μIntroducción a `C++` a partir de `C`

Procesado de Audio y Voz (PAV)

Antonio Bonafonte, UPC

14 de febrero de 2017

`C++` es un lenguaje utilizado con frecuencia en el desarrollo de aplicaciones de procesado multimedia. `C++` es una extensión del `C` que facilita la programación *orientada a objetos*, al definir conceptos como clases, herencia, etc. Además, añade características al lenguaje, y la librería estándar de `C++` que lo hace más cómodo que el lenguaje `C`, conservando todas las características del `C`.

En Linux, para compilar o *lincar* suele utilizarse `g++`, con la misma sintaxis que `gcc`.

Existen muchos libros y recursos en internet, de distintas extensión, para aprender el lenguaje. Por ejemplo, en `http://www.cplusplus.com` hay `tutoriales` y los manuales de referencia.

Este documento intenta condensar los conceptos mínimos que se utilizan en la asignatura, que deberá complementarse con el manual de referencia de las funciones que se utilicen.

1. Extensión del lenguaje

- Se añade el tipo `bool`, y los operadores lógicos `and` y `or`.
- Pueden añadirse comentarios mediante "`//`" (el resto de la línea es comentario)
- Se define `class`, una generalización de los `struct` pero que permite definición de tipos públicos y privados, herencia (definir una clase basada en otra), etc. (véase sección 2).
- Las variables pueden definirse en cualquier lugar del código. También puede, y es recomendable, declarar variables dentro de un *bloque*, creándose al entrar al bloque, y liberándose al dejarlo.

```
for (int i = 1; i < 100; i++) {
  float x;
  ...
}
// here 'x' and 'i' are no longer declared
```

- El lenguaje incluye los operadores `new`, `delete`, y `delete[]` para reservar memoria dinámica, sustituyendo a las funciones de la librería de C, `malloc, free`, etc.

```
cout << "Enter size: ";
int n;
cin >> n;
```

```
int *x = new int [n];
int *y = new int;
Sound *snd = new Sound;

...
delete[] x; //delete vector
delete y;
delete snd;
```

- Un inciso sobre la librería de entrada/salida: `cin`, `cout` y `cerr` equivalen a `stdin`, `stdout` y `stderr` de `C`, es decir, entrada estándar y salida estándar, y salida estándar de errores. Los operadores `<<` y `>>` son más cómodos de usar y pueden concatenarse:

```
cout << "Enter hours minutes and seconds: ";

int h, m, s;
cin >> h >> m >> s;
cout << "Total seconds: " << s + (m  + h * 60 ) * 60 << '\n';
```

- Pueden definirse funciones con mismo nombre y distinto tipo o número de argumentos. El compilador elegirá la adecuada.

```
int MAX(int a, int b) {
  if (a >= b) return a;
  else return b;
}

int MAX(int a, int b, int c) {
  return MAX(MAX(a,b),c);
}
```

Incluso pueden ponerse valores por *defecto*, puede ser uno o más de uno, empezando por el final:

```
void sort_data(float *data, int size, bool ascending_order=true);

main() {
  float x[5] = {5,1,4,2,3};
  sort_data(x, 5, false);
  sort_data(x, 5);
}
```

En ocasiones verá `extern "C"` para indicar que la etiqueta para la función del compilador es la que se generaría en `C` (en este caso, como en `C`, no puede funciones con el mismo nombre y distintos argumentos). De esta forma puede *enlazarse* código objeto compilado con `C` y con `C++`

- La función `MAX(.)` anterior se ha definido con argumentos enteros. ¿Cuál cree que sería el resultado de la siguiente instrucción?

```
float r = MAX(3.1416, 2.7183, 1.4142);
```

Podríamos definir otras funciones utilizando argumentos `float`, y `double`, etc., pero es muy redundante. En `C++` se pueden definir plantillas, *templates*, que son funciones o clases que dependen de uno o varios tipos genéricos. Al utilizarse, se *instancia* con el tipo adecuado.

El ejemplo siguiente, se define para un tipo genérico, que hemos llamado `T`, y que en este ejemplo, según vemos en la línea 3, deberá cumplir que tenga definido el operador `>=`.

```
1  template <typename T>
2  T MAX(T a, T b) {
3    if (a >= b) return a;
4    else return b;
5  }
6
7  template <typename T>
8  T MAX(T a, T b, T c) {
9    return MAX(MAX(a,b),c);
10 }
11
12 int main() {
13   cout << MAX(3.1416, 2.7183, 1.4142) << '\n';
14   cout << MAX<int>(3.1416, 2.7183, 1.4142) << '\n';
15 }
16
```

Como vemos, se puede indicar explícitamente el tipo con el que se *instancia* la función (segunda llamada, que devolverá 3), o el propio compilador elige la más adecuada (primera llamada, que devolverá 3.1416).

- Notación para el paso por referencia. Al llamar a una función los argumentos pueden pasarse *por valor* o *por referencia*.

  Por valor, significa que se realiza una copia antes de llamar a la función. Y si la función lo cambia, no tiene efecto fuera de la función. Cuando en `C` pasamos una variable, se hace por valor. Para pasar por referencia, se debe pasar la dirección de la variable, *el puntero*. De esta forma, la función puede cambiar el valor en la dirección indicada por *el puntero*. Puede probar el siguiente ejemplo:

```
void Power_4_and_8(float x, float *x4, float *x8) {
  x  =  x * x;
  *x4 =  x * x;
  *x8 = *x4 * *x4;
}

int main() {
  float v = 2, v2, v4;
  power_4_and_8(v, &v4, &v8);
  cout << v << ' ' << v4 << ' ' << v8 << '\n';
}
```

En C++ puede indicarse a la función que es por referencia, sin necesidad de utilizar la *notación de punteros*. Para ello se añade en la declaración el carácter '&':

```
void Power_4_and_8(float x, float &x4, float &x8) {
  x  = x * x;
  x4 = x * x;
  x8 = x4 * x4;
}

int main() {
  float v = 2, v2, v4;
  power_4_and_8(v, &v4, &v8);
  cout << v << ' ' << v4 << ' ' << v8 << '\n';
}
```

Cuando el argumento a utilizar por la función es un tipo de datos *grande* (que ocupa mucha memoria), el paso por valor no es muy eficiente ya que siempre se hace una copia. Por ello, en C no se permite para vectores: el nombre del vector se sustituye por la dirección (puntero) del primer elemento. Si un argumento se pasa por referencia por razones de eficiencia, pero no se va a modificar en la función, se indica mediante la declaración `const`. Esto *documenta* el código y evita errores/ineficiencias por asumir que se modifica o no en el interior. La declaración `const` también es válido para el paso por referencia en C++. El siguiente ejemplo muestra dos declaraciones, una utilizando punteros, otra con referencia de C++, que se pasa por referencia el `struct VAD_DATA` , pero que no se puede modificar dentro de la función.

```
typedef struct {
  ...
} VAD_DATA;

void print_vad_data_v1(const VAD_DATA * data) {data->rate = 8000;}
void print_vad_data_v2(const VAD_DATA & data) {data.rate = 8000;}
```

- En C++ se pueden definir *espacios de nombres*, para evitar conflictos entre varias definiciones de clases con mismo nombre. En proyectos complejos podemos encontrarnos varias declaraciones de clases como `vector`, `image`, `event`, etc. Podemos indicar que las declaraciones forman parte de un espacio de nombre mediante el *keyword* `namespace`:

```
namespace upc {
  typedef struct {
    ...
  } VAD_DATA;

  void print_vad_data_v1(const VAD_DATA * data);
  void print_vad_data_v2(const VAD_DATA & data);
}
```

Y luego para usarlo:

```
upc::VAD_DATA data;
upc::print_vad_data_v1(&data);
```

```
    upc::print_vad_data_v2(data);
```

O bien, indicando que se va a usar ese `namespace`. Por ejemplo, todas las funciones y clases de la librería estándar están en el espacio de nombres `std`. Y es frecuente en muchos programas encontrar:

```
    using namespace std;
```

y así evitar expresiones como `std::cout` o `std::string`.

2. Clases C++

La programación orientada a objetos es una forma de programación en las que se definen unos *objetos*, que suelen ser *datos* y además, el *interfaz* para acceder o manipular los datos. El tener un interfaz bien definido, sin necesidad de saber cómo se implementa ni se almacena simplifica la utilización de las funcionalidades y también su mantenimiento o mejora.

La programación orientada a objetos puede realizarse en *C*. Por ejemplo, la librería `stdio` declara el objeto `FILE` y las operaciones que soporta. No sabemos cómo se implementa el `FILE`, pero podemos declarar un *objeto* de este tipo, y usar las funciones definidas (*el interfaz*) para operar con él. Veamos como podríamos definir un tipo de datos en `C` para definir objetos *números complejos*:

```
struct Complex {
  float r, i;
};

void complex_set(Complex *c, float re, float im) {
  c->r = re;
  c->i = im;
}

float complex_abs(const Complex *c) {
  return sqrt(c->r * c->r + c->i * c->i);
}

void complex_add(Complex *c, const Complex *c1, const Complex *c2) {
  c->r = c1->r + c2->r;
  c->i = c1->i + c2->i;
}

...

int main() {
  Complex v1, v2, *ptv2 = &v2;
  complex_set(&v1, 1, -1);  //v1.r = 1; v1.i = -1;
  complex_set(ptv2, 2,  1); //ptv2->r = 2 or v2.r = 2, etc.
  cout << complex_abs(&v1) << ' ' << complex_abs(ptv2) << '\n';
}
```

En C++ se generaliza el `struct` en clases, que permite expresar más gráficamente las funciones *del interfaz* de la clase, e indicar si las variables o *miembros* son públicas o privadas, es decir, si puede o no accederse a ellas desde fuera de las *funciones de la clase*. Denegando el acceso directo a los datos, más adelante podrá cambiarse como se representa sin que tengan consecuencias en los programas que utilizan estos objetos. El tipo `class` permite definir, además de datos, funciones para que el programa acceda o utilice esos datos:

```
class Complex {
  public:
  float r, i;
  void set(float re, float im) {r = re; i = im;}
};

int main() {
  Complex c, *ptc = &c;
  c.r = 1;
  c.i = 1;
  cout << c.r << '+' << c.i << "j\n";
  c.set(-1,0);
  cout << ptc->r << '+' << ptc->i << "j\n";
}
```

Las funciones que definen *el interfaz* se incluyen dentro de la clase. La llamada aplica a un objeto concreto así que no hay que indicar el objeto como argumento: accedemos a la variables internas del objeto para el que se llama la función:

Puede indicase qué funciones y variables pueden accederse directamente desde fuera de la clase (`public`). Por defecto, todo es `private`. Con la siguiente definición el compilador daría un error si desde fuera de la clase accedemos directamente a `c.r` u otros miembros privados.

```
class Complex {
  private:
  float r, i;
  public:
  void set(float re, float im) {r = re; i = im;}
  float abs() {return sqrt(r * r + i * i);}
};

int main() {
  Complex c1, v[2], *ptc = v+1;
  c1.set(0,1);
  v[0].set(3,  -4);
  v[1].set(1, -1);
  cout << c1.abs() << ' ' << v[0].abs() << ' ' << ptc->abs() << '\n';
}

//Result: 1 5 1.41421
```

Otras características de las clases en C++:

- Hay unas funciones *especiales*, los constructores, que se ejecutan cuando se declara la clase. Su nombre es el mismo que la propia clase:

```
class Complex {
  ...
```

```
    Complex(float re = 0, float im = 0) {r = re; i = im;}
    Complex(const Complex &c) {r = c.r; i = c.i;}
  };

  int main() {
    Complex v1(1,-1);
    Complex v2(v1);
  }
```

Cuando una de las variables miembros de la clase es otra clase, pueden inicializarse directamente a un valor poniendo ':' antes de la implementación. Puede usarse esta sintaxis en el ejemplo anterior:

```
  Complex(float re = 0, float im = 0) : r(re), i(im) {}
```

Esta forma es más eficiente, y es la única posible cuando todos los constructores de la variable miembro requieran argumentos.

- Análogamente, se define el destructor, sin argumentos, cuyo nombre es el de la clase precedido por '~' que se ejecuta cuando se libera la variable. Suele utilizarse para cerrar ficheros y liberar memoria dinámica:

```
  class VectorFlt {
    int size;
    float *x;

    public:
    VectorFlt(int n): size(n) {x = new float[size];}
    ~VectorFlt() {delete[] x;}
    ...
 };
```

- Así como la declaración `const` nos indica las variables pasadas por referencia que no se cambian, la palabra `const` después de una función miembro de una clase, nos indica que las variables de ese objeto no se cambian. La función `abs` de la clase `Complex` no modifica las variables miembro `r` ni `i`:

```
class Complex {
  ...
  void set(float re, float im) {r = re; i = im;}
  float abs() const {return sqrt(r * r + i * i);}
};
```

- Se pueden utilizar como nombres de las funciones los *operadores* típicos. Por ejemplo, para la clase `Complex` tendría sentido poder definir funciones `'+'`, `'-'`, `'*'`, `'+='`, `'=='`, `'!='`, etc. Una clase que fuera una base de datos indexada por un identificador, tendría sentido utilizar para acceder a ella '`database[id]`'. O una clase que implemente un módulo *procesador*, por ejemplo la `FFT`, podría ser útil llamarla usando el operador '`FFT(x)`': aunque parece una función, podría ser una clase,

con variables miembro para tablas precalculadas. Otros operadores que pueden definirse: `'++'`, `'--'`, `'*='`, `'&'`, `'&&'`, `'<<'`, etc. Veamos un ejemplo con la clase `Complex`:

```
class Complex {
  ...
  bool operator==(const Complex &c) const {
    if (c.r == r and c.i == i) return true;
    else return false;
  }
  const Complex & operator=(const Complex &c) {
    r = c.r;
    i = c.i;
    return *this;
  }
};

int main() {
  Complex c1(1,1), c2;

  c2 = c1; //equivalent to: c2.operator=(c1)

  if (c1 == c2) cout << "Equal values!\n";
  else cout << "Different values\n";

  //equivalent to c1.operator==(c2);
}
```

- En `C++` pueden definirse clases que se basan en otra, *herencia*:

```
class Rectangle: Public Polygon {
  //Rectangle has all the variable members and methods
  //of Polygon, plus the ones defined here:
  ...
}
```

Puede definirse polimorfismo, etc.

3. Librería estándar

Una parte importante de `C++` es su librería estándar. Contiene entre otros: *contenedores*, funciones para entrada/salida (ficheros, etc.)., la clase `string`, para almacenar y operar con cadenas de caracteres, algunos algoritmos como `sort`, `copy`, `find`, etc. Otras clases útiles son `complex`, `pair`, `tuple`, `regex`, etc.

3.1. Contenedores

`C++` ofrece una implementación unificada de los contenedores más utilizados en programación: vectores, listas, colas, pilas, *mapas*, etc. Están implementados mediante `templates`, de forma que puede definirse un contenedor de objetos definidos en una clase. Los distintos tipos de datos ofrecen ventajas respecto a otros. Por ejemplo, un vector es muy fácil de acceder (`x[i]`, acceder directamente a posición `i`), pero es muy costoso eliminar o insertar valores, pues se han de desplazar todos los de después.

Una ventaja de los contenedores es que tienen funciones muy sencillas para reservar memoria de forma dinámica, y que los destructores de estas clase se encargan de liberarla.

El número de funciones de cada contenedor es limitado, así que es recomendable y factible realizar una lectura rápida de un manual de referencia[1]. Los siguientes ejemplos quieren ilustrar las funciones más comunes:

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
  //declara un vector de int, sin tamaño
  vector<int> x;
  if (x.empty())
    cout << "x is an empty vector\n";


  //cambia tamaño a 10
  x.resize(10);

  //se accede como vectores en C
  for (int i=0; i < x.size(); i++) x[i] = i*10;
  for (int i=0; i < x.size(); i++) cout << x[i] << ' ';
  cout << '\n';


  //se añaden valores al fina: el vector tendrá tamaño 12
  x.push_back(100);
  x.push_back(110);
  cout << "Size: " << x.size() << '\n';
  cout << "Last value: " << x.back() << '\n';

  //se copia un vector en otro
  vector<int> y = x;

  //borrar un vector; queda sin tamaño
  x.clear();
  if (x.empty())
    cout << "x is an empty vector\n";
  if (!y.empty())
    cout << "Size of y: " << y.size() << '\n';

  //En vectores C, se pueden usar punteros,
  float xc[10];
  float *p;
  for (p = xc; p != xc + 10; p++)
    *p = 0;

  //En c++, la generalización, iteradores, que pueden usarse para otros
    contenedores
  vector<int>::iterator it;
  for (it = y.begin(); it != y.end(); it++) {
    *it /= 10;
    cout << *it << ' ';
  }
  cout << '\n';


  return 0;
}
```

[1] Veáse por ejemplo: `http://www.cplusplus.com/reference/vector/vector/`

```
#if 0
Resultados:

x is an empty vector
0 10 20 30 40 50 60 70 80 90
Size: 12
Last value: 110
x is an empty vector
Size of y: 12
0 1 2 3 4 5 6 7 8 9 10 11
#endif
```

vector.cpp

El concepto de *iteradores*, que puede verse en el ejemplo es muy importante pues generaliza la idea y el uso de punteros en vectores a otros tipos de contenedores. Mediante ellos, se puede iterar sobre cualquier contenedor. Si `Container` es un tipo de datos contenedor, e `iterator` es un *iterador* adecuado para este contenedor, será válido:

```
  for (iterator =  Container.begin();
       iterator != Container.end();
       ++iterator) {
         do_something(*iterator);
  }
```

Como vemos, la función `begin()` *apunta* el primer elemento, y la función `end()` apunta a un elemento *ficticio*, ya fuera del contenedor.

Otro ejemplo usando listas, y mapas:

```
#include <iostream>
#include <list>
#include <map>
using namespace std;

int main() {
  // **** LISTA ***

  //declara una list
  list<int> x;
  if (x.empty())
    cout << "x is an empty list\n";

  //añadir 20 elementos
  for (int i=0; i<10; i++) {
    x.push_back(i);
    x.push_back(0);
  }

  //no está implementado x[i] para evitar tentaciones:
  //en una lista ésta es una operación costosa, pues hay que ir de elemento
     en elemento.

  // Usaremos iteradores para acceder elementos
  // const_iterator indica que no modificaremos; no sería posible *it = 4;
  list<int>::const_iterator itc;
  for (itc = x.begin(); itc != x.end(); itc++) cout << *itc << ' ';
  cout << '\n';

  //Borrar elementos con valor cero:
```

```
  list<int>::iterator it;
  for (it = x.begin(); it != x.end();) {
    if (*it == 0) it = x.erase(it);
    else ++it;
  }
  for (itc = x.begin(); itc != x.end(); itc++) cout << *itc << ' ';
  cout << '\n';

  // **** MAPA, the int_ID to float ***

  map<int, float> m;

  m[100000] = 2.7183;
  m[401962] = 1.4142;
  m[0]      = 3.1416;

  map<int,float>::iterator m_it;

  cout << "Map size: " << m.size() << '\n';
  for (m_it = m.begin(); m_it != m.end(); ++m_it) {
    cout << m_it->first << " => " << m_it->second << '\n';
  }

  int keys[4] = {7, 100000, 13, 0};
  for (int i=0; i<4; i++) {
    m_it = m.find(keys[i]);
    if (m_it != m.end())
      cout << keys[i] << " => " << m_it->second << '\n';
    else
      cout << keys[i] << " not found\n";
  }

  return 0;
}


#if 0
Resultados:

x is an empty list
0 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8 0 9 0
1 2 3 4 5 6 7 8 9
Map size: 3
0 => 3.1416
100000 => 2.7183
401962 => 1.4142

7 not found
100000 => 2.7183
13 not found
0 => 3.1416

#endif
```

containers.cpp

3.2. Cadenas de caracteres: clase `strings`

Las cadenas de caracteres o texto son un tipo muy importante en la programación. En `C`, se implementan mediante *vectores* de caracteres, cuya longitud de indica mediante carácter nulo al final, y con funciones específicas para reservar memoria, copiar, conocer longitud,

etc. En C++ existe la clase `string`, mucho más amigable. Un ejemplo:

```
#include <string>
#include <iostream>
using namespace std;
int main() {

  //hello world
  string h = "hello";
  string w = "world";

  string s = h + ' ' + w;
  cout << s << '\n';
  if(! s.empty()) {
    cout << s.size() << "\n\n";
  }

  //Read words and delete vowels
  int num_words = 0;
  while (cin >> s) {
    string::size_type pos = s.find_first_of("aeiou");
    while (pos != string::npos)  {
      s[pos]='.';
      pos=s.find_first_of("aeiou",pos+1);
    }
    cout << ++num_words << '\t' << s << endl;
  }
}

#if 0
Results:
echo "no por mucho madrugar amanece más temprano" | ./string
hello world
11

1 n.
2 p.r
3 m.ch.
4 m.dr.g.r
5 .m.n.c.
6 más
7 t.mpr.n.
#endif
```

string.cpp

Otro ejemplo, combinado con la clase `map`, para crear una tabla que codifica cada palabra distinta con un número:

```
#include <map>
#include <string>
#include <iostream>
using namespace std;


int main() {
  map<string, int> dict;
  string s;
  //Leer de palabra en palabra, desde entrada estándar
  //y crear diccionario

  int id = 0;
  while (cin >> s) {
    if (dict.find(s) == dict.end())
```

```
      dict[s] = ++id;
  }

  //Mostrar diccionario
  map<string,int>::iterator idict;
  for (idict = dict.begin(); idict != dict.end(); ++idict)
    cout << idict->first << ": " << idict->second << '\n';
  cout << "------------------------\n";

  //Buscar palabra en diccionario
  s = "isabel";
  idict = dict.find(s);
  if (idict == dict.end())
    cout << s << " not found in dict\n";
  else
    cout << s << ": " << dict[s] << '\n';
}

#if 0
Results:

$ echo tanto monta monta tanto isabel como fernando | ./string_map
como: 4
fernando: 5
isabel: 3
monta: 2
tanto: 1
------------------------
isabel: 3
#endif
```

string–map.cpp

3.3. I/O

A lo largo de los ejemplos hemos utilizado `cout <<` y `cin>>` para escritura y lectura por la salida estándar, sustituyendo a las funciones de `C printf, scanf`. Estas mismas funciones puede aplicarse para escritura y lectura de ficheros de texto.

```
  //mostrar un fichero, añadiendo numero de lineas
  //usar getline para conservar fin de linea.
  #include <iostream>
  #include <fstream>

  int main(int argc, char const *argv[]) {
    if (argc != 2) {
       cerr << argv[0] << " input_file\n";
       return 1;
    }
    istream is(argv[1]); //open file
    if (!is.good()) {
      cerr << "Error opening file: " << argv[1] << '\n';
      return 2;
    }
    string s;
    int nline = 0;
    while (getline(is, s)){
      cout << ++nline << '\t' << s << '\n';
    }
  }
```

La clase `stream` también incluye funciones para lectura de ficheros binarios, `read, write, seek, eof, close`, etc., similares a las de `C`, `fread, fwrite, fseek, feof, fclose`.

Comentario

Es dificil describir un lenguaje de programación, para que pueda leerse y utilizarse en unas pocas horas. Existen muchos recursos en internet, pero al ser dirigidos a un público amplio, creo que son largos, o no describen todo lo que se necesita. Mi intención es que sea *relativamente* rápido de leer para nuestros alumnos, que saben un poquito de `C`, y son listos y capaces. Lógicamente, tendréis que ir a los manuales de referencia de la librería estándar para saber que funciones se disponen: no son tantas, es rápido mirar las funciones de los `string` u otra clase. La librería tiene otras definiciones, como `complex`, `pair`, etc. Y hay librerías que no son de la `stdc++` pero son muy útiles, como la `boost`, que contiene librería que quizás sean parte de la estándar en las siguientes versiones, con funcionalidades como expresiones regulares, programación concurrente, acceso a directorios, gráfos, etc.

Agradeceré cualquier comentario: ¿te ha servido? ¿quitarías una parte? ¿no se entiende algo? ¿Has encontrado errores?