UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Electrónica

Manejo de Punteros y Objetos en Memoria Dinámica en C++

ELO329: Diseño y Programación Orientados a Objetos

Departamento de Electrónica

Universidad Técnica Federico Santa María

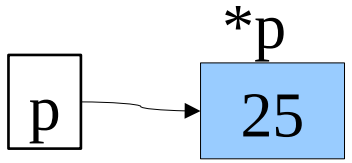
Creando un Objeto en el heap

```
int * p = new int;
```

- Usando el operador **new**, aquí creamos un entero en el heap y asignamos su dirección a **p**.
- Ahora podemos usar el puntero de la misma manera como en los ejemplos previos.

```
*p = 25;              // assign a value


cout << *p << endl;
```



Asignación Dinámica de Memoria

- Asignación Dinámica de memoria se logra al crear objetos (o tipos básicos) en zonas de memoria asignadas en tiempo de ejecución. Para ello se usa el operador new.
- Esta es la única forma usada en Java para crear objetos.
- Los objetos creados con new son almacenados en el heap, un gran espacio de memoria gestionado por el sistema operativo.
- Cuando objetos son creados de esta manera, éstos permanecen en el heap hasta que son removidos con el operador delete o el programa termina.
- A diferencia de Java, debemos remover explícitamente los objetos ubicados en el heap. Recordar que Java usa el recolector de basura activado por la máquina virtual.

Operadores new y delete

```
Student * ps = new Student;      // llama al constructor Student()
```

- El operador new retorna la dirección al objeto recién creado. El operador delete invoca al destructor y retorna la memoria al heap. El objeto ya no es accesible.

```
// suponga que aquí usamos ps previo por un rato  … y luego
   delete ps;     // invoca destructor, elimina el objeto y
                  //  retorna espacio de memoria!
                  //   Versión C++  (en C existe función 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 MyFunction() {
  Student * ps = new Student; // al término ps desaparece
                              // automáticamente, pero el objeto creado no.
  // usamos Student por un rato… luego debemos hacer


  delete ps;          // llama al destructor ~Student() y borra el
}                     // estudiante apuntado por ps.
```

- Si olvidamos el último delete, hay fuga de memoria

Fugas de Memoria (Memory Leaks)

- Un memory leak (o fuga de memoria) es una condición indeseable creada cuando un objeto es dejado en el heap y ningún puntero contiene su dirección. Esto puede pasar si el puntero al objeto queda fuera de alcance:

```
void MyFunction()  {
  Student * ps = new Student;
  // usamos el estudiante ps por un rato

}  // ps sale del alcance
```

- El objeto Student permanecerá en el heap !!! hasta el final del programa, puede generar falla por uso total de la memoria si la función es invocada muchas veces.

Direcciones retornada por Funciones o métodos

- 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)...

- Al llamar la función se recibir una dirección y se puede almacenar en una variable puntero. El valor apuntado hace sentido mientras el objeto Student es accesible.

```
Student * ps;
ps = MakeStudent();

// Ahora ps apunta a Student
// esto es OK, pero en algún momento hay que
// retornar la memoria invocando delete ps.
```

Invalidación de Punteros

- Un puntero se invalida cuando el objeto referenciado es borrado. Si tratamos de usar el puntero, genera un error de ejecución irrecuperable.

```
double * pd = new double;
*pd = 3.523;

delete pd;   // luego de esto la dirección pd no es nuestra...

*pd = 4.2;   // error! De ejecución.
```

Arreglos y Punteros

- El nombre de un arreglo es compatible en asignaciones con un puntero al primer elemento de un arreglo .

```
int scores[50];     // scores es equivalente a un puntero constante
int * p = scores;  // p también es puntero, pero lo podemos modificar.
*p = 99;              // *p corresponde a scores[0]
cout << scores[0];    // "99"

p++;              // es ok.  Ahora *p corresponde a scores[1]
scores++;      // error: scores es const
                  // al incrementar un puntero, éste avanza en el
                  // tamaño del objeto apuntado.
```

Arreglos de Punteros

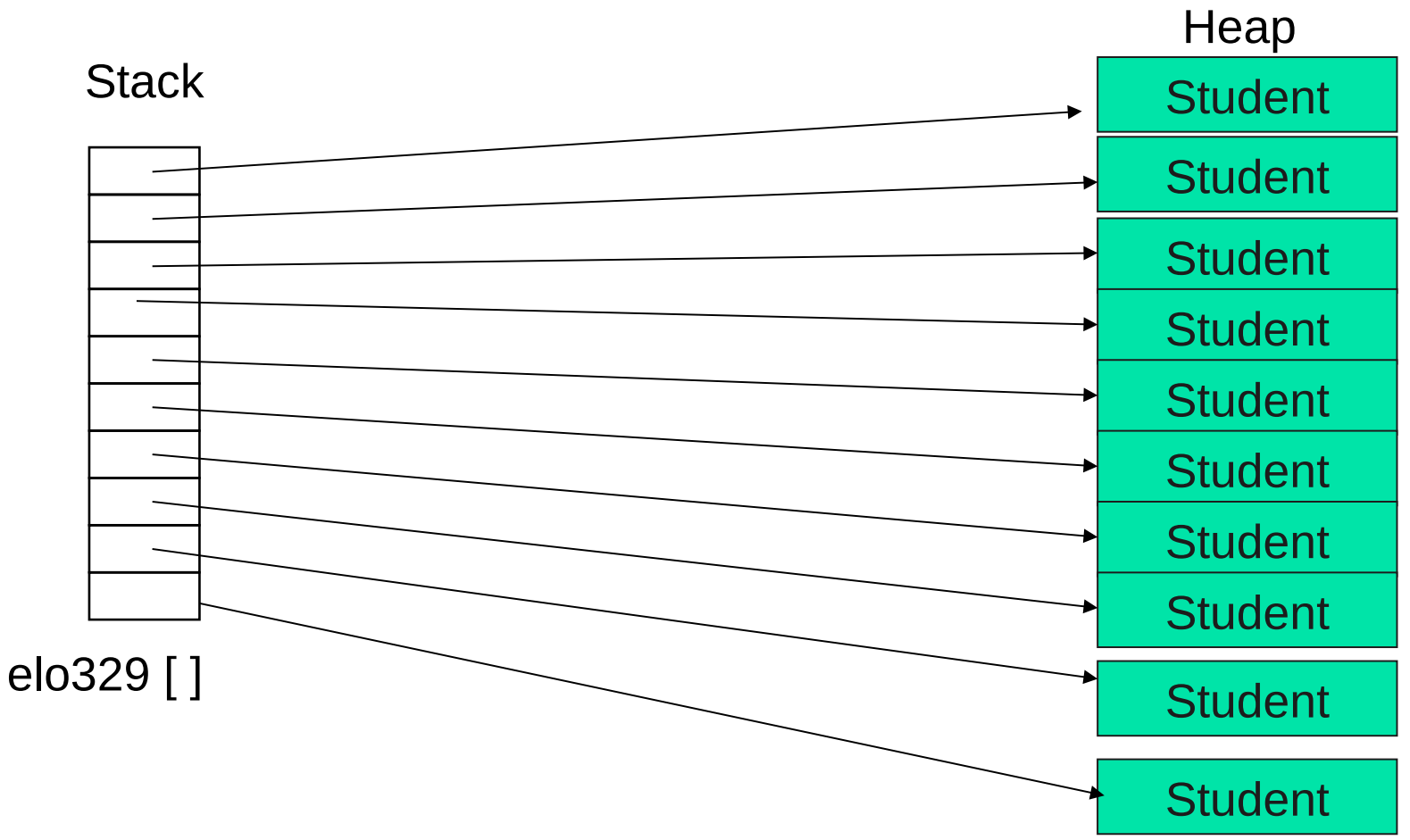
- Un arreglo de punteros usualmente contiene la dirección de objetos en memoria dinámica. Esto ocupa poco almacenamiento para el arreglo y mantiene la mayor parte de los datos en el heap.

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

Diagrama 

```
// eliminación
for(int i = 0; i < 10; i++) {
   delete elo329[i];
}
```

Arreglo de Punteros



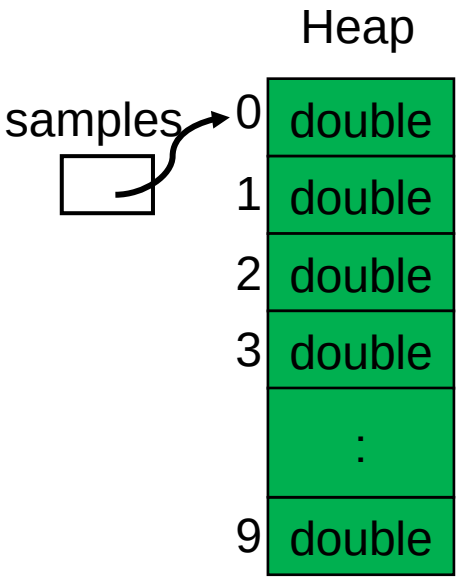
No necesariamente adyacentes en heap!

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]; // tamaño definido
                                     // en tiempo de ejecución
  // samples es un arreglo en heap ....
  samples[0] = 36.2;

  delete [ ] samples;  //eliminación de un arreglo desde el heap
}                      // no se requiere poner su tamaño
```



Punteros como atributos de Clases

- Los punteros son efectivos cuando los encapsulamos en clases porque podemos controlar su tiempo de vida.
- Debemos poner cuidado con la copia baja o copia profunda ya vista en Java.

```
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);          // Usa constructor copia por omisión

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

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

Otro caso de cuidado: Paso de parámetros por valor

- Cuando usamos paso por valor, la variable local es creada usando el constructor copia de la clase.
- void foo (Student s) { ….}
  // luego usamos
  Student jose;
  foo(jose); // s es inicializado usando Student s(jose)

- Esto es importante, NO creer que se invoca algo del tipo
  Student s;
  s=jose; // aquí se usaría operador asignación.
- Lo anterior sugiere implementar el constructor copia.
- La acción por omisión del constructor copia y la asignación **es** copia baja.

Copia en profundidad

- Para prevenir este tipo de problemas, creamos un constructor copia que efectúa una copia en profundidad.

```
Student::Student(const Student & s2)  { // constructor copia
  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;
}
```

Regla de ORO:
Si una clase requiere un constructor copia, la sobrecarga del operador asignación o implementación del destructor, también requerirá los otros.

Contenedores C++ en Clases

- Cuando usamos contenedores estándares de C++ como listas y vectores en una clase, no hay problema con el constructor de copia porque todos ellos implementan adecuadamente el constructor copia, la sobrecarga de asignación y el destructor.

```
class Student {
public:
  Student();

 private:
  vector<string> courses;
};
```