1.00 Clase 37

Breve estudio de C++:
Guía para leer programas de C++

Programas de C(++) sin clases

```
// Archivo main_temp.C: programa main
#include <iostream.h>        // Archivo de encabez. estándar (C)
#include "celsius.h"         // Archivo de encabezado de usuario

int main( ) {                         // Puede ignorar args main
   for (double temp= 10.0; temp <= 30.1; temp+=10.0)
       cout << celsius(temp) << endl;    // System.out.println
   cout << " Fin de la lista" << endl;    // C usa  printf()
   return 0;}                             // Java System.exit(0)

// Archivo celsius.C: archivo con código con función
double celsius(double fahrenheit)       // Llamado por valor
{return (5.0/9.0*(fahrenheit - 32.0));}

// Archivo celsius.h: archivo de encabezado con función
prototype double celsius(double fahrenheit);

// Sin relación entre nombres de archivos, métodos o clases en C++
// Distribuye archivos .h y .o pero no archivos .C a usuarios
```

Llamada con ejemplo de referencia

```
#include <iostream.h>
void triple( int& base);          // & signif. enviar referencia
                                 // Normalmente arriba en archivo .h
int main() {
   int value;
   cout << "Escriba un entero: ";
   cin >> value;
   triple( value);
   cout << " El valor triplicado es: " << value << endl;
   return 0;}

void triple(int& base) {         // Debe coincidir con prototipo
   base *= 3;}

// En C++ puede elegir la llamada por referencia o por valor para
// casi todos los primitivos y objetos, al contrario que en Java
// donde primitivos son "por valor" y objetos "referencia por valor"
// La llamada por valor ocurre si no se utilizan &, como en Java
```

Llamada con ejemplo de puntero

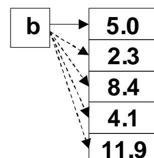
```
#include <iostream.h>
void triple( int* base);        // * significa puntero (dirección)
                                   // La referencia es constante
ptr int main() {                      // Ptr es una ref variable
can
   int value;                     // apunta a tipo
   cout << "Escriba un entero: ";
   cin >> value;
   triple( &value);              // Debe enviar dirección (ptr)
   cout << " El valor triplicado es: " << value << endl;
   return 0;}

void triple(int* base) {          // Debe coincidir con prototipo
   *base *= 3;}                   // * significa dereferencia o
                                  // usa el valor, no la
                                   // dirección
// Está anticuado (C no admite llamada por referencia)
// Tecnicamente, el puntero es pasado por el valor ("ref falsa")
```

Aritmética de puntero

```
#include <iostream.h>
double average1(double b[], int n);      // Prototipos de función
double average2(double b[], int n);

int main() {
   double values[]= {5.0, 2.3, 8.4, 4.1, 11.9};
   int size= 5;          // Los arrays de C++ no conocen su tamaño
   double avg1= average1(values, size);
   double avg2= average2(values, size);
   cout << "Las medias son: " << avg1 << " y " << avg2 << endl;
   return 0;}
```

Aritmética de puntero, p.2

```
double average1(double b[], int n) {
   double sum= 0;
   for (int i= 0; i < n; i++)
       sum += b[i];
   return sum/n; }          // Debería comprobar n> 0

double average2(double b[], int n) {  // double* b también ok
   double sum= 0;
   for (int i= 0; i < n; i++)
       sum += *(b + i);
   return sum/n; }
```



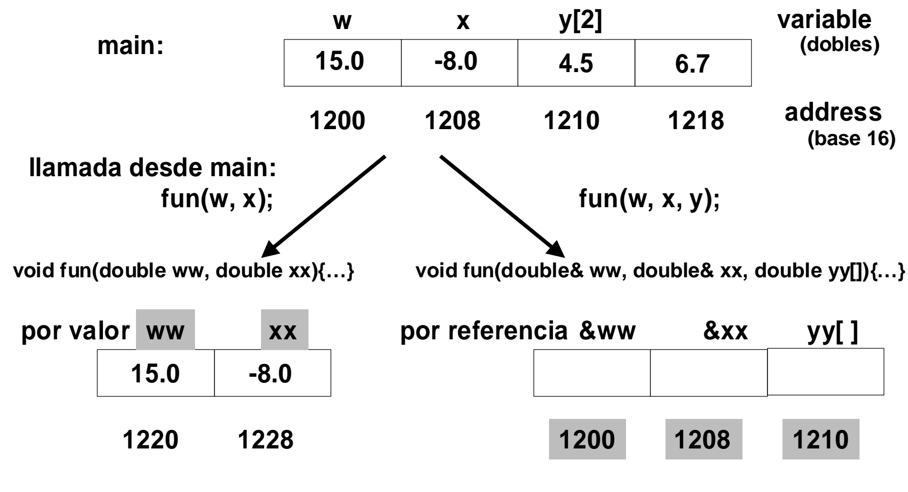
Programa *bracket* (C, C++)

```
#include <math.h>    // Obsoleto, usa #include <cmath>
#define FACTOR 1.6   // Obsoleto, usea const double FACTOR=1.6;
#define NTRY 50      // Usa const int NTRY= 50;

int zbrac(float (*func)(float), float *x1, float *x2) {
   void nrerror(char error_text[]);       // Función prototipo
   int j;            // Puede definir vars encima o espontáneas
   float f1, f2;
   if (*x1 == *x2) nrerror("Rango inicial incorrecto");
   f1= (*func)(*x1);
   f2= (*func)(*x2);
   for (j=1; j <=NTRY; j++) {
       if (f1*f2 < 0.0) return 1;     // false/true son 0, no 0
       if (fabs(f1) < fabs(f2))
              f1= (*func)(*x1 += FACTOR*(*x1-*x2));
       else
              f2= (*func)(*x2 += FACTOR*(*x2-*x1));
    }
   return 0;
}
```

Main() para *bracket*

```
#include <math.h>           // Obsoleto, usa #include <cmath>
#include <iostream.h>       // Obsoleto, usa #include <iostream>
using namespace std;

// Cualquier lugar aquí o en archivo .h que es #included
aquí: int zbrac(float (*func)(float), float *x1, float *x2);
float f(float x);

int main() {
   float n1= 7.0;
   float n2= 8.0;
   int bracketFound= zbrac(f, &n1, &n2);
   cout << "Lower " << n1 << " upper " << n2 << endl;
   return 0;   }

void nrerror(char error_test[]) { cout << error_test << endl; }

float f(float x) {  return x*x -2.0;}
```

Pasar argumentos: value, reference



Clase Point

```
// Archivo Point.h
#include <iostream>
using namespace std;

class Point{
  public:
      Point (double a=0, double b=0);  // Constructor predet.
      ~Point(){};                      // Destructor
      double getX();                   // Sólo prototipo
      double getY();
      void print();
      Point addPoint (Point& g);       // Elige valor o ref
  private:
      double x, y;
} ;
```

Clase Point, cont.

```
// Archivo Point.C
#include "Point.h"
Point::Point(double a, double b) {x=a; y=b;}
double Point::getX() {return x;}
double Point::getY() {return y;}
void Point::print(){cout << '(' << x << ',' << y <<')'<< endl;}
Point Point::addPoint(Point& g)          // Pasa por ref
{
       Point h;                     // No usa nuevo si no dinámico
       h.x= x + g.x;                // x,y son la x de este punto
       h.y= y + g.y;                // x,y de g se añaden
       us return h;
}
// Operador de resolución de alcance :: indica que la función es
// miembro de la clase Point . Sin él, no acceso a datos privados
```

Ejemplo de programa Point

```
// Archivo pointDemo.C
#include <iostream>    // Nuevo encabezado de estilo (podría usar antiguo)
using namespace std;
#include "Point.h"

int main(){
   double a;
   Point p1(3,4), p2(2), p3;    // Constructor (no requiere 'new')
   p3= p1.addPoint(p2);            // Agrega point1 y point2
   p3.print();
   a= p3.getX();
   cout << "x: " << a << endl;
   return 0;}
```

Clase Point exquisita

```
// Archivo Point.h
#include <iostream>
#include <cmath>
using namespace std;

class Point{
   friend double distance1(const Point& p, const Point& q);
   friend ostream& operator<<(ostream& os, const Point& p);
public:
   Point(double a=0.0, double b=0.0);    // Constructor
   Point(const Point& p);                // Copia constructor
   Point operator+(const Point& p) const;  // Añade 2 Point
   Point operator-() const;       // Unario menos
   Point& operator=(const Point& p);     // Asignación
   ~Point() {};                          // Destructor
private:
   double x, y;
};
```

Clase Point exquisita, p.2

```
// Archivo Point.C con cuerpos de función (necesita cmath e iostream)
  #include "Point.h"
  Point::Point(double a, double b)          // Constructor
     { x=a; y=b;}
  Point::Point(const Point& p)              // Copia constructor
     {x=p.x; y=p.y;}
  Point Point::operator+(const Point& p2) const  // Añade 2 Point
     { return Point(x+p2.x, y+p2.y);}
  Point Point::operator-() const            // Unario menos
     { return Point(-x, -y);}
  Point& Point::operator=(const Point& p2) // Asignación
     { if (this != &p2) {                   // Comprueba si p2=p2
          x= p2.x;
          y= p2.y;}
     return *this;}
```

Clase Point exquisita, p.3

```
// Archivo Point.C con cuerpos de función, continuación
// Funciones admitidas: distance and output (cout)
double distance1(const Point& p, const Point& q)
   { double dx= p.x - q.x;
     double dy= p.y - q.y;
     return sqrt(dx*dx + dy*dy);}

ostream& operator<<(ostream& os, const Point& p)
   { os << '(' << p.x << ',' << p.y << ')';
     return os; }
```

Uso de la clase Point

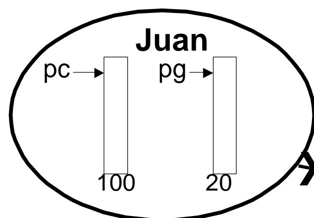
```
#include <iostream>
using namespace std;
#include "Point.h"

int main(){
   Point p1(3,4), p2(2);    // Usa constructor, args predet.
   Point p3(p2);            // Usa copiar constructor
   Point p4= Point(1,2);    // Operador asig.(copia miembros)
   p3= p1 + p2;             // Igual que p3= p1.operator+(p2)
   p4= p1 + p2 + p3;        // Encadenamiento
   p3= -p1;                 // Unario menos
   p2= p4 + p1;             // Podríamos implementar resta
   double a;
   a= distance1(p1, p2);
   cout << "La distancia de  p1 a p2 es: " << a << endl;
   cout << "p1= " << p1 << " and p2= " << p2 << endl; return
   0; }
   // Nuestros métodos de matrices de  Java funcionarían mejor ante
      sobrecarga de operaciones
```

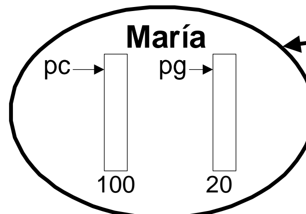
Constructores y destructores

Asignación de memoria dinámica

```
Class Student
  public:
  Student(…)
     { pc= new courselist[ncourses];
       pg= new gpalist[nterm];}
    ~Student()
     { delete[] pc;
       delete[] pg; }
```

Constructores y destructores, p.2





Programa main

```
int main() {
   {     //Func o bloquea alcance
   Student Joe;
   …
   }    // Fin alcance
…
   {    // Otro alcance
   Student Mary;
   …
   }    // Fin alcance
…
}
```

Constructores y destructores, p.3

- No hay gestión de memoria en el programa o las funciones main: objetivo de C++
  - En C, la memoria se gestionaba para cada variable
    - Era preciso recordar asignarla y liberarla al terminar, ocurriesen o no estos eventos.
    - Los errores de memoria dinámica superan el 50% de los errores de programa de C
  - En C++, construimos la gestión de memoria en clases
    - 'New' sólo en constructores; 'delete' sólo en destructores
    - El desarrollador de la aplicación ve casi automáticamente la recolección de basura. "Nunca" utiliza *new*; crea objetos *new* sólo definiéndolos: estudiante Joe, igual que int i
    - El desarrollador de las clases tiene el control del recolector de basura cuando lo necesita
  - La recolección de basura en C++ es difícil en listas, árboles, etc.

Errores de gestión de memoria

- Eliminar la misma memoria dos veces es un error
  - ¡Difícil de encontrar!
  - Truco: comente todas las eliminaciones si se presenta un error extraño en un programa con memoria dinámica
    - Si el error desaparece, significa que había eliminado memoria dos veces
- No eliminar memoria cuando hay que hacerlo supone una pérdida
  - Perder 100 bytes por llamada en un servidor web con un millón de llamadas al día supone comprar gigabytes de memoria y obtener errores periódicamente de todos modos

Clase de plantilla Stack

```
template <class TYPE>
class stack {
 public:
  explicit stack(int size): max_len(size), top(EMPTY)
    {s= new TYPE[size];}
  ~stack()
    { delete [] s;}
  void reset()
    { top= EMPTY;}
  void push(TYPE c)
    { assert(top != max_len -1) ; s[++top]= c;}
  TYPE pop()
    { assert (top != EMPTY); return s[top--];}
  TYPE top_of() const
    { assert (top != EMPTY); return s[top];}
  bool empty()
    { return ( top == EMPTY);}
  bool full()
    { return ( top == max_len -1);}
private:
  enum {EMPTY = -1};
  TYPE* s;
  int max_len;
  int top;};
```

Programa main, plantilla Stack

```
int main(){                              // Debe #include iostream
  int size;
  char book;
  cout << "Enter size of stack: ";
  cin >> size;
   stack<char> library(size);
   if (!library.full())
    library.push('a');
  if (!library.full())
    library.push('x');
  if (!library.empty()){
      book= library.pop();
      cout << "Primero guardado: " << book << endl;}
  if (!library.empty()){
      book= library.pop();
      cout << "Libro anterior, guardado después: " << book << endl;}
  if (!library.full())
    library.push('g');
  book= library.top_of();
  cout << "Siguiente libro devuelto: " << book << endl;}
```

Herencia: especificadores de acceso

```
class Base : {
public: …
protected: …
private: …};

class Derived : AccessSpecifier Base {
public:…
protected:…
private:… };
```

Tipo de miembro base	Especificador de acceso		
	Publico	Protegido	Privado
Publico	Publico	Protected	Privado
Protegido	Protegido	Protected	Privado
Privado	Inaccesible	Inaccesible	Inaccesible

Se conserva el especificador más restrictivo. Casi siempre utiliza herencia pública. La accesibilidad no se hereda y no es transitiva.

ResearchProject revisado

```
class Student {
public:
  // Array de estudiantes de RProject requiere constructor predet.
   Student() {firstName= " "; lastName= " ";} Student(
   const string fName, const string lName):
        firstName(fName), lastName(lName) {}

   // Función GetData es virtual: interfaz obligatoria,
   // implementación predet. Para clases derivadas
   virtual void GetData() const
   { cout << firstName << " " << lastName << " ";}
   virtual ~Student {}            // Destructor
private:
   string firstName, lastName;
};

// Si agregamos:   virtual double GetPay() = 0;
// Student se convierte en abstracto; requiere que cada clase
// derivada implemente GetPay(). También podemos definir
// métodos const , como métodos finales de Java
```

Undergrad

```
class Undergrad : public Student {
public:
   Undergrad(string fName, string lName, double hours,
   double rate);           // Cuerpo en archivo distinto
   double GetPay() const
        { return UnderWage * UnderHours;}
   virtual void GetData() const
   {
        Student::GetData(); // Instead of super.GetData()
        cout << "weekly pay: $" << GetPay() << endl;
   }
private:
   double UnderWage;
   double UnderHours;
};
// Mismo modelo para grad, clases grad especiales
```

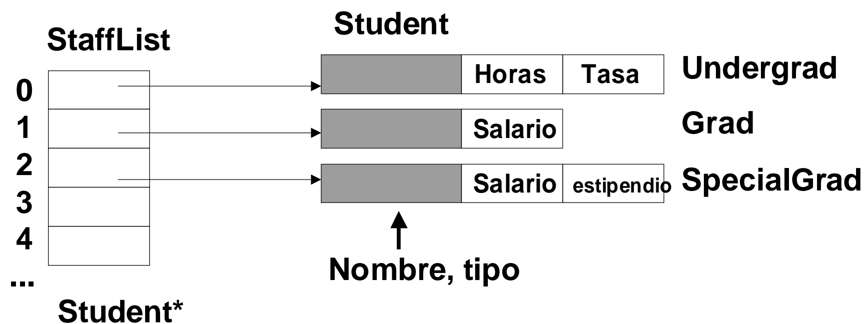
Clase ResearchProject

```
class RProject {
public:
   explicit RProject(int Size);          // Constructor
   void addStudent(Student* RMember);    // Agrega puntero
   void listPay();          // Enumera estudiantes, refleja proyecto
private:
   Student** StaffList;     // Array de punteros a Student
   int StaffSize;           // Tamaño máximo de staff
   int count;               // Tamaño real de staff
};

// No puede almacenar el objeto Student directamente; no tendría
// datos adicionales de las clases derivadas

// Debería tener un destructor, etc. si fuese para uso real
// Rproject 'tiene un' array Student
```

Clase ResearchProject, p.2

```
RProject::RProject(int Size) {
   StaffSize= Size;
   StaffList= new Student*[StaffSize];
   count= 0;}

void RProject::addStudent(Student* RMember){
   StaffList[count++]= RMember;
   return; }

void RProject::listPay() {
   cout << "Project staff " << endl;
   for (int i= 0; i < count ; i++)
     StaffList[i]->GetData();    // (*StaffList[i]).GetData()
   return; }

// La salida es la misma que antes:
// Lista de estudiantes con aportación específica al proyecto
```

Clase ResearchProject



C++ define dinámicamente los punteros de Student en Undergrad, Grad, SpecGrad para obtener la aportación de cada uno
Si almacenásemos los objetos Student en StaffList y no como punteros, sólo tendríamos los datos básicos de la clase Student

Programa main

```
int main( ) {
// Define 3 estudiantes (sin cambios desde la
   última vez) Undergrad Ferd("Ferd", "Smith",
   8.0, 12.00); Ferd.GetData();
   Grad Ann(”Ann", "Brown", 1500.00);
   Ann.GetData();
   SpecGrad Mary("Mary", "Barrett", 2000.00);
   Mary.GetData();
   cout << endl;

// Agrega 3 estudiantes al proyecto y muestra su aportación
   RProject CEE1(5);
   CEE1.addStudent(&Ferd);
   CEE1.addStudent(&Ann);
   CEE1.addStudent(&Mary);
   CEE1.listPay();
   return 0;}        // La salida es la lista de estudiantes y
                        su aportación
```

Otros elementos

- **Las excepciones son casi las mismas que en Java**
  - **Intentar, arrojar, detectar**
- **Biblioteca estándar de plantilla (STL) con vectores, listas, pilas, Parecido a Java**
  - **No puede heredar de clases STL**
- **Múltiple herencia admitida**
- **C tiene dos tipos de cadenas (con conversiones)**
  - **Arrays terminados en *null* (antiguos)**
  - **Clase *String***
- ***Arrays*: sólo una ubicación de memoria (ref.)**
  - **El tamaño del *array* se pasa como argumento indep.**
  - **Muy problemático (“sobrecarga en búfer”)**
- **Recursión admitida, como en Java**