor de retorno devuelto por referencia es de “sólo lectura” para el llamador de una función/método de clase.

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de “sólo lectura” para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado (“solo lectura” de nuevo);
- Un valor de retorno devuelto por referencia es de “sólo lectura” para el llamador de una función/método de clase.

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de “sólo lectura” para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado (“solo lectura” de nuevo);
- Un valor de retorno devuelto por referencia es de “sólo lectura” para el llamador de una función/método de clase.

El const es una manera de “especificar” el comportamiento del código con respecto a la lectura/escritura de valores. Conviene usarlo porque el compilador nos va a alertar de algunos errores comunes.

```
// Una constante numerica  
const double PI = 3.1416;  
  
// Calcula el area: no modifica a c  
double area(const Circulo & c);  
  
// El punto devuelto se mira y no se toca.  
// Tampoco modifica al circulo  
const punto2d & Circulo::centro() const;  
  
// este punto si se puede modificar! (ojo!)  
// tampoco modifica al circulo  
punto2d & Circulo::centro() const;
```

- Cosas del tipo **T** se pueden usar en lugares donde es necesario algo del tipo **const T** (el compilador hace la conversión)...
- ... pero cosas del tipo **const T** NO se pueden usar en lugares donde es necesario algo del tipo **T**! Se les ocurre por qué?
- **const T** es *más restrictivo* que **T**. El compilador no va a hacer la conversión de **const T** a **T** porque es inseguro (estaría convirtiendo algo de solo lectura en algo escribible).

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.
- Estar atentos a las conversiones automáticas
- Prestar especial atención cuando creamos nuestras propias clases:
 - qué métodos tienen const sobre el parámetro implícito **this**
 - y cuáles sobre los parámetros de entrada y/o de retorno.

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

- Constructor que toma cualquier tipo y cantidad de parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

- Constructor que toma cualquier tipo y cantidad de parámetros
- Si se define, el compilador no provee el constructor por defecto

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

- Constructor que toma cualquier tipo y cantidad de parámetros
- Si se define, el compilador no provee el constructor por defecto
- Se pueden agregar tantos como se quiera a una clase

Ejemplo: constructor por defecto y con parámetros

```
class Test {  
    // Aca tenemos Test() implicito  
    int a;  
};
```

Ejemplo: constructor por defecto y con parámetros

```
class Test {  
    // Aca tenemos Test() implicito  
    int a;  
};  
  
class Test2 {  
    // Al declarar este, no hay  
    // constructor Test2() implicito  
    Test2(long l);  
};
```

```
int main() {  
    Test unTest; // cual sera el valor de unTest.a ?  
    Test2 elOtro; // compila?
```


Ejemplo: constructor por defecto y con parámetros

```
class Test {  
    // Aca tenemos Test() implicito  
    int a;  
};  
  
class Test2 {  
    // Al declarar este, no hay  
    // constructor Test2() implicito  
    Test2(long l);  
};
```

```
int main() {  
    Test unTest; // cual sera el valor de unTest.a ?  
    Test2 elOtro; // compila?  
    Test2 posta(200); // ahora si!  
}
```

- Constructor de la clase T que siempre toma un *const* $T\&$

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- **Invocado cuando se pasan parámetros por copia.**
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- **Invocado cuando se pasan parámetros por copia.**
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar.**

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- **Invocado cuando se pasan parámetros por copia.**
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar.**
- Si alguno de los miembros es un puntero...

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- **Invocado cuando se pasan parámetros por copia.**
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar.**
- Si alguno de los miembros es un puntero...

Constructor por copia

- Constructor de la clase T que siempre toma un *const T&*
- **Invocado cuando se pasan parámetros por copia.**
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar.**
- Si alguno de los miembros es un puntero... ¡va a copiar el valor del puntero! Eso no puede terminar bien...

Constructor por copia - ejemplos

```
class Copiable {  
    Copiable (const Copiable & c2) {  
        this→a = NULL;  
        // Lo tenemos que copiar  
        // explicitamente (si corresponde)  
        if (c2.a != NULL)  
            this→a = new int (*(c2.a));  
    }  
  
    int* a;  
};
```

```
int main() {  
    Copiable ccc; // como se construye ccc?  
    Copiable otroC(ccc); // como se construye ccc?  
}
```

Listas de inicialización

- Sintaxis usada en los constructores para construir variables miembro (algunas o todas). Importa el orden.
- Se vuelven indispensables cuando tenemos miembros sin constructor por defecto.

Listas de inicialización

- Sintaxis usada en los constructores para construir variables miembro (algunas o todas). Importa el orden.
- Se vuelven indispensables cuando tenemos miembros sin constructor por defecto.

```
// Esfera.h
class punto3d {
    double x,y,z;
    punto3d(double x0, double y0, double z0)
        : x(x0), y(y0), z(z0) { }
};

class esfera {
    double radio; punto3d centro;
    esfera(); // necesita lista!
};
```

Listas de inicialización (2)

```
// Esfera.cpp  
esfera::esfera()  
: radio(1), centro(0,0,0)  
{  
}
```

```
// o equivalentemente:  
esfera::esfera()  
: centro(0,0,0)  
{  
    radio = 1;  
}
```

Listas de inicialización - Otro ejemplo (3)

```
class Pp {  
    private:  
        int &valor;  
  
    public:  
        //Compila?  
        Pp(int &v)  
        {  
            valor = v;  
        }  
        int sumarYRetornar();  
};
```

Listas de inicialización - Otro ejemplo (3)

```
class Pp {  
    private:  
        int &valor;  
  
    public:  
        //Ahora sí.  
        Pp(int &v) : valor(v) { };  
        int sumarYRetornar();  
};
```

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos?

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos?

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (dèjà vu)

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (dèjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.
- Si alguno de los miembros es un puntero...

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.
- Si alguno de los miembros es un puntero...

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (dèjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.
- Si alguno de los miembros es un puntero... ¡va a copiar el valor del puntero! Eso no puede terminar bien...
- Se aplica sobre una instancia YA CONSTRUIDA: **¡hay que limpiar lo que ya estaba!**

Operador de asignación - ejemplos

```
class assignable {  
    assignable& operator=(const assignable& a2) {  
        if (this == &a2) return *this;    // a = a  
        // HAY QUE BORRAR LO VIEJO!  
        if (this→a != NULL) {  
            delete this→a;  
            this→a = NULL;  
        }  
        // copiamos si corresponde  
        if (a2.a != NULL)  
            this→a = new int (*(a2.a));  
        return *this;  
    }  
    int* a;  
};
```

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del `scope` de una variable

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del `scope` de una variable
- *No se lo llama explícitamente.*

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del `scope` de una variable
- *No se lo llama explícitamente.*
- Debe realizar todas las tareas de limpieza de memoria dinámica necesarias... ¡no queremos perder memoria!

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del `scope` de una variable
- *No se lo llama explícitamente.*
- Debe realizar todas las tareas de limpieza de memoria dinámica necesarias... ¡no queremos perder memoria!

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del `scope` de una variable
- *No se lo llama explícitamente.*
- Debe realizar todas las tareas de limpieza de memoria dinámica necesarias... ¡no queremos perder memoria!

```
class leaker {  
    int * p;  
  
    leaker(int tam){  
        p = new int[tam];  
    }  
  
    ~leaker() {} // No hago nada  
};
```

```
class limpita {  
    int * p;  
  
    limpita(int tam) { p = new int[tam]; }  
    ~limpita() { delete[] p; }  
    // delete p hace lo mismo?  
    // depende del compilador!!  
    // puede borrar solo la primera posicion  
    // delete[] es lo correcto  
}
```

“Regla de tres”

Cuando nos veamos obligados a definir...

- ...el constructor por copia
- ...el operador de asignación
- ...o el destructor

...probablemente tengamos que definir los tres.

“Regla de tres”

Cuando nos veamos obligados a definir...

- ...el constructor por copia
- ...el operador de asignación
- ...o el destructor

...probablemente tengamos que definir los tres.

Atención

Estos tres se autogeneran por el compilador si no los declaramos, por lo que, si lo que el compilador autogenera no sirve en un caso, probablemente tampoco sirva en los demás.

Ejercicio: Lista de alumnos de una materia

Cada elemento de esta lista es una tupla (L.U, Edad) implementada con un struct Alumno. Queremos algunas operaciones básicas:

- Crear una lista de alumnos
- Conocer su longitud
- Agregar elementos al principio/final
- Obtener o eliminar el *i*ésimo alumno de la lista

Ejercicio: Lista de alumnos de una materia

Cada elemento de esta lista es una tupla (L.U, Edad) implementada con un struct Alumno. Queremos algunas operaciones básicas:

- Crear una lista de alumnos
- Conocer su longitud
- Agregar elementos al principio/final
- Obtener o eliminar el *i*ésimo alumno de la lista

El archivo ListaAlumnos.zip se encuentra colgado en la página para que se diviertan.

```
typedef unsigned long Nat;  
struct Alumno { Nat LU, tel; };  
class ListaAlumnos {  
    // Constructores...  
    // Observadores  
    Nat longitud() const;  
    const Alumno & iesimo(Nat i) const;  
    Alumno & iesimo(Nat i);  
    // Operaciones  
    void agAdelante(const Alumno & elem);  
    void agAtras(const Alumno & elem);  
    void eliminar(Nat i);  
  
    private:  
    struct Nodo { /* ... */ };  
};
```