Unidad 11
Sobrecarga de Operadores

La sobrecarga de operadores es uno de los mecanismos que nos permite ampliar las capacidades de los lenguajes de programación orientados a objetos. En C++, la declaración y definición de una sobrecarga de operador es muy similar a la declaración y definición de una función cualquiera.
Los símbolos como +,-.= son llamados operadores, Estos operadores son definidos para tipos de datos como int o double.

Las clases generadas para este tipo de sobrecarga generalmente son llamadas complejas.
Una vez definida la clase Compleja, se puede definir la suma, resta, multiplicación y división. Generalmente tendrías algo como esto:

```
Compleja c1(1, 0), c2(0, 1);
Compleja c3 = c1 + c2;
```

```
// Titulo..: programa sobrecarga01.cpp
// Objetivo: demostración de sobrecarga de operadores
#include <iostream>
using namespace std;
class Pareja {
public:
   double a, b;
   // constructor parametrizado
   Pareja(const double a,const double b)
   {
      this->a = a;
      this->b = b;
   }
};

// Sobrecarga del operador +
Pareja& operator +(const Pareja &p1,const Pareja &p2)
{
  return *(new Pareja(p1.a + p2.a, p1.b + p2.b) );
}

int main()
{
   Pareja A(50, 75 );
   Pareja B(150, 175 );
   Pareja C = A + B;

   cout << "A = " << A.a << ',' << A.b << "\n";
   cout << "B = " << B.a << ',' << B.b << "\n";
   cout << "C = " << C.a << ',' << C.b << "\n";
   return 0;
}
```

Sobrecarga permitida de operadores

En C++ no es posible sobrecargar todos los operadores, pero al menos la mayoría de ellos sí. Los operadores que no se pueden sobrecargar son: operadores de directivas de procesador #, ## ; Selector directo de componente . ; operador para valores por defecto de funciones de clase : ; Operador de acceso a ámbito :: ; Operador de indirección de puntero-a-miembro .* ; Condicional ternario ?; sizeof y typeid.

Sobrecargas permitidas:

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Unidad 12
Excepciones

Las excepciones son en realidad errores durante la ejecución. Si uno de esos errores se produce y no implementamos el manejo de excepciones, el programa sencillamente terminará. Es muy probable que si hay procesos abiertos no se guarde el contenido de los buffers, ni se cierren, además ciertos objetos no serán destruidos, y se producirán fugas de memoria.

En programas pequeños podemos prever las situaciones en que se pueden producir excepciones y evitarlos. Las excepciones más habituales son las de peticiones de memoria fallidas

Veamos este ejemplo, en el que intentamos crear un *array* de cien millones de enteros:

```
#include <iostream>
using namespace std;
int main()
 {
   int *x = NULL;
   int y = 100000000;
      x = new int[y];
  x [10] = 0;
   cout << "Puntero: " << (void *) x << endl;
   delete[] x;
      return 0;
}
```

El sistema operativo se quejará, y el programa terminará en el momento que intentamos asignar un valor a un elemento del *array*.

Podemos intentar evitar este error, comprobando el valor del puntero después del new:

```
#include <iostream>
using namespace std;
int main()
{
  int *x = 0;
  int y = 100000000;
    x = new int[y];
  if(x) {    x[10] = 0;    cout << "Puntero: " <<
(void *) x << endl;
    delete[] x;  } else {    cout << "Memoria
insuficiente." << endl;
  }
return 0;}
```

Pero esto tampoco funcionará, ya que es al procesar la sentencia que contiene el operador new cuando se produce la excepción. Sólo nos queda evitar peticiones de memoria que puedan fallar, pero eso no es previsible.

Sin embargo, C++ proporciona un mecanismo más potente para detectar errores de ejecución: las excepciones. Para ello disponemos de tres palabras reservadas extra: try, catch y throw, veamos un ejemplo:

```
#include <iostream>
using namespace std;
int main() {
 int *x;
 int y = 100000000;
try
{
x = new int[y];
 x[0] = 10;
cout <<"Puntero:"<<(void *)x<< endl;      delete[]x;
}
catch(std::bad_alloc&)
{
cout << "Memoria insuficiente" << endl;
}
return 0;
}
```