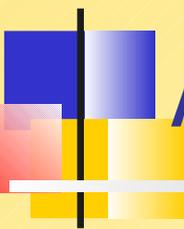


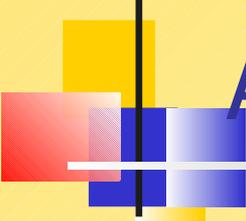


UNIVERSIDAD TÉCNICA
FEDERICO SANTA MARÍA



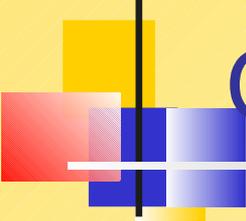
Asignación Dinámica de Memoria

Agustín J. González
ELO 329



Asignación Dinámica

- Asignación Dinámica es la creación de un objeto mientras el programa está en ejecución. Para ello se usa el operador `new`.
- Los objetos creados con `new` son almacenados en un gran espacio de memoria libre llamado `heap`.
- Cuando son creados de esta manera, los objetos permanecen en el `heap` hasta que son removidos de él.
- El operador `delete` remueve los objetos del `heap` liberando ese espacio de memoria.



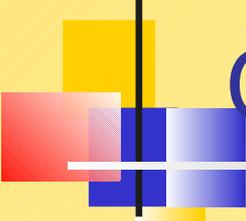
Creando un Objeto

Usando el operador `new`, aquí creamos un objeto entero en el heap y asignamos su dirección a `P`.

```
int * P = new int;
```

Ahora podemos usar el puntero de la misma manera como en los ejemplos previos.

```
*P = 25;           // assign a value  
  
cout << *P << endl;
```



Operadores new y delete

El operador `new` retorna la dirección a objeto recién creado. El operador `delete` borra el objeto y lo deja no disponible.

Llama al constructor `Student()`



```
Student * pS = new Student;
```

```
·
```

```
·
```

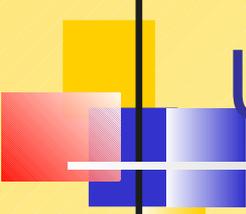
```
// use the student for a while...
```

```
·
```

```
·
```

```
delete pS; // elimina el objeto y
```

```
// retorna espacio de memoria! Versión C++ de free.
```



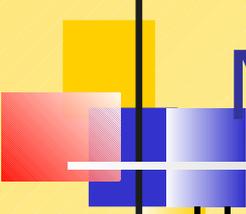
Usando new en Funciones

Si se crea un objeto dentro de una función, lo más probable es que haya que eliminar el objeto al interior de la misma función. En el ejemplo, la variable `pS` se sale del alcance una vez terminado el bloque de la función.

```
void MySub()
{
    Student * pS = new Student;

    // use the Student for a while...

    delete pS;        // delete the Student
}                    // pS disappears
```



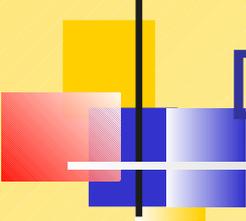
Memory Leaks (fuga de memoria)

Un *memory leak* (o *fuga de memoria*) es una condición de error creada cuando un objeto es dejado en el heap con ningún puntero conteniendo su dirección. Esto puede pasar si el puntero al objeto sale fuera del alcance:

```
void MySub()
{
    Student * pS = new Student;
    // use the Student for a while...

} // pS goes out of scope
```

(el objeto Student permanecerá en el heap)

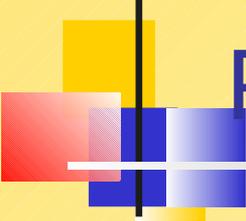


Direcciones retornada por Funciones

Una función puede retornar la dirección de un objeto que fue creado en el heap.

```
Student * MakeStudent()  
{  
    Student * pS = new Student;  
  
    return pS;  
}
```

(más) 



Recibiendo un puntero

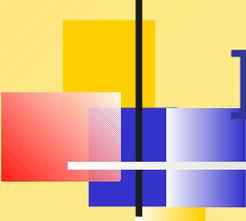
(continuación)...

El que llama la función puede recibir una dirección y almacenarla en una variable puntero. El puntero permanece activo mientras el objeto Student es accesible.

```
Student * pS;
```

```
pS = MakeStudent();
```

```
// now pS points to a Student
```



Invalidación de Punteros

Un *puntero se invalida* cuando el objeto referenciado es borrado y luego tratamos de usar el puntero. Esto puede generar un error de ejecución irrecuperable.

```
double * pD = new double;
```

```
*pD = 3.523;
```

```
·
```

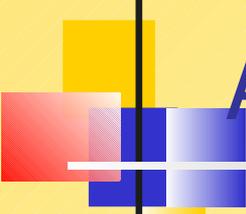
```
·
```

```
delete pD; // pD es inválido...
```

```
·
```

```
·
```

```
*pD = 4.2; // error!
```

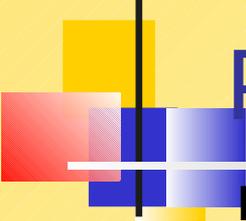


Abolición de Punteros inválidos

Se recomienda asignar NULL a los punteros tan pronto los objetos referenciados son borrados.

Y, obviamente chequear por NULL antes de usar un puntero. ¿Qué es mejor: tener un error de ejecución o un programa que parece funcionar pero arroja resultados no correctos?

```
delete pD;  
pD = NULL;  
.  
.  
if( pD != NULL )    // check it first...  
    *pD = 4.2;
```



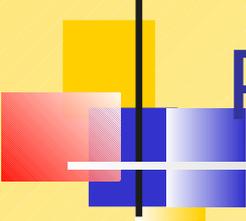
Paso de Punteros en Funciones

El paso de un puntero a una función es casi idéntico al paso por referencia. La función tiene acceso de escritura y lectura a los datos referenciados.

Así es como `swap()` se implementa en C, antes que C++ introdujera parámetros por referencia:

```
void swap( int * A, int * B ) /* C */
{
    int temp = *A;
    *A = *B;
    *B = temp;
}
```

```
void swap( int & A, int & B ) // C++
{
    int temp = A;
    A = B;
    B = temp;
}
```

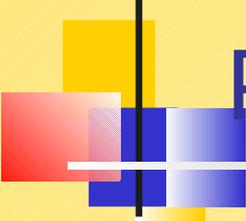


Punteros con el calificador Const

Un puntero calificado como `const` garantiza que el programa sólo tenga acceso de lectura de los datos referenciados por el puntero.

```
void MySub( const int * A ) // Contenido constante!  
{  
    *A = 50; // error  
    A++;    // ok  
}
```

El puntero sí puede ser modificado, pero esto no tiene efecto duradero o posterior ya que puntero es pasado por valor (se crea uno local y se copia el valor el parámetro actual).

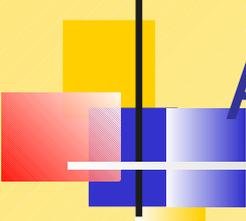


Punteros Constantes

La declaración de un **puntero constante** sólo garantiza que el puntero en si no pueda ser modificado.

```
void MySub( int * const A ) // Puntero constante
{
    *A = 50; // ok
    A++;    // error
}
```

Los datos referenciados por el puntero si pueden ser ser modificados.



Arreglos y Punteros

El nombre de un arreglo es compatible en asignaciones con un puntero al primer elemento de un arreglo . Por ejemplo, *p contiene scores[0].

```
int scores[50]; // scores es equivalente a un puntero
                constante
```

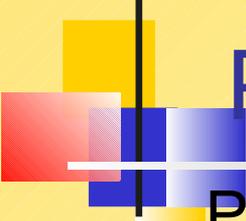
```
int * p = scores;
```

```
*p = 99;
```

```
cout << scores[0]; // "99"
```

```
p++; // ok
```

```
scores++; // error: scores is const
```



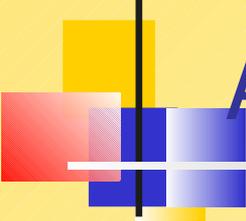
Recorriendo un Arreglo

Programadores C y C++ frecuentemente usan punteros para recorrer arreglos. Hoy día el acceso vía índices corre tan eficientemente como el recorrido con punteros debido a la optimización hechas hoy por compiladores.

```
int scores[50];
int * p = scores;

for( int i = 0; i < 50; i++)
{
    cout << *p << endl;
    p++;           // increment the pointer
}
```

Actualmente el código con recorrido con índice es también eficiente.



Arreglos de Punteros

Un arreglo de punteros usualmente contiene la dirección de objetos dinámicos. Esto ocupa poco almacenamiento para el arreglo y mantiene la mayor parte de los datos en el heap.

```
Student * elo329[10];
```

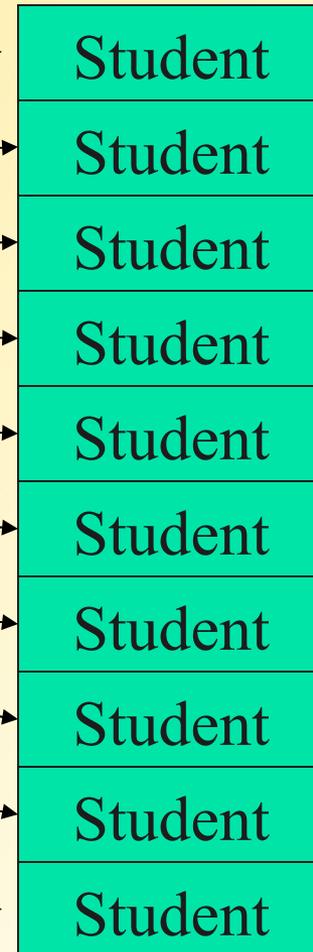
```
for(int i = 0; i < 10; i++)  
{  
    elo329[i] = new Student;  
}
```

diagrama 

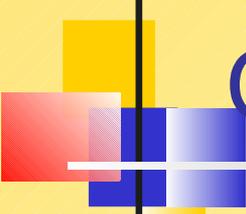
Arreglo de Punteros

Stack

Heap



elo329 []

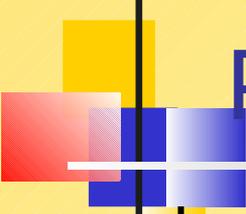


Creación de un Arreglo en el heap

Podemos crear arreglos completos en el heap usando el operador `new`. Hay que recordar eliminarlo cuando corresponda. Para ello basta incluir "[]" antes del nombre del arreglo en la sentencia delete.

```
void main()  
{  
    double * samples = new double[10];  
  
    // samples is now an array....  
    samples[0] = 36.2;  
  
    delete [] samples; //eliminación de un arreglo  
}
```

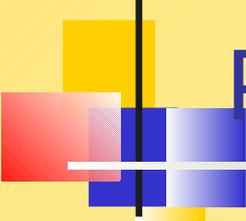
array size



Punteros y Clases

Los punteros son efectivos cuando los encapsulamos en clases porque podemos controlar su tiempo de vida.

```
class Student {  
public:  
    Student();  
    ~Student();  
  
private:  
    string * courses; // array of course names  
    int count;       // number of courses  
};  
  
// más...
```

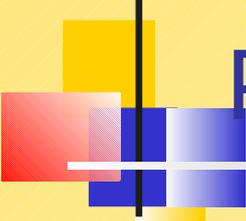


Punteros en Clases

El constructor crea el arreglo, y el destructor lo borra.
De esta forma pocas cosas pueden salir mal ...

```
Student::Student()  
{  
    courses = new string[50];  
    count = 0;  
}
```

```
Student::~~Student()  
{  
    delete [] courses;  
}
```



Punteros en Clases

...excepto cuando hacemos una copia de un objeto Student. El constructor de copia de C++ conduce a problemas.

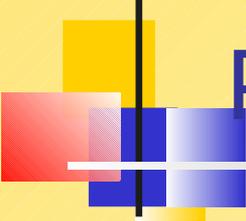
Por ejemplo aquí un curso asignado al estudiante X termina en la lista de cursos del estudiante Y:

```
Student X;
```

```
Student Y(X);          // construct a copy
```

```
X.AddCourse("elo 329"); // suponemos que tenemos este  
método en Student
```

```
cout << Y.GetCourse(0); // "elo 329" , suponemos que este  
método existe
```

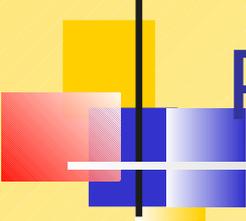


Punteros en Clases

Para prevenir este tipo de problemas, creamos un **constructor de copia** que efectúa lo conocido como una *copia en profundidad*.

```
Student::Student(const Student & S2)
{
    count = S2.count;
    courses = new string[count];

    for(int i = 0; i < count; i++)
        courses[i] = S2.courses[i];
}
```



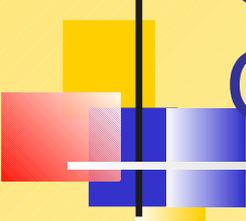
Punteros en Clases

Por la misma razón, tenemos que sobrecargar (overload) el operador de asignación.

```
Student & Student::operator =(const Student & S2)
{
    delete [] courses; // delete existing array
    count = S2.count;
    courses = new string[count];
    for(int i = 0; i < count; i++)
        courses[i] = S2.courses[i];

    return *this;
}
```

Si una clase requiere un constructor de copia, también requerirá la sobrecarga del operador asignación y definición del destructor.



C++ Containers (contenedores) en Clases

Cuando usamos contenedores estándares de C++ tales como listas y vectores en una clase, no hay problema con el constructor de copia en C++ porque todos ellos implementan adecuadamente estas operaciones.

```
class Student {  
public:  
    Student();  
  
private:  
    vector<string> courses;  
};
```