Memoria Dinámica en C++

Algoritmos y Estructuras de Datos II

DC-FCEyN-UBA

26 de Agosto de 2015

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.

Propiedades básicas

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

- Nombre: "**x**"

Propiedades básicas

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

- Nombre: "**x**"
- Tipo: **int**

Propiedades básicas

```
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**

Propiedades básicas

```
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;
```

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++){
      int g = 75;
}
cout << g << endl; //esto esta bien??
```

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++){
    double h = 29; //compila??
}
cout << h << endl;
```

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++){
    double h = 29; //compila??
}
cout << h << endl;
```

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).

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++){
    double h = 29; //compila??
}
cout << h << endl;
```

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).

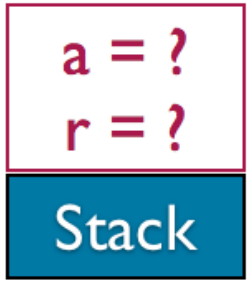
Veamos un ejemplo!

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;
}
```



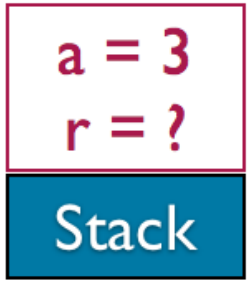
Variables en el stack (2/10)

```
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;



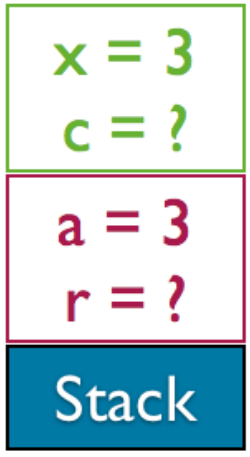
Variables en el stack (3/10)

```
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)



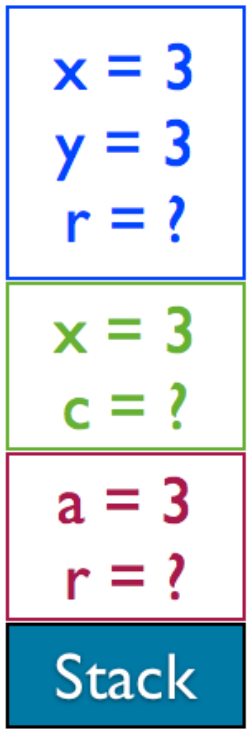
Variables en el stack (4/10)

```
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);



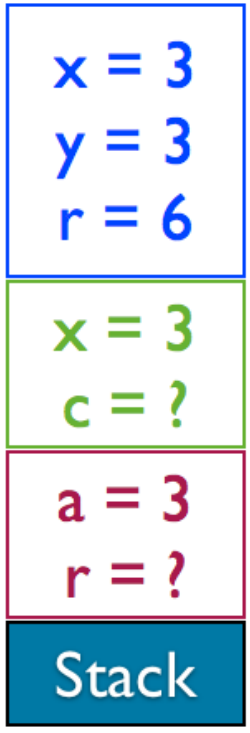
Variables en el stack (5/10)

```
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;



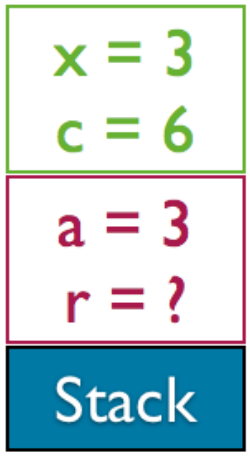
Variables en el stack (6/10)

```
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;
}

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

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



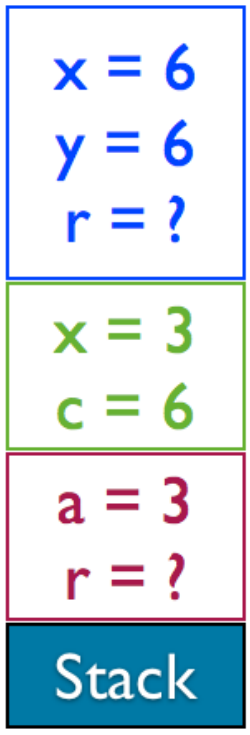
Variables en el stack (7/10)

```
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);



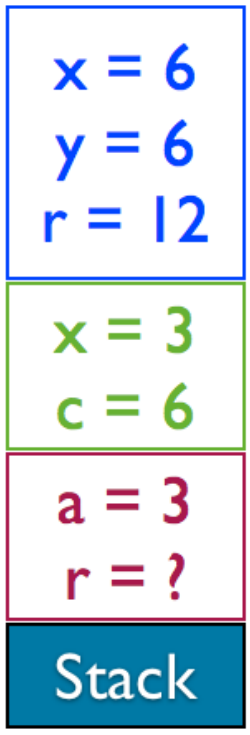
Variables en el stack (8/10)

```
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;



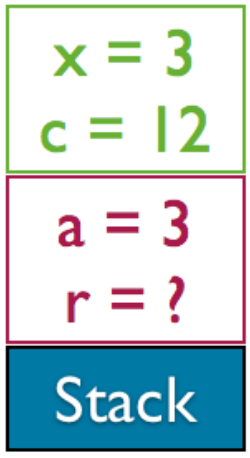
Variables en el stack (9/10)

```
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 = ...
```



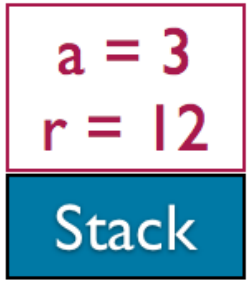
Variables en el stack (10/10)

```
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;
}
```

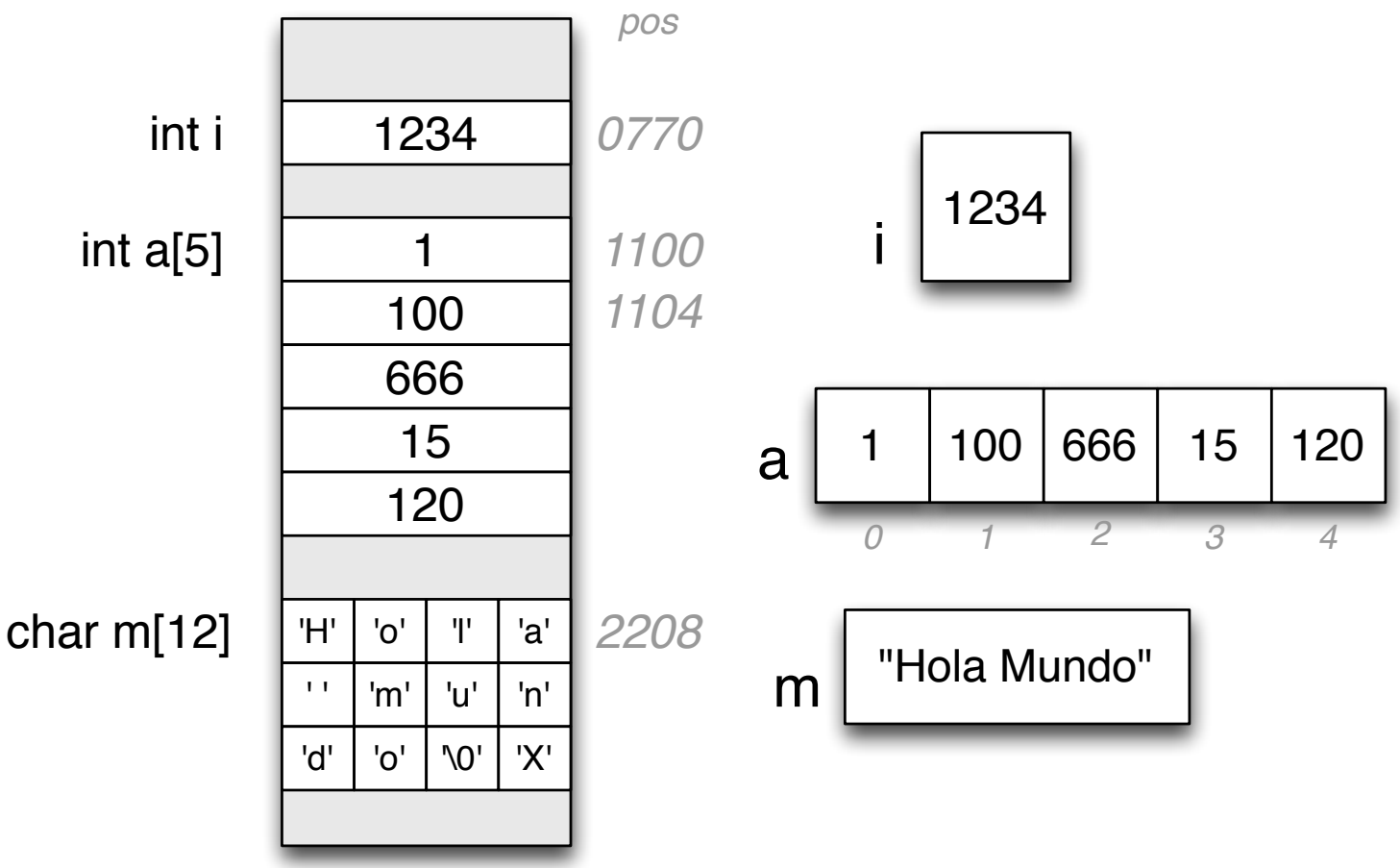
return c;
r = ..



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

Veamos un mapa



Punteros

- Dado el tipo T, el tipo T* se denomina “puntero a T”

Punteros

- Dado el tipo T, el tipo T* se denomina "puntero a T"
- Valor (numérico): *una dirección de memoria*

Punteros

- Dado el tipo T, el tipo T* se denomina “puntero a T”
- Valor (numérico): *una dirección de memoria*
- El valor de memoria “0” (NULL) está reservado como “invalido”

Punteros

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

Punteros

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

Repitan conmigo: FLECHAS

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

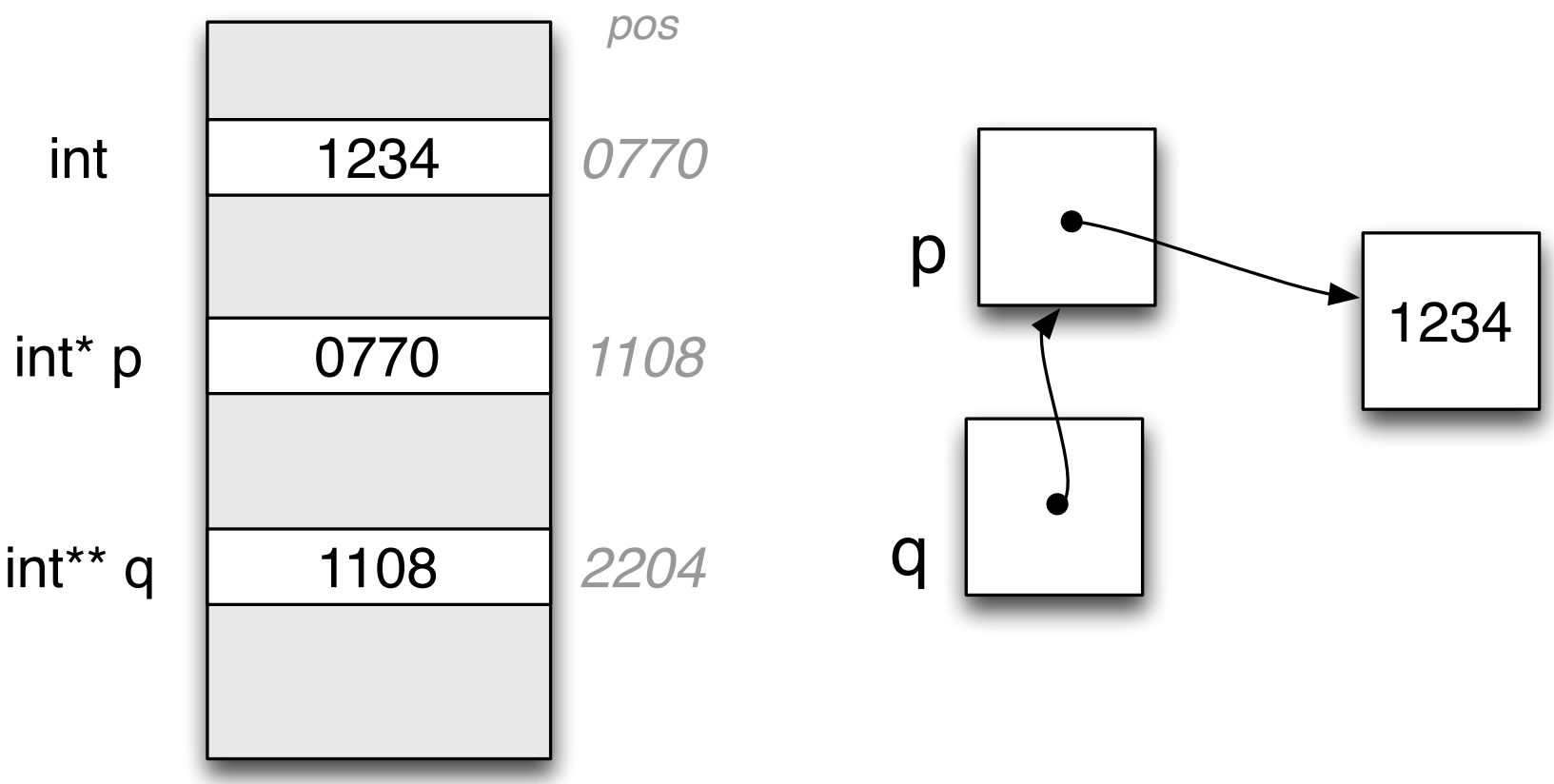
Repitan conmigo: FLECHAS

- 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!!)

Repitan conmigo: FLECHAS

- 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!)

¡Veo los punteros!



Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)

Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)

Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)
- *p[i]*: devuelve el iésimo elemento de un array creado dinámicamente con *new T[N]* y **p* devuelve la primera posición.

Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)
- *p[i]*: devuelve el iésimo elemento de un array creado dinámicamente con *new T[N]* y **p* devuelve la primera posición.
- *delete p / delete [] p*: borra lo que está siendo apuntado por el puntero p (¡sólo si lo que apunta fue creado con new!). delete[] borra arrays dinámicos. No confundirlos!!!

Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)
- *p[i]*: devuelve el iésimo elemento de un array creado dinámicamente con *new T[N]* y **p* devuelve la primera posición.
- *delete p / delete [] p*: borra lo que está siendo apuntado por el puntero p (¡sólo si lo que apunta fue creado con new!). delete[] borra arrays dinámicos. No confundirlos!!!
- **p*: devuelve el **valor** apuntado por p

Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)
- *p[i]*: devuelve el iésimo elemento de un array creado dinámicamente con *new T[N]* y **p* devuelve la primera posición.
- *delete p / delete [] p*: borra lo que está siendo apuntado por el puntero p (¡sólo si lo que apunta fue creado con new!). delete[] borra arrays dinámicos. No confundirlos!!!
- **p*: devuelve el **valor** apuntado por p
- *p→(...)*: equivalente a "(*p).(...)". Para usar con clases/structs.

Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)
- *p[i]*: devuelve el iésimo elemento de un array creado dinámicamente con *new T[N]* y **p* devuelve la primera posición.
- *delete p / delete [] p*: borra lo que está siendo apuntado por el puntero p (¡sólo si lo que apunta fue creado con new!). delete[] borra arrays dinámicos. No confundirlos!!!
- **p*: devuelve el **valor** apuntado por p
- *p→(...)*: 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 funcAburrida(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
  pk2 = new long[N]; // nuevo array de long en heap
  cout << ( *pk2 == pk2[0] ) << endl;
  pk2[20] = 2932;         // asigno posicion 20 en pk2
  delete pk;
  delete [] pk2;  // borro toda la memoria del array
}
```

Ejemplos (2)

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

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

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- Dado el tipo T, T& es el tipo referencia a un objeto de tipo T

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- 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

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- 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"

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- 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 (*pero recordemos que se implementan con punteros...*)

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- 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 (*pero recordemos que se implementan con punteros...*)
- *Pero por qué las hay?*

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- 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 (*pero recordemos que se implementan con punteros...*)
- *Pero por qué las hay?*

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- 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 (*pero recordemos que se implementan con punteros...*)
- *Pero por qué las hay?* Porque en el fondo son punteros... !!!

Ejemplos (2)

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

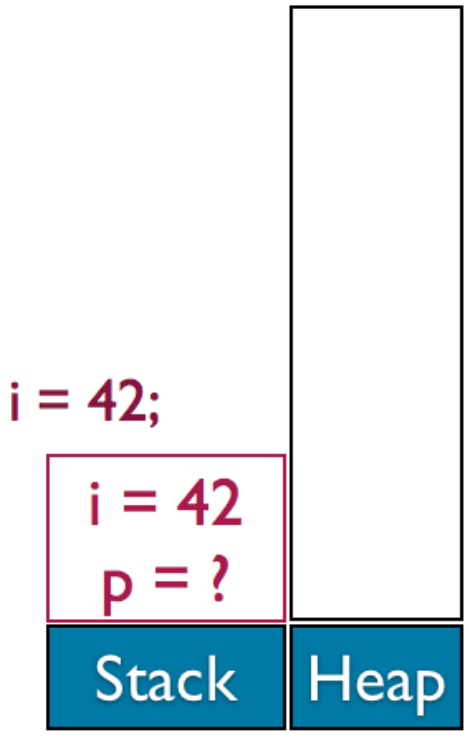
Atención

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)

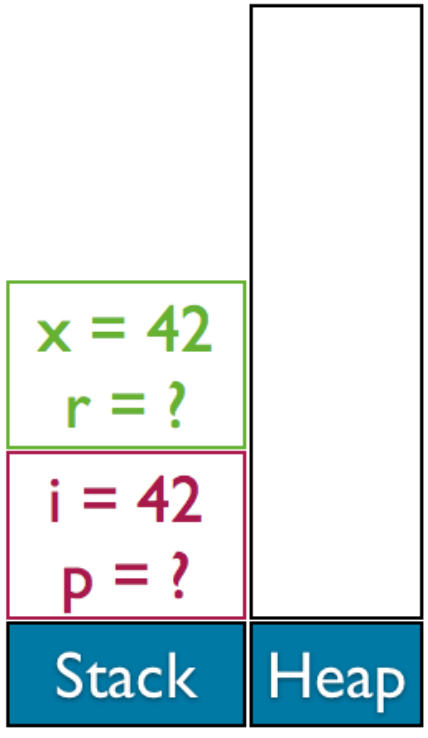
```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```



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;
}
```

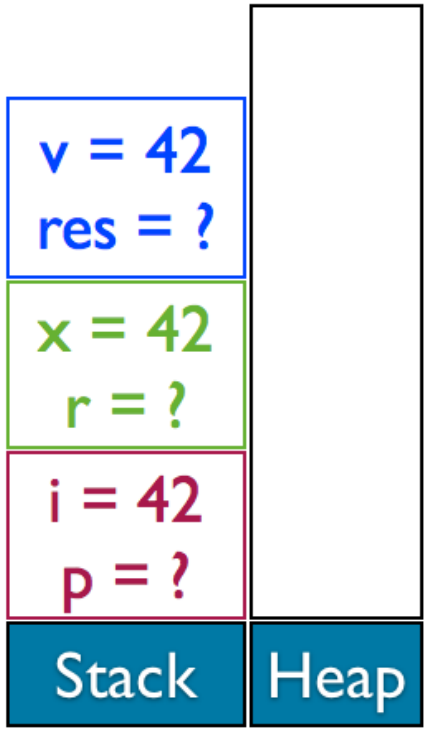
pasaManos(i);



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;
}
```

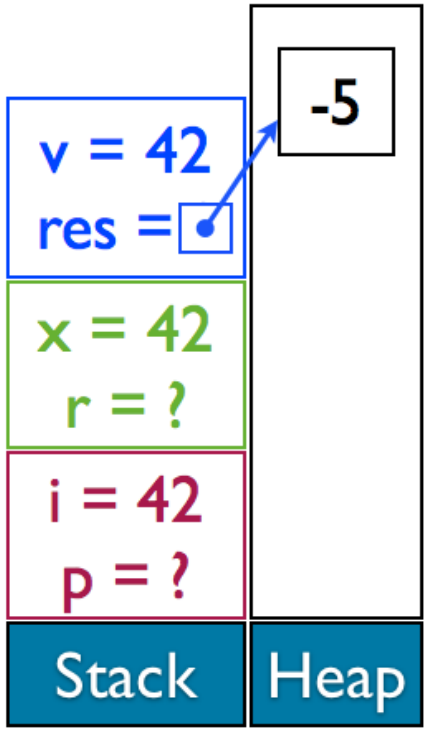
crearOtro(x);



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;
}
```

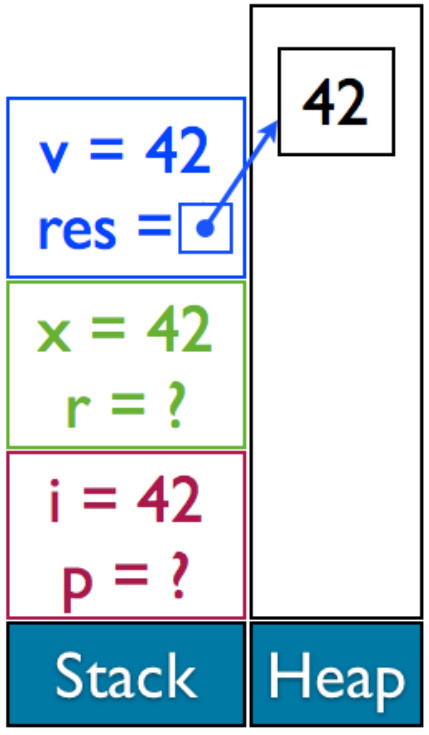
res = new int;



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;
}
```

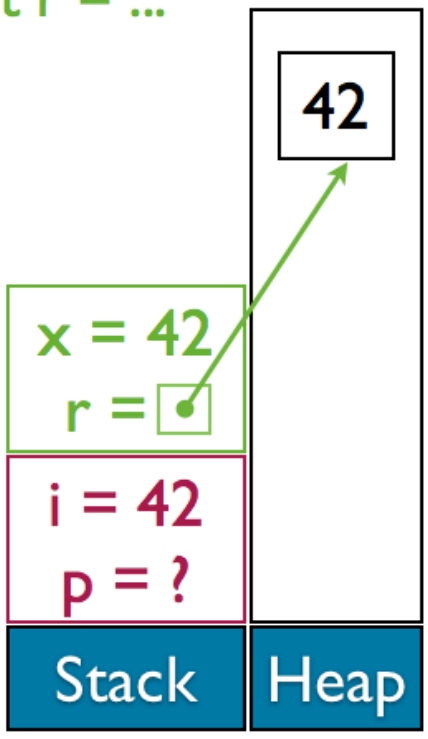
*res = v;



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 = ...

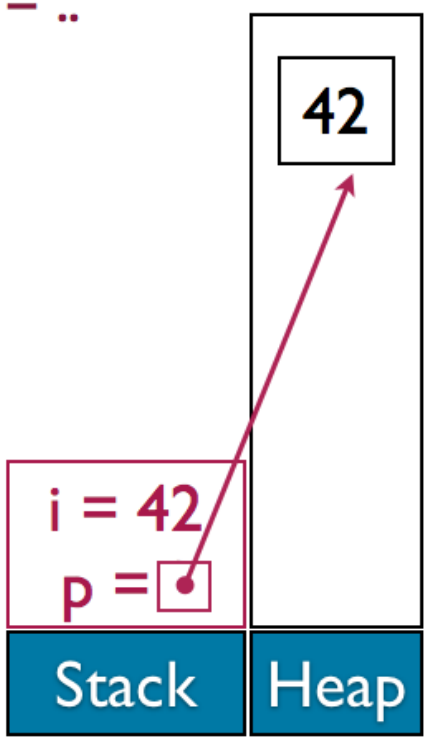


Variables en el heap (7/9)

```
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;

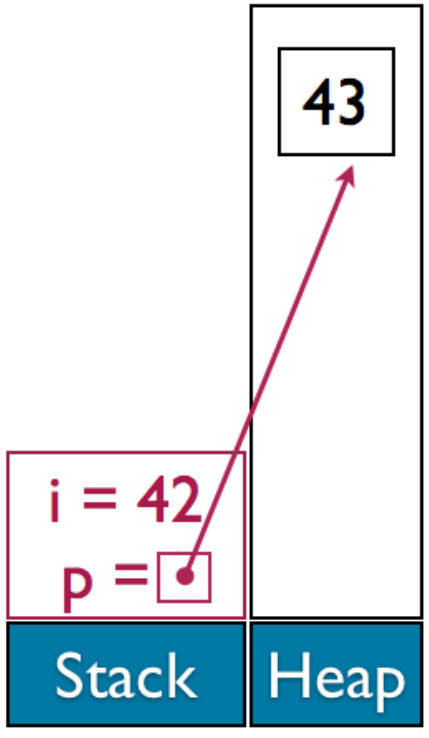
p = ..



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;
}
```

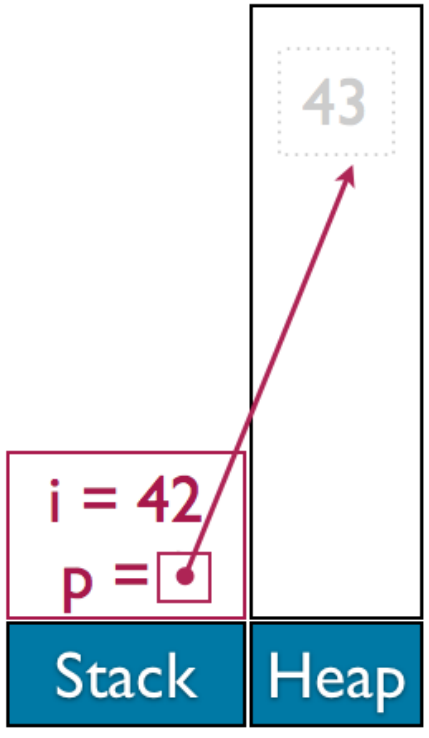
*p = *p + 1;



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.

Stack Vs. Heap: conclusiones y diferencias

C++

UN GRAN PODER CONLLEVA UNA GRAN RESPONSABILIDAD...

Clases en C++

Veremos...

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

Clases en C++

Veremos...

- El const
- Constructor por defecto
- Constructor con parámetros
- Constructor por copia
- Listas de inicialización
- Destructor
- Operador de asignación
- Pero primero, un poco de motivación

Estructuras de datos, para qué

Ejemplo: secuencia

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

Estructuras de datos, para qué

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?

Estructuras de datos, para qué

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.

Estructuras de datos, para qué

Ejemplo: secuencia

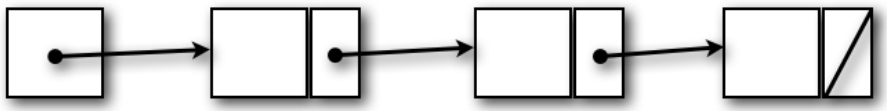
- Queremos una secuencia de tamaño dinámico, con operaciones: nueva, agregarAtras/Adelante, iésimo, longitud, etc... ¿Cómo hacemos?
- ¿Cómo podemos implementar la secuencia con arreglos?
- Solución: cadena de nodos de longitud variable.
- Sin usar memoria dinámica no alcanza... ¿por qué?

Ejemplo de estructuras

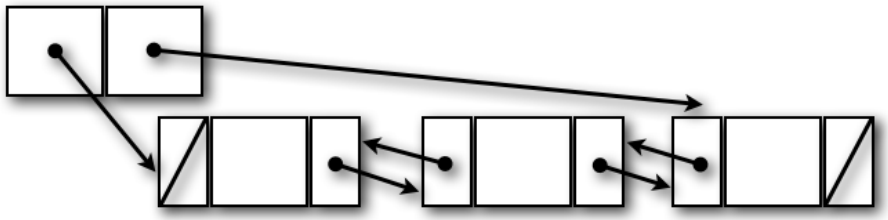
Listas simple y doblemente encadenadas.

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

ListaSimple



ListaDoble

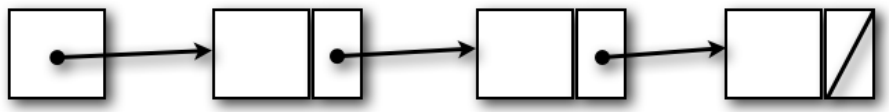


Ejemplo de estructuras

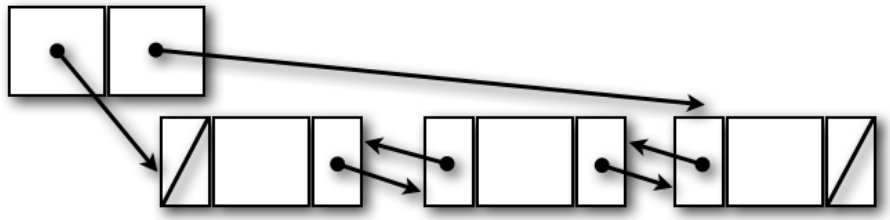
Listas simple y doblemente encadenadas.

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

ListaSimple



ListaDoble



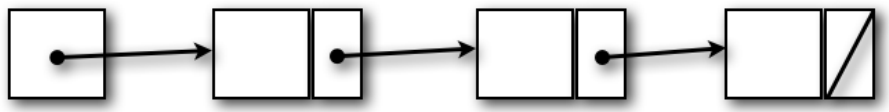
Hagamos una lista en el pizarrón!!

Ejemplo de estructuras

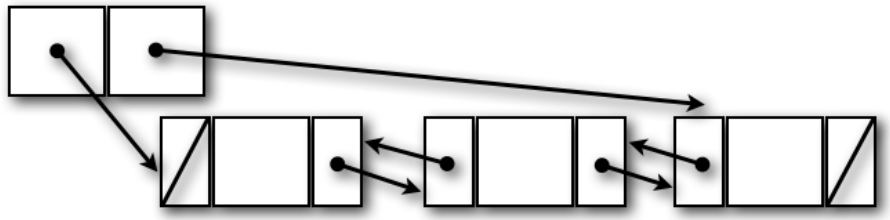
Listas simple y doblemente encadenadas.

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

ListaSimple



ListaDoble



Hagamos una lista en el pizarrón!!

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

Reflexiones sobre punteros y memoria dinámica

- 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?

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.

¡const!

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.

¡const!

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);

¡const!

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.

¡const!

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.

¡const!

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.

const, punteros y referencias

- El tipo **const T *** se leen como *puntero a un T constante*. El que es de sólo lectura es el T apuntado.

const, punteros y referencias

- El tipo **const T *** se leen como *puntero a un T constante*. El que es de sólo lectura es el T apuntado.
- El tipo **const T&** y se lee como *referencia constante a T*. El que es de sólo lectura es el T referenciado.

const, punteros y referencias

- El tipo **const T *** se leen como *puntero a un T constante*. El que es de sólo lectura es el T apuntado.
- El tipo **const T&** y se lee como *referencia constante a T*. El que es de sólo lectura es el T referenciado.
- Cuidado: **T** y **const T** son tipos distintos!!

const - conversiones

- Cosas del tipo **T** se pueden usar en lugares donde es necesario algo del tipo **const T** (el compilador hace la conversión)...

const - conversiones

- 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 - conversiones

- 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).

const - ejemplos

```
// Una constante numerica
const double PI = 3.1416;
```

const - ejemplos

```
// Una constante numerica
const double PI = 3.1416;

// Calcula el area: no modifica a c
double area(const Circulo & c);
```

const - ejemplos

```
// 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;
```

const - ejemplos

```
// 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;
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() { return 7; }
  // mas codigo...
};

// esto no va a compilar... por?
void mostrarVidas(const GatoMontes& g) {
  cout << g.vidas() << endl;
}
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() const { return 7; }
  // mas codigo...
};

// Ahora si!
void mostrarVidas(const GatoMontes& g) {
  cout << g.vidas() << endl;
}
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() { return 7; }
  // mas codigo...
};

// esto compila?
void mostrarVidas(GatoMontes g) {
  cout << g.vidas() << endl;
}
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() { return 7; }
  // mas codigo...
};

// esto compila?
void mostrarVidas(GatoMontes g) {
  cout << g.vidas() << endl;
}

// cual es el sentido de esto?
void mostrarVidasBis(const GatoMontes g) {
  cout << g.vidas() << endl;
}
```

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.
- Estar atentos a las conversiones automáticas

Reflexiones sobre el const

- 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:

Reflexiones sobre el const

- 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**

Reflexiones sobre el const

- 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);
};
```

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 por copia

- 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;
};
```

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 (déjà 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 (déjà 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 (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... ¡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 asignable {
  asignable& operator=(const asignable& 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
};
```

Destructor

```
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.

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.
- Los constructores y destructores nos dan un forma de meternos con el scope de una variable y *manejar* su tiempo de vida.

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.
- Los constructores y destructores nos dan un forma de meternos con el scope de una variable y *manejar* su tiempo de vida.
- Podemos definir una función (el constructor) que se llama cuando *comienza* el scope de la variable y otra para cuando *finaliza* (el destructor)

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.
- Los constructores y destructores nos dan un forma de meternos con el scope de una variable y *manejar* su tiempo de vida.
- Podemos definir una función (el constructor) que se llama cuando *comienza* el scope de la variable y otra para cuando *finaliza* (el destructor)
- Respetar la convención: "El que lo crea, lo destruye".

Pasaje de parámetros por referencia o por copia

```
class Grandota {
  Grandota(){ a = new int[100000];}
  Grandota(const Grandota& gr) {
    this->a = new int[100000];
    for (int i=0; i < 100000, i++) {
      this->a[i] = gr.a[i];
    }
  }
  ~Grandota(){ delete [] a; }

  int* a;
};
```

Pasaje de parámetros por referencia o por copia

```
class Grandota {
  Grandota(){ a = new int[100000];}
  Grandota(const Grandota& gr) {
    this->a = new int[100000];
    for (int i=0; i < 100000, i++) {
      this->a[i] = gr.a[i];
    }
  }
  ~Grandota(){ delete [] a; }

  int* a;
};

void funPesada(Grandota g); //que recibe la funcion?
```

Pasaje de parámetros por referencia o por copia

```
class Grandota {
  Grandota(){ a = new int[100000];}
  Grandota(const Grandota& gr) {
    this->a = new int[100000];
    for (int i=0; i < 100000, i++) {
      this->a[i] = gr.a[i];
    }
  }
  ~Grandota(){ delete [] a; }

  int* a;
};

void funPesada(Grandota g); //que recibe la funcion?
void funLiviana(Grandota& g); //y aca?
```

Pasaje de parámetros por referencia o por copia

```
class Grandota {
  Grandota(){ a = new int[100000];}
  Grandota(const Grandota& gr) {
    this->a = new int[100000];
    for (int i=0; i < 100000, i++) {
      this->a[i] = gr.a[i];
    }
  }
  ~Grandota(){ delete [] a; }

  int* a;
};

void funPesada(Grandota g); //que recibe la funcion?
void funLiviana(Grandota& g); //y aca?
void funLivianaYSegura(const Grandota& g); //y aca?
```

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Destructor

- No usar new nos garantiza no perder memoria, y por lo tanto, no definir el destructor.
- Liberar con delete la memoria que pedimos con new, únicamente de nuestra clase y **no** en otras.

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes? Templates!

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes? Templates!

Sugerencias y consejos adicionales

- Empezar con el testing a medida que van implementando, es decir **durante** la creación de la clase.

  Tengan en cuenta que algunos métodos pueden ser necesarios para muchas de las funciones de test, por lo tanto, es aconsejable empezar por esos test.
- Además, de los casos de test que puedan ser provistos por la cátedra les recomendamos fuertemente que realicen **siempre** sus propios tests.
- Dado que la implementación no debe perder memoria, es una buena práctica utilizar la herramienta *valgrind* **durante** el desarrollo.
- Si sus clases son *Template*, probar para distintas instancias de tipos (no sólo los tipos básicos).

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)
- Buscar casos borde donde pueda fallar

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)
- Buscar casos borde donde pueda fallar
- Documentar el propósito de cada test y que chequea

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)
- Buscar casos borde donde pueda fallar
- Documentar el propósito de cada test y que chequea

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)
- Buscar casos borde donde pueda fallar
- Documentar el propósito de cada test y que chequea

Valgrind para chequear memory leaks:

- Compilar con información de debug:

```
$ g++ -g <archivos> -o binario
```

Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)
- Buscar casos borde donde pueda fallar
- Documentar el propósito de cada test y que chequea

Valgrind para chequear memory leaks:

- Compilar con información de debug:

```
$ g++ -g <archivos> -o binario
```

- Ejecutar por consola:

```
$ valgrind --leak-check=full -v ./binario
```

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 iésimo 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 iésimo alumno de la lista

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

Clase ListaAlumno

```
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 { /*...*/ };
};
```