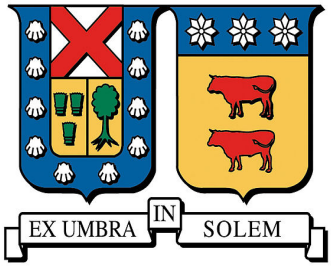


Universidad Técnica Federico Santa María

Departamento de Informática

Estructura de Datos

Python → C++

Autor:
Renato Sanhueza Ulsen
Correcciones:
Ariel Sanhueza Román

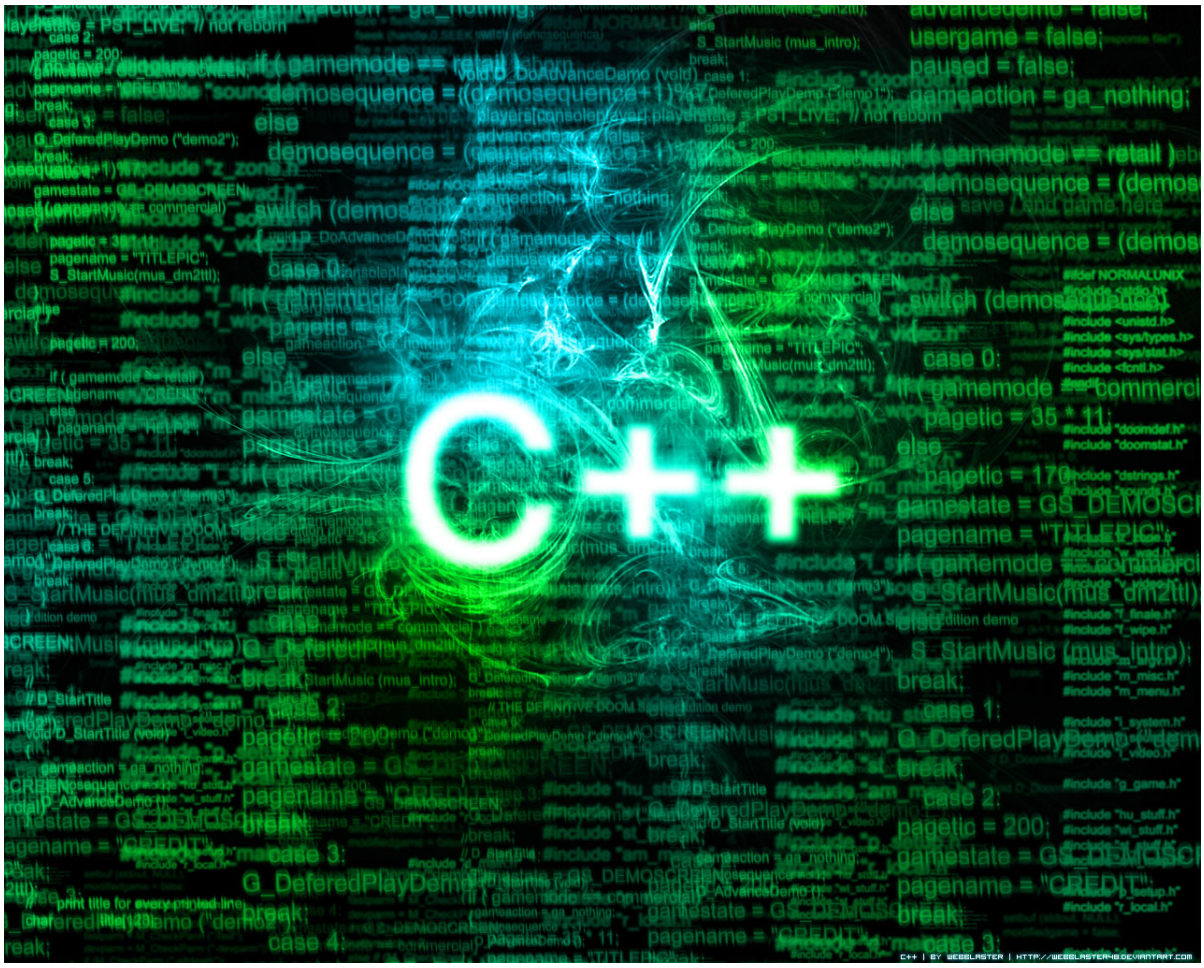
Ingeniería Civil Informática

Valparaíso - 12 de marzo de 2014

Índice

1. Introducción 1
2. Compilación 3
3. Estructura básica de un programa 9
4. Tipos de Variables 10
5. Entrada y Salida de datos 11
6. Sentencias condicionales 12
7. Iteraciones 14
8. Arreglos 17
9. Funciones 20
10. Punteros 22
11. Paso por referencia 24
12. Arreglos dinámicos 25
13. Estamos trabajando para usted. 27

1. Introducción



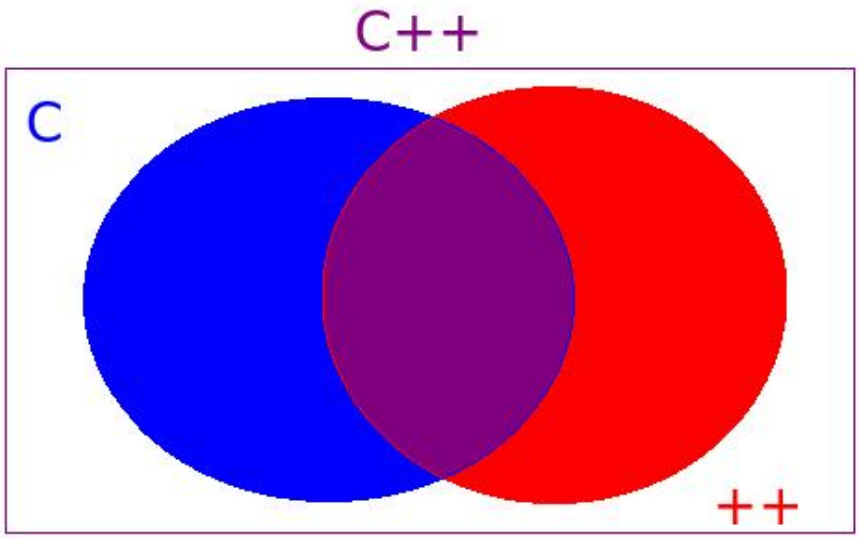
"Un paradigma de programación es una propuesta tecnológica que es adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que unívocamente trata de resolver uno o varios problemas claramente delimitados. La resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software."[1]
Es probable que ustedes ya hayan creado programas con el lenguaje C, este lenguaje pertenece al paradigma imperativo y los programas consisten en un conjunto de instrucciones que le indican al computador como realizar una determinada tarea. Este paradigma es considerado el más común, incluso el lenguaje de máquina está escrito de forma imperativa.
Este semestre en el ramo Estructura de Datos usaremos un nuevo lenguaje de programación llamado C++ el cual no es imperativo. C++ obedece al paradigma de la orientación a objetos (el más usado hoy en día).Aprender un nuevo paradigma de programación toma tiempo y no es un objetivo de este ramo, es por esta razón que no usaremos Orientación a Objetos. Ustedes se preguntarán ¿Como programaremos en C++ si no usaremos Orientación a Objetos? La respuesta a esta pregunta surgirá naturalmente una vez expliquemos la relacion que existe entre C y C++.

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie. Posteriormente C++ fue diseñada a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido [2].
En la siguiente figura podemos apreciar que C++ abarca 3 regiones claramente delimitadas que explicaremos a continuación:

- En la región azul estan todos los elementos de C puro que se conservan intactos en C++.

[1] http://es.wikipedia.org/wiki/Paradigma_de_programaci%C3%B3n
[2] http://es.wikipedia.org/wiki/C%2B%2B

- En la región roja estan todos los elementos puros de C++ entre los cuales encontramos las herramientas para la Orientacion a Objetos.
- En la región morada estan las “mejoras” a elementos ya existentes en el lenguaje C.



A estas alturas es probable que algunos ya se hayan dado cuenta que programaremos utilizando solo 2 de las 3 áreas anteriormente mencionadas, la azul y la morada. El objetivo de este apunte es que todos puedan aprender a programar en C++ de forma imperativa. Una vez se aprende a programar en cualquier lenguaje, aprender uno nuevo es fácil si mantenemos el paradigma. En programación se le enseñó a programar en Python de forma imperativa, aún cuando este tambien es Orientado a Objetos(igual que C++) así que no se asuste, no está haciendo nada nuevo.
En este apunte se repasarán los conceptos básicos de la programación con el uso extensivo de ejemplos ya que la mejor forma de aprender a programar es justamente programando. Si está recién aprendiendo reescriba los programas en su computador y mire como funcionan. Si ya tiene una idea puede intentar mejorarlos a modo de ejercicio ya que los programas no están escritos con el fin de ser eficientes sino mostrar de la mejor manera como funcionan las distintas herramientas que usaremos para programar.

2. Compilación

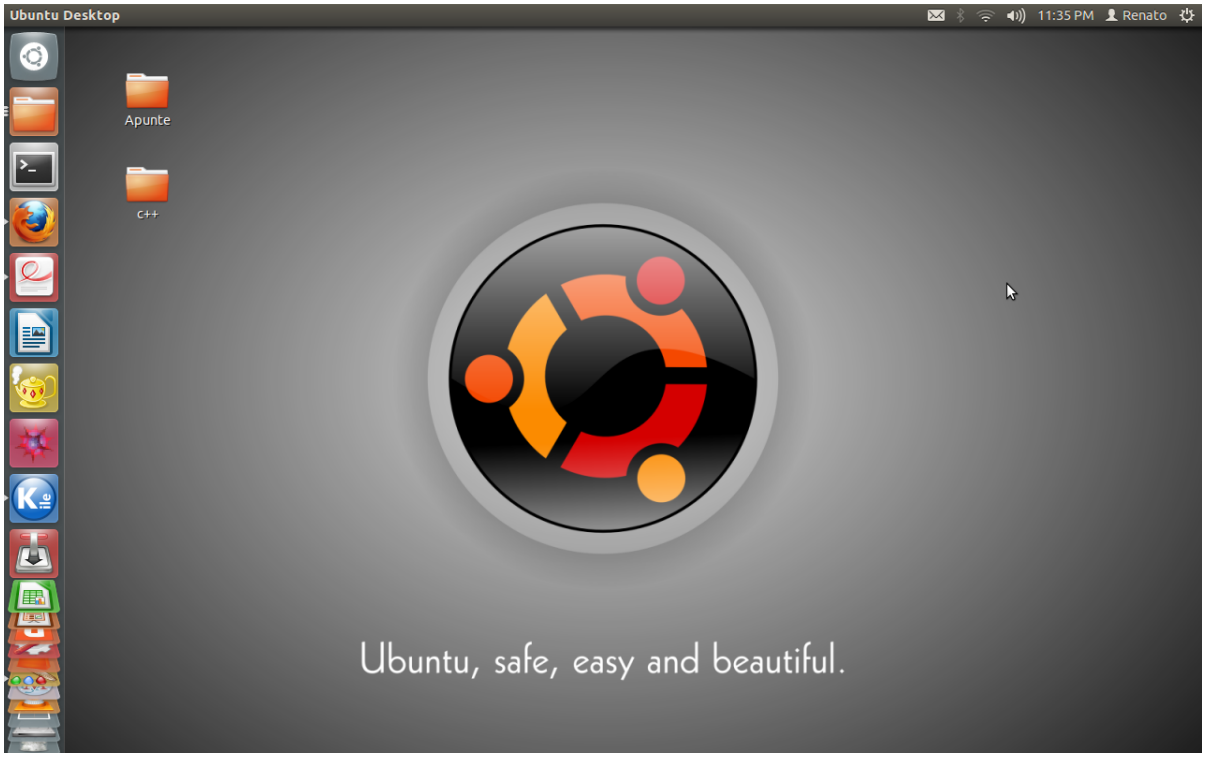
C++ es un lenguaje compilado, es decir, requiere que un programa(compilador) transforme el programa escrito por nosotros en otro equivalente que pueda ser interpretado por el computador(lenguaje de máquina). Esta la primera diferencia que notaremos entre Python y C++ ya que Python es un lenguaje interpretado donde el intérprete traduce el programa solo en la medida que sea necesario(generalmente instrucción por instrucción) y no guardan un resultado de tal traducción.

Ahora se explicará paso por paso, como compilar un programa una vez terminado, através de un entorno de consola. Recomendamos encarecidamente que instale alguna distribución de Linux para hacer las tareas de Estructura de Datos ya que si bien C++ es un lenguaje que puede ser compilado en cualquier sistema operativo, en Windows existen librerias que no estan disponibles en Linux y viceversa, además, las tareas serán probadas en el entorno linux ofrecido por el Laboratorio de Computación (a.k.a LabComp) por lo que si su tarea no compila ahí, se considerará como mala.

En fin, ha llegado la hora de poner manos a la obra. Para compilar nuestros programas necesitaremos una terminal, que en pocas palabras es una interfaz serial para comunicarse con un computador, un teclado para entrada de datos y una pantalla para exhibición de únicamente caracteres alfanuméricos (sin gráficos).[3]
Datos útiles al momento de usar la terminal:

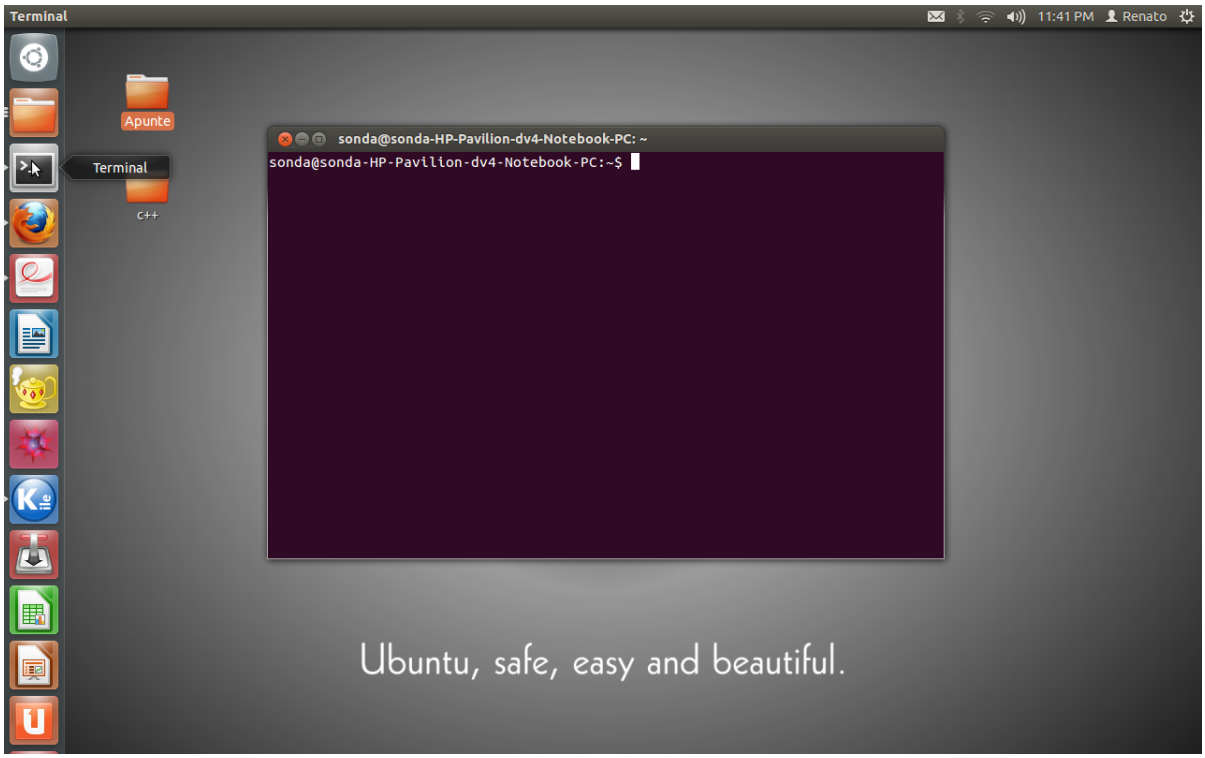
- Cada vez que abrimos la consola, esta se encuentra ubicada en la carpeta Home.
- Los comandos necesarios para manejarse decentemente en la consola son ls, mkdir, cd, pwd, rm.

Hechemos una primera mirada a nuestro escritorio.

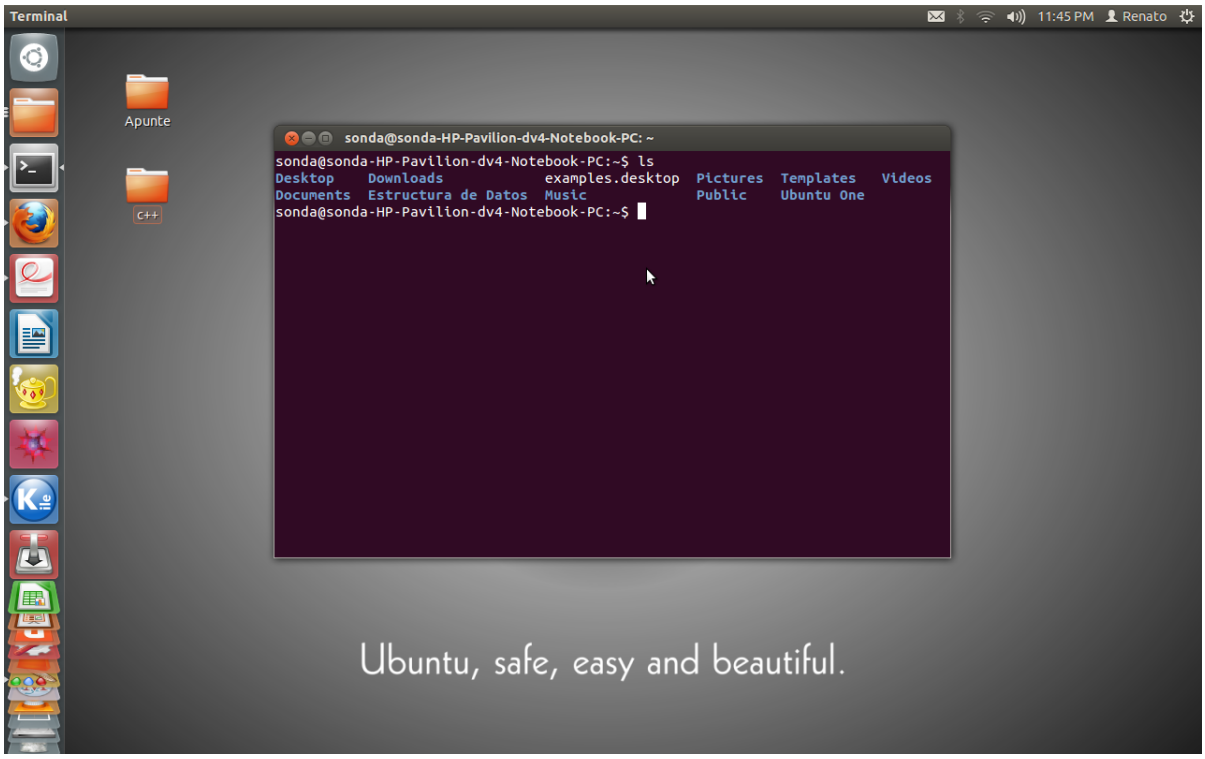


Hasta ahora tenemos dos carpetas en el escritorio nada fuera de lo normal. Abriremos la terminal.

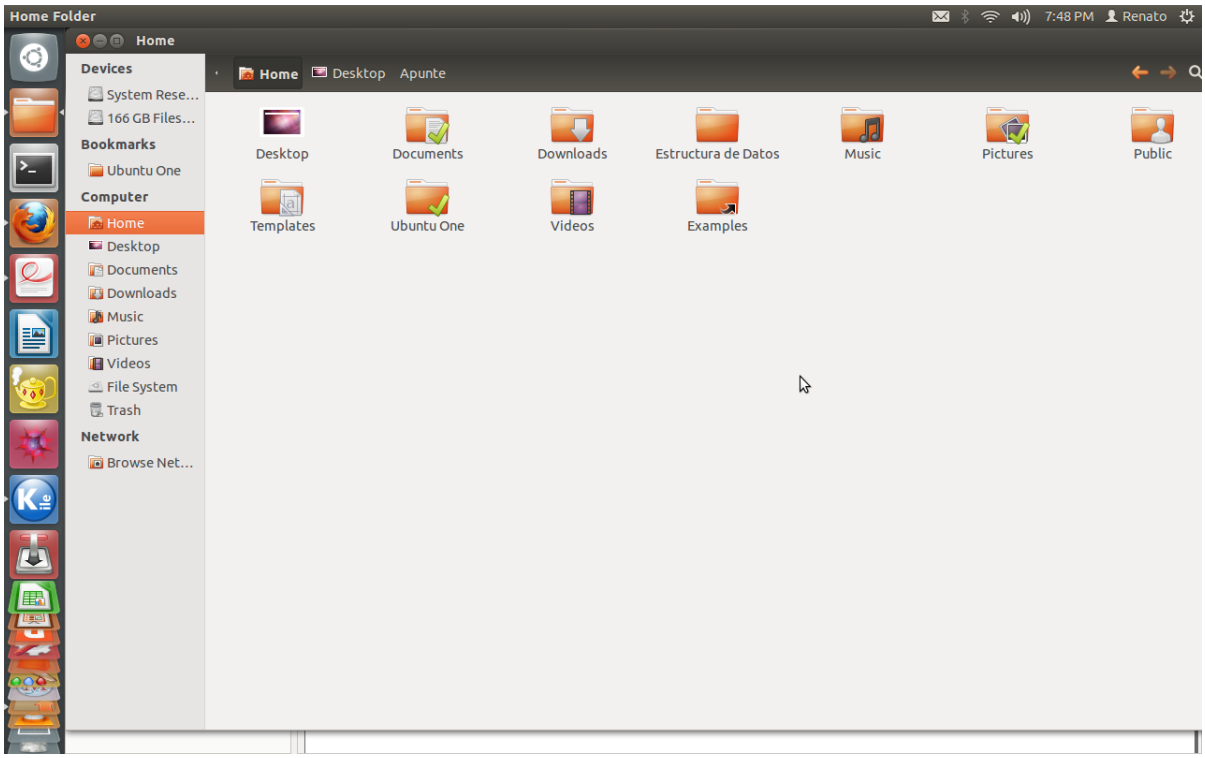
[3] http://es.wikipedia.org/wiki/Terminal_%28inform%C3%A1tica%29



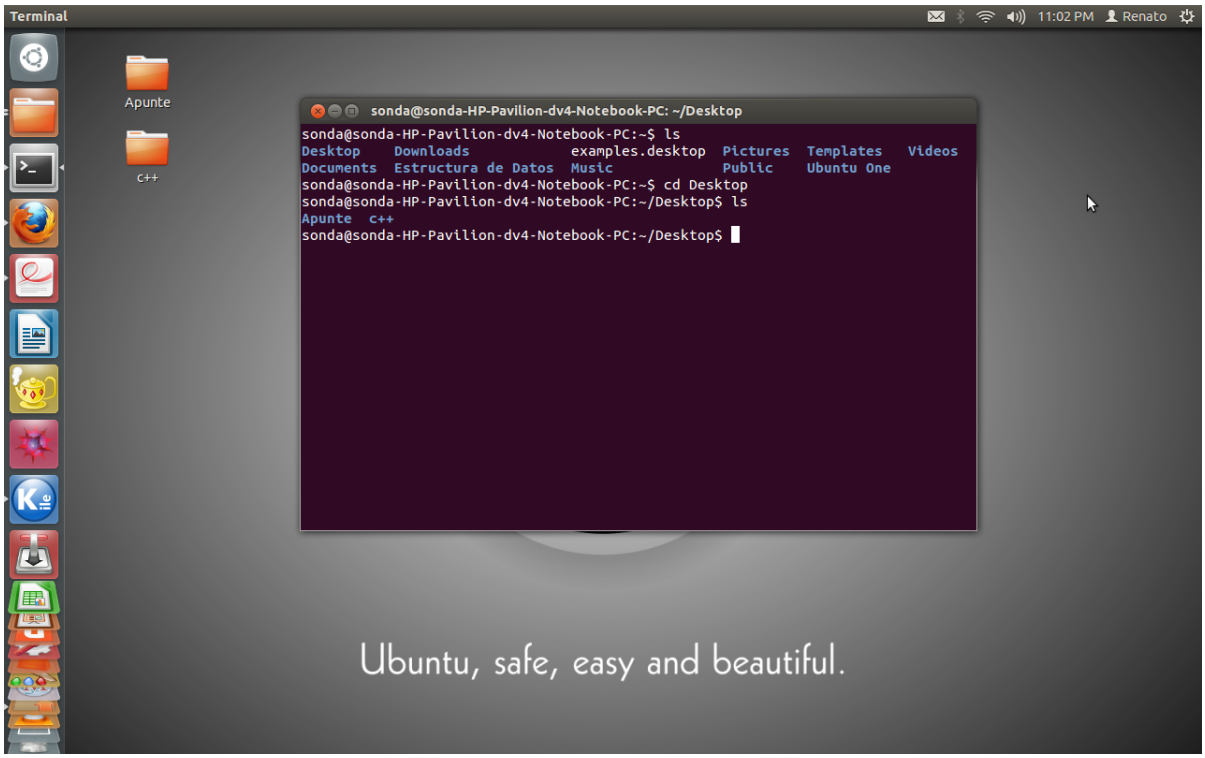
Ahora que tenemos nuestra terminal abierta probemos esos nuevos comandos que hemos mencionado. Partamos con ls, este comando nos muestra todas las cosas que se encuentran en la carpeta en la cual estamos(recordar que la terminal siempre comienza en home)



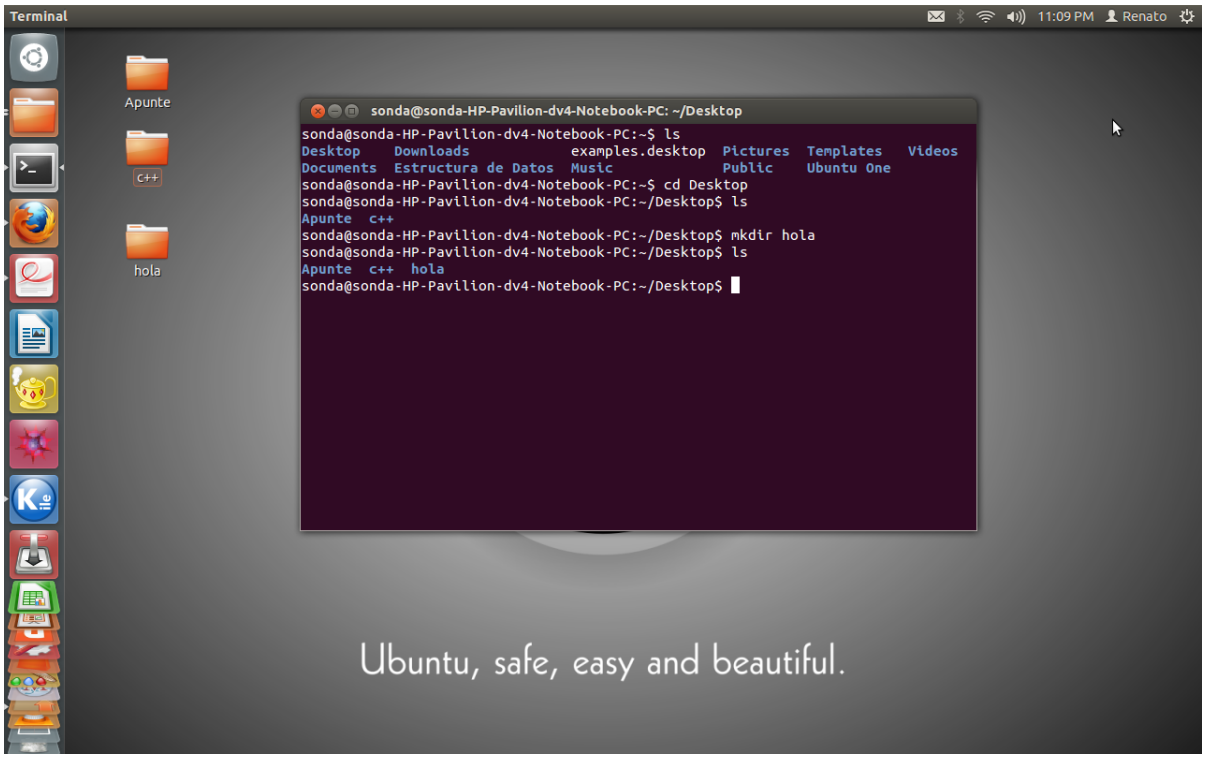
Si ahora buscamos la carpeta home sin usar la terminal nos encontraremos con el mismo resultado.



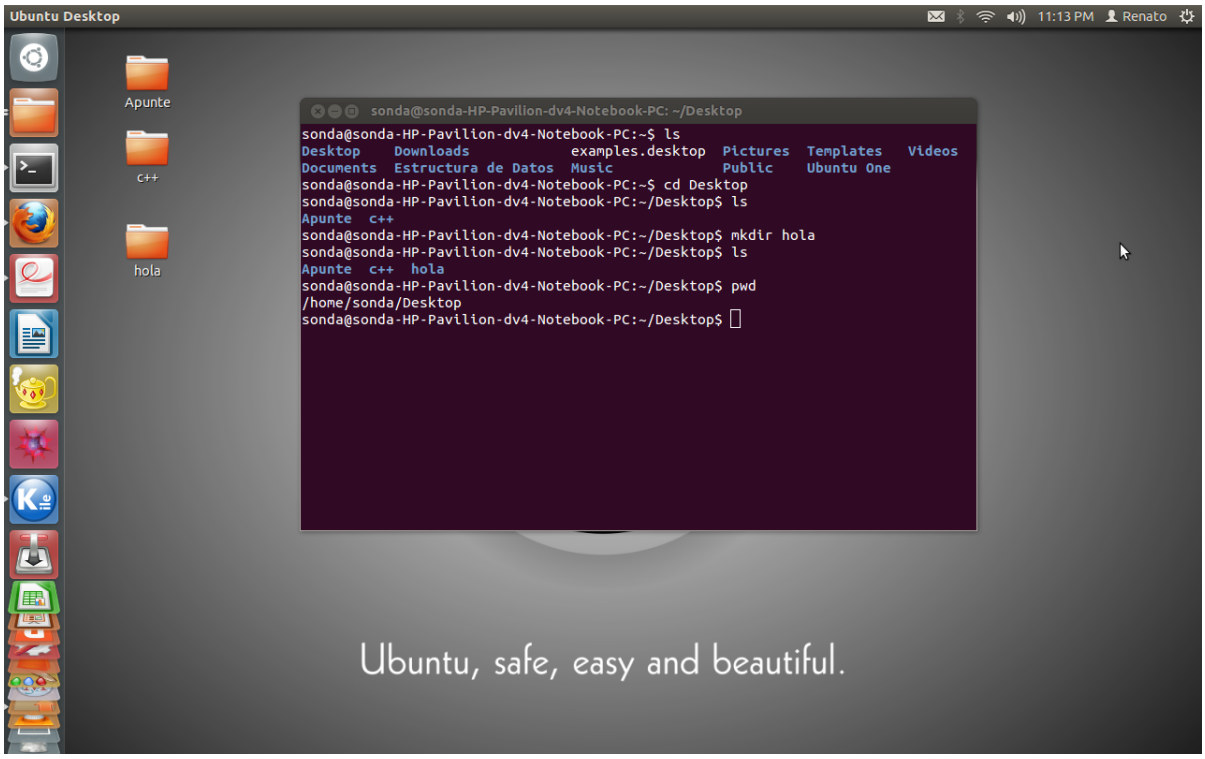
El comando cd permite situarnos en otra carpeta. Eligiremos Desktop y luego usaremos ls para revisar que contiene esta carpeta.



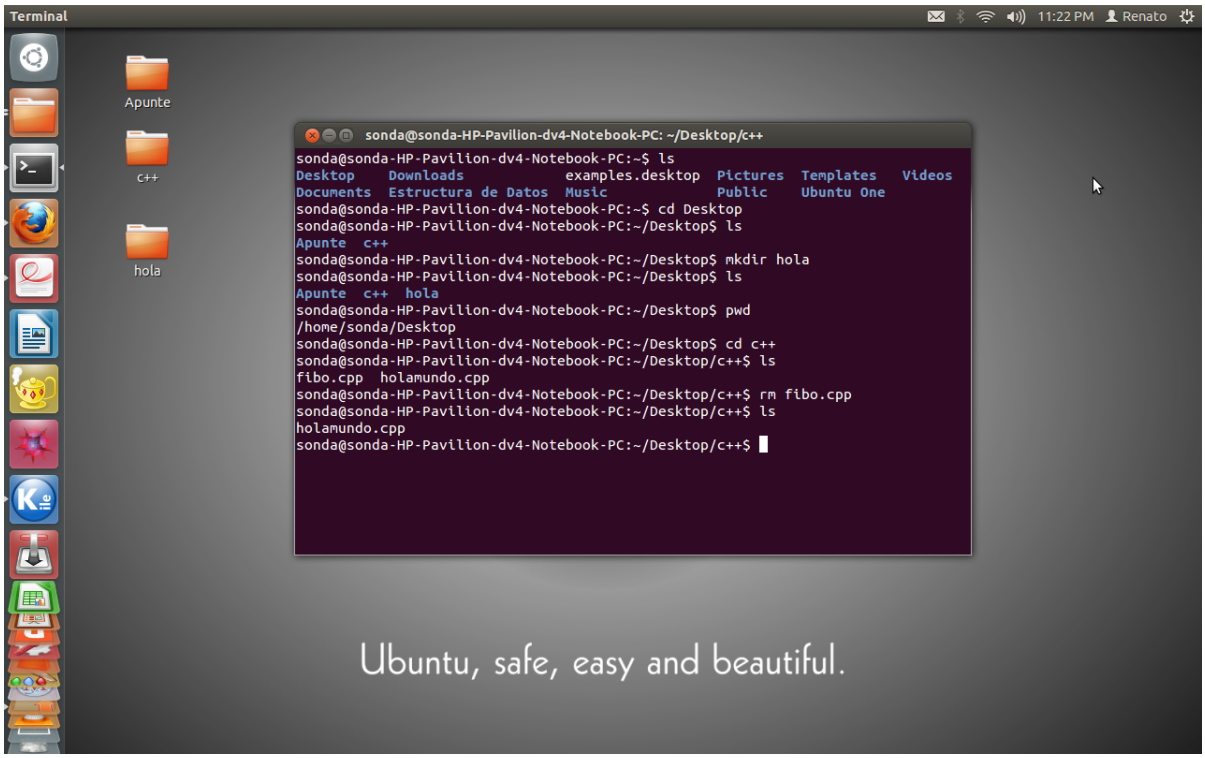
Nos damos cuenta que el contenido de la carpeta Desktop no es nada más ni nada menos lo que hay en el escritorio. Ahora usaremos mkdir el cual sirve para crear una carpeta donde estamos ubicados. Crearemos la carpeta “hola” y luego usaremos el comando ls para verificar que fue creada.



El comando pwd nos muestra el directorio actual.

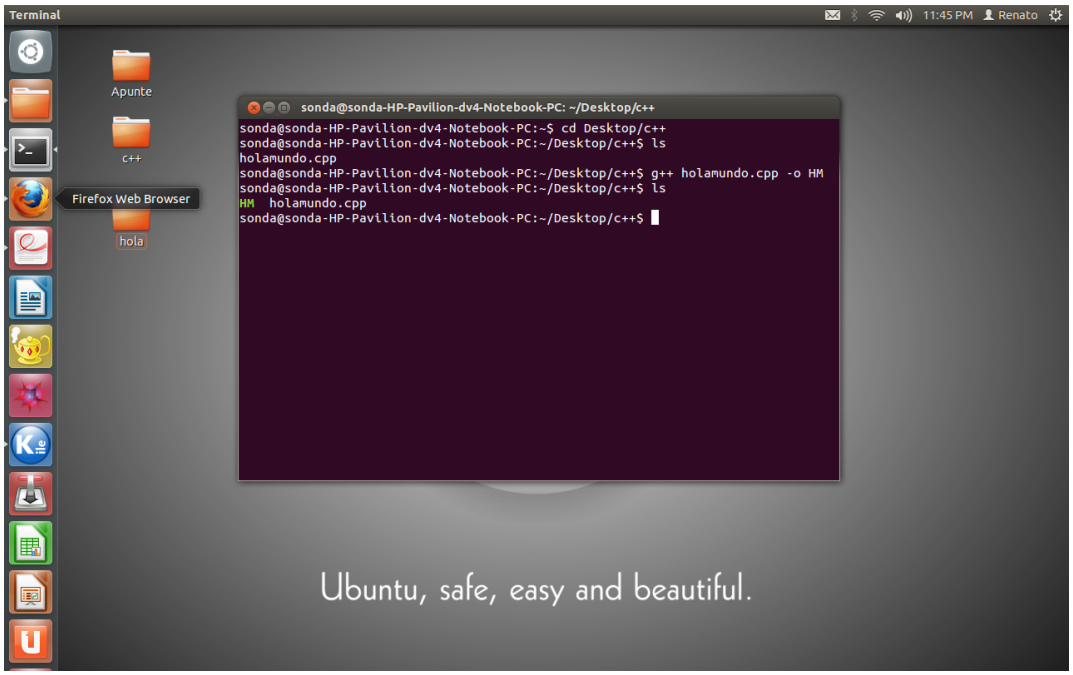


Y finalmente rm sirve para borrar un archivo en la carpeta actual. Ingresaremos en la carpeta c++, usaremos ls para ver que contiene, borraremos el archivo fibo.cpp utilizando rm y finalmente verificaremos que se haya borrado.

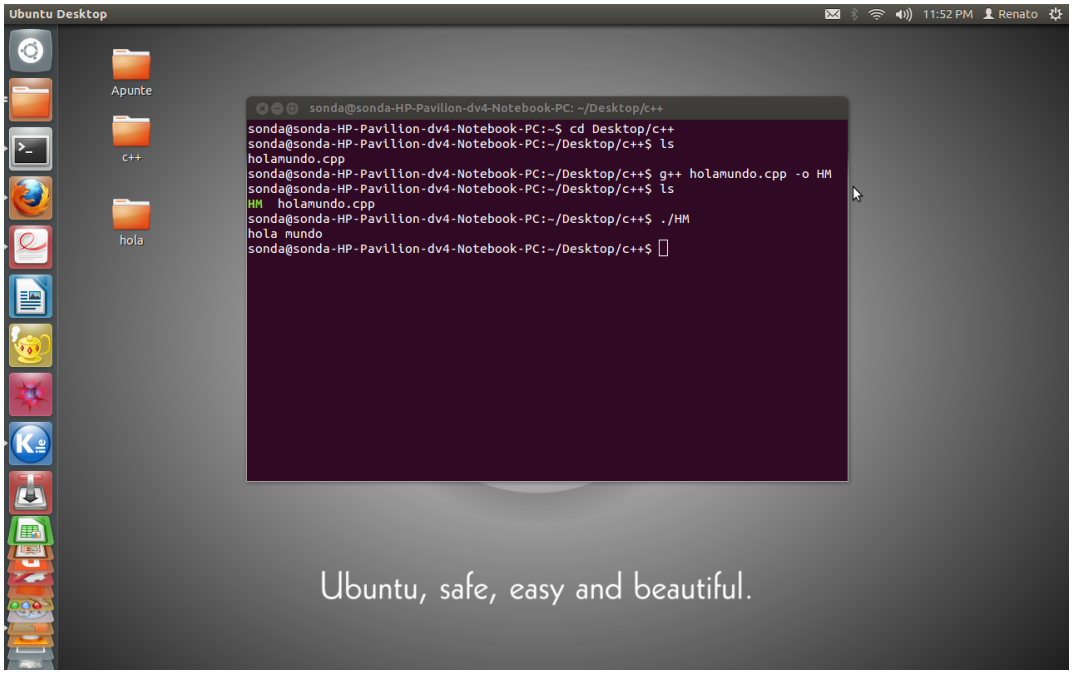


Y bien, ya conocemos algunos comandos y sus usos básicos, pero existen más. Incluso los acá mencionados pueden usarse de manera más eficiente, pero aprender esto será tarea del estudiante. La comunidad Ubuntu entrega mucho soporte a los nuevos usuarios por lo cual recomendamos que su primer acercamiento al mundo de Linux sea con Ubuntu 12.04.

En fin, ha llegado la hora de cumplir con lo prometido: La compilación de nuestro programa. En la carpeta c++ vimos un archivo llamado holamundo.cpp . Este archivo fue escrito en un editor de texto, ustedes pueden usar el que más les acomode. Una ves terminado para compilar nos ubicaremos en la carpeta donde esta nuestro programa y escribiremos la siguiente linea en la terminal: g++ holamundo.cpp -o HM
En esta frase g++ es el compilador que usamos(asegúrense de haberlo instalado), holamundo.cpp es el programa escrito en lenguaje de programación entendible por humanos (c++) y HM es el programa en un lenguaje que la máquina puede entender. En otras palabras HM es un ejecutable y aparece con letras verdes al usar ls.



Para ejecutarlo escribimos:./HM



Con esto concluye este pequeño tutorial para compilar por consola. Cabe mencionar que g++ (y en general, los compiladores) tiene un montón de opciones a la hora de compilar, por lo que si desean aprender más, puede buscarlo en los manuales de su compilador[4]. Ahora a programar!

[4] man gcc

3. Estructura básica de un programa

Para poder apreciar la estructura básica de un programa no existe nadie mejor que nuestro viejo amigo “hola mundo"

holamundo.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout<< ("hola mundo") << endl;
8      return 0;
9  }
10
```

- La primera línea sirve para incluir la biblioteca estándar “input output stream” con los elementos que premiten la entrada y salida de datos por consola.
- la línea 2 y 4 son líneas en blanco cuyo único fin es hacer más comprensible el codigo para quien lo lee (fin estético). Se pueden omitir si usted lo desea pero es recomendable siempre mantenur un código ordenado.
- La linea 5 es la declaracion de la función main, que todo programa debe tener. A partir de esta función se ejecuta el programa. Comienza con int ya que por tradición la función main debe retornar un entero cuando termina su ejecución. Dentro de los paréntesis no hay nada ya que no requiere datos de entrada.
- En la linea 6 vemos un paréntesis de llave cuya función es indicar el comienzo del cuerpo de la función main.
- En la linea 7 cout que es el encargado de la salida de datos por consola. Cabe notar que realmente la sentencia debe escribirse como std::cout, pero al escribir la línea 3 nos ahorramos escribir std:: cada vez que queramos enviar algo por pantalla. Los símbolos << indican la direccion del flujo de datos y endl realiza el salto de línea. Finalmente todas las sentencias en C++ terminan con un punto y coma.
- En la línea 8 se termina la ejecución del programa con return. En programas pequeños la importancia de esta línea puede pasar desapercibida, pero en proyectos más complejos que dependen de muchos programas es muy útil. Si la función main retorna cero durante su ejecución indica que el programa se ejecuto correctamente.
- Finalmente la línea 9 nos indica el fin de la función main.

4. Tipos de Variables

En Python, cuando ustedes necesitaban una variable simplemente elegian un nombre y le daban algún valor, teniendo la libertad incluso para cambiar ese valor por otro de otro tipo. Por ejemplo:

```
1  numero = 10
2  doble = numero*2
3  doble = "chocolate"
```

En C++ esto sería considerado una atrocidad. Primero que todo, en C++, todas las variables que usaremos deben ser declaradas [5]. Por convención esta declaración se hace al comienzo de cada función. En la declaración se declara el tipo de la variable y solo se le pueden asignar valores de ese tipo a la variable en cuestión.
Los tipos de variables básicos son:

- Los números enteros(int).
- Los números de coma flotante(float).
- Los números de coma flotante con doble presición(double).
- Los caracteres(char).
- Los Booleanos(bool).

Una de las mejoras que se le realizaron al lenguaje C durante la creación de C++ fue en el manejo de string(cadena de caracteres). Incluyendo la biblioteca “string” pueden usar el tipo de variable string como si fuera un tipo básico. Lo bueno es que el compilador g++ al incluir la biblioteca “iostream” incluye también “string” entonces pueden usar este tipo con total libertad incluyendo solamente la biblioteca “iostream”(En C era una tortura trabajar con string). A pesar de lo anterior, es recomendable que se incluya de todas formas la biblioteca “string”, dado que esta sea incluída junto con “iostream” es una particularidad de g++ y puede que no todos los compiladores lo hagan, así que por temas de portabilidad y compatibilidad, se recomiendo incluírla de todas formas.

A continuación presentaremos un programa en el cual se declararán variables y se les asignará un valor para que el lector pueda apreciar directamente cual es el uso que cada uno tiene.

tipos.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int entero;
7      float pi_aprox;
8      double pi;
9      char letra;
10     bool valor_de_verdad;
11     string palabra;
12
13     entero=3;
14     pi_aprox= 3.14159;
15     pi=3.14159265358979323846264338327950288419716939937510 58;
16     letra='a';
17     valor_de_verdad=false;
18     palabra="chocolates";
19
20     cout<<"Te apuesto "<<entero<<" "<<palabra<<" "<<letra<<" que "<<(pi-pi_aprox)<<"<0 es "<<valor_de_verdad<<endl;
21     return 0;
22 }
```

Si usted compila el programa anterior se dara cuenta q la variable $valor\_de\_verdad$ se mostrará por pantalla como un 0. En c++ por $false = 0$ y por defecto $true = 1$ [6]

[5] Entendamos por ahora que declarar es decirle al compilador que reserve la memoria para algo

[6] A pesar de que en C++ existe el tipo bool, muchos compiladores implementan dichos tipos de la forma que se hacía en C: 0 es falso y cualquier número diferente de 0 es verdadero.

5. Entrada y Salida de datos

Como ya vimos en el programa anterior la salida de datos por pantalla se hace con cout (std::cout) separando variables o string que queremos mostrar mediante el símbolo << (llamado operador de flujo). Para la entrada de datos usamos std::cin. A continuación mostraremos con un ejemplo como recibir y mostar datos por consola.

entradaysalida.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    int anno_actual,edad,nacimiento;

    cout<<"Ingrese su edad y el año actual"<<endl;
    cin >> edad >> anno_actual;
    nacimiento=anno_actual-edad;
    cout<< "Usted nacio el año " << nacimiento<<"."<< endl;
    return 0;
}
```

6. Sentencias condicionales

Ustedes ya estarán familiarizados con algunas del ramo de programación. En c++ tenemos las siguientes sentencias: if,else,switch. La más usada es if(junto con else) aunque switch puede ser más útil bajo ciertas cirscuntancias.

```
if(condiciones){

   /*cosas que pasan si se cumplen las condiciones*/

}
```

Donde las condiciones deben ser un valor booleano.

En cuanto a else, se usa de forma similar con la diferencia que no tiene condiciones ya que se usa después de un if y el cuerpo del else se ejecuta en caso que no se cumpla la condición del if. Acontinuación se ilustrará con un ejemplo.

parimpar.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int numero;

    cout<< "ingrese un numero"<< endl;
    cin>> numero;
    if(numero%2==0)
    {
        cout<<numero<<" es par"<<endl;
    }
    else
    {
        cout<<numero<<" es impar"<<endl;
    }
    return 0;
}
```

La sentencia switch es particularmente útil en los menús y su estructura es la siguiente:

```
switch (variable)
{
    case 'a':
        /*lo que se hace si el valor de "variable" es "a"*/
        break;
    case 'b':
        /*lo que se hace si el valor de "variable" es "b"*/
        break;
    default:
        /*lo que se hace si el valor de "variable" no es ninguno de los
         anteriores*/
         break;
}
```

Cabe notar switch no tiene un límite de casos, lo cual es una gran ventaja en comparación con if que trabaja con valores booleanos (verdadero y falso). Sin embargo se pueden usar varios if (incluso anidarlos) para emular un switch. La idea es usar las herramientas que tenemos a disposición para escribir un código lo más ordenado posible. El break termina la ejecución del switch por lo cual no es necesario agregarlo en el default ya que es el ultimo caso.
En el siguiente ejemplo se simulará un vendedor de frutas.

feria.cpp

```cpp
#include <iostream>

using namespace std;

int main(){
    float deuda,kilos;
    char fruta;
    string a;


    cout <<"¿Desea comprar alguna fruta?(si/no)"<<endl;
    cin >>a;

    if(a=="si"){
        cout <<"¿Que fruta desea comprar?"<<endl;
        cout <<"Ingrese M para manzana, N para naranja, P para palta"<<endl;
        cin >> fruta;
        switch (fruta){
            case 'M':
                cout <<"¿Cuantos kilos desea?"<< endl;
                cin >> kilos;
                deuda= kilos*500;
                cout <<"Son "<< deuda<<" pesos porfavor."<<endl;
                break;
            case 'N':
                cout <<"¿Cuantos kilos desea?"<< endl;
                cin >> kilos;
                deuda= kilos*600;
                cout <<"Son "<< deuda<<" pesos porfavor."<<endl;
                break;
            case 'P':
                cout <<"¿Cuantos kilos desea?"<< endl;
                cin >> kilos;
                deuda= kilos*1500;
                cout <<"Son "<< deuda<<" pesos porfavor."<<endl;
                break;
            default:
                cout<<"No tenemos ese tipo de fruta"<<endl;
                break;

        }
    }
    else{
        cout<<"vuelva pronto"<<endl;
    }
    return 0;
}
```

7. Iteraciones

En el programa anterior algunos pudieron darse cuenta que el vendedor solo vendía una cosa y se acaba su interacción con el comprador. Esto no es muy práctico en la vida real por lo que si queremos un vendedor más inteligente tendremos que intentar algo distinto. El programa se ejecuta de forma secuencial, cada vez que se ejecuta una línea se salta a la siguiente para nunca más volver(hasta ahora). Podríamos copiar el mismo código para que el vendedor ofrezca sus productos varias veces, pero esto no es práctico y no sirve para una cantidad arbitraria de veces. Para cumplir esta noble misión necesitamos los iteradores: while, do while y for.
Estructura de for.

```
for(inicialización;condición de término;incremento){

    /*Se ejecuta si la condición de término no se ha alcanzado y
     se repite mientras no se cumpla esta condición */

}
```

Estructura de while.

```
while(condiciones){

    /*Lo que se hace si se cumplen las condiciones. Además la
    ejecución se repite mientras se cumplan las condiciones*/

}
```

Estructura de do while.

```
do{

    /* El cuerpo del do while se ejecuta porlomenos la primera vez.
     Si se cumplen las condiciones alfinal de cada ejecucón se repite*/

}while(condiciones)
```

Al igual que en if, las condiciones deben ser un valor booleano.

A continuación analizaremos algunos ejemplos en donde se aplican cada uno de ellos.

- El primer programa que veremos en esta sección calcula la siguiente sumatoria:

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

  Donde n es un número natural ingresado por consola. Si nos sabemos esta fórmula entonces nisiquiera necesitamos escribir un programa para calcularlo, pero en el caso que se nos olvide tendríamos que calcularla en el modo “tradicional” que en otras palabras es sumar los n primeros cuadrados. Usaremos for para generar el bucle que sumara repetidamente los cuadrados sin tener que escribir varias veces el mismo código.

sumcuadrados.cpp

```cpp
#include <iostream>

using namespace std;

int main(){
    int n,i,suma;

    suma=0;
    cin>> n;
    for(i=1;i<=n;i++){
        suma=suma+i*i;
    }
    cout<< suma;
    return 0;
}
```

- El segundo programa nos indica si un número natural ingresado por consola es primo o no. La forma más común de determinar si un número es primo o no es dividir dicho número por todos los naturales menores a él. Si en alguna división el resto es cero entonces el número no es primo. Notar que como no sabemos a priori el número que ingresara el usuario no sabemos realmente cuantas veces tendremos que dividir para verificar si es primo o no a diferencia del ejemplo anterior que sabiamos de antemano que sumaríamos n veces.

primo.cpp

```cpp
#include <iostream>

using namespace std;

int main(){
    int numero,aux;
    bool primo;

    primo = true;
    cout << "ingrese un número natural"<<endl;
    cin >> numero;
    aux= numero-1;
    while(aux>1){
        if(numero%aux==0){
            primo=false;
        }
        aux--;
    }
    if(primo){
        cout<< numero <<" es primo"<<endl;
    }
    else{
        cout<< numero <<" no es primo"<<endl;
    }
    return 0;
}
```

- El último programa de esta sección puede que les parezca familiar. Es similar al programa de la sección anterior que simulaba un vendedor de feria, pero esta vez lo hicimos más inteligente con la ayuda de do-while.

feriainteligente.cpp

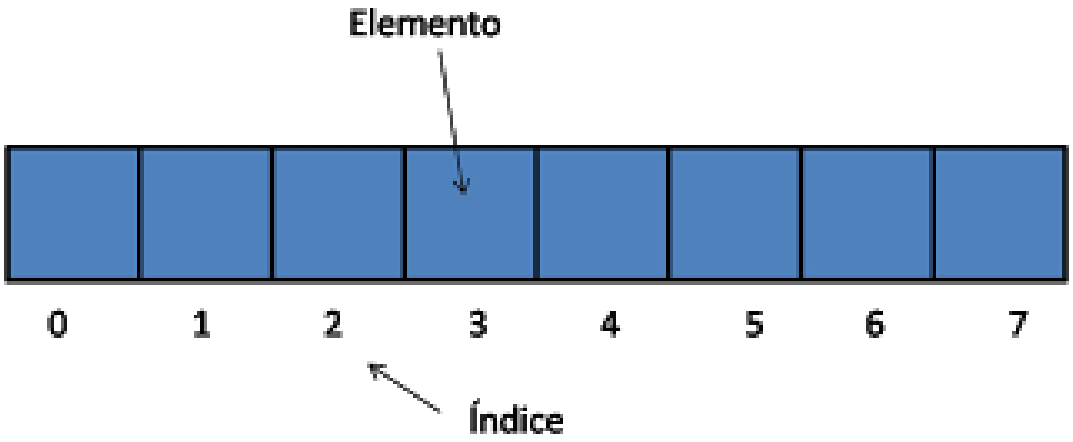
```cpp
#include <iostream>

using namespace std;

int main(){
    float deuda,kilos;
    char fruta;
    string a,respuesta;

    cout <<"¿Desea comprar alguna fruta?(si/no)"<<endl;
    cin >>a;
    do{
        if(a=="si"){
            cout <<"¿Que fruta desea comprar?"<<endl;
            cout <<"Ingrese M para manzana, N para naranja, P para palta"<<endl;
            cin >> fruta;
            switch (fruta){
                case 'M':
                    cout <<"¿Cuantos kilos desea?"<< endl;
                    cin >> kilos;
                    deuda= kilos*500;
                    cout <<"Son "<< deuda<<" pesos porfavor."<<endl;
                    break;
                case 'N':
                        cout <<"¿Cuantos kilos desea?"<< endl;
                        cin >> kilos;
                        deuda= kilos*600;
                        cout <<"Son "<< deuda<<" pesos porfavor."<<endl;
                        break;
                case 'P':
                        cout <<"¿Cuantos kilos desea?"<< endl;
                        cin >> kilos;
                        deuda= kilos*1500;
                        cout <<"Son "<< deuda<<" pesos porfavor."<<endl;
                        break;
                default:
                        cout<<"No tenemos ese tipo de fruta"<<endl;
                        break;
            }
            cout<<"¿Desea llevar algo más(si/no)?"<<endl;
            cin>>respuesta;
        }

        else{
            cout<<"vuelva pronto"<<endl;
        }
    }while(respuesta=="si");
    return 0;
}
```

Si comparamos feria.cpp y feriainteligente.cpp nos damos cuenta que este último solo tiene 2 líneas más de código(en programación es prácticamente nada ya que los programas que usamos a diario estan hechos con miles) las cuales convirtieron a un vendedor que probablemente se iría directo a bancarrota por un vendedor razonable. En programación es bastante útil ser ingenioso al usar las herramientas escasas que tenemos para lograr nuestras metas.

8. Arreglos

Con las herramientas que tenemos hasta el momento podemos crear una infinidad de programas, pero si nos damos cuenta en los ejemplos vistos en este apunte solo trabajamos con una cantidad reducida de variables. Los programas que usamos en nuestros computadores generalmente requieren de miles de variables. Como ya vimos para usar una variable hay que declararla previamente ¿Se imaginan lo confuso que se vería el codigo al declarar unas 5000 variables? Para solucionar esta problemática existen los arreglos. Los arreglos son estructuras de dato lineales que almacenan datos del mismo tipo y todos estos datos son guardados contiguamente en memoria principal (uno detrás de otro). Es decir, podemos crear arreglos de enteros, de caracteres, de booleanos, etc.

Para que el lector se haga una idea de lo que es un arreglo, estos pueden ser representados por un dibujo. Lo que usted obtiene al declarar un arreglo de 8 elementos es lo siguiente:



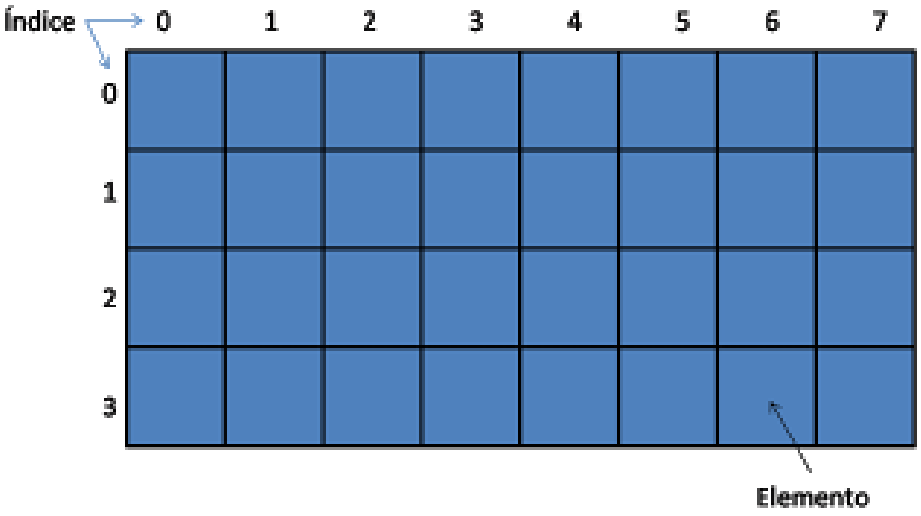
En cada "casilla" podemos guardar un elemento, si lo declaramos como un arreglo de enteros solo podremos guardar enteros en él. Además cada casilla tiene un índice. Los índices parten del 0 y terminan en el tamaño del arreglo menos uno. Estos índices nos permiten acceder a las casillas que en el fondo contiene una variable.

En el siguiente programa un profesor calcula los promedios de sus alumnos de una manera poco ética. Su curso es de 120 alumnos y asigna los promedios de forma aleatoria(entre 1 y 100), el índice en el arreglo es el número de cada alumno en la lista del curso(lista que parte en cero y termina en 119).

profesorflojo.cpp

```cpp
#include <iostream>
#include <stdlib.h> /*esta librería nos permite usar la función que
                      genera los números aleatorios*/
using namespace std;

int main(){
    int notas[120]; /*declaración de un arreglo*/
    int i;

    for(i=0;i<120;i++){/*se llena el arreglo con notas aleatorias*/
        notas[i]= 1+rand()%(101-1);
    }
    for(i=0;i<120;i++){/*se muestran las notas por pantalla*/
        cout<<notas[i]<<endl;
    }
    return 0;
}
```

El arreglo que vimos recién es un arreglo unidimensional también conocido como vector, sin embargo existen arreglos de dos, tres o incluso más dimensiones. Los que más usaremos nosotros serán los vectores y las matrices(arreglos de dos dimensiones). A continuación mostraremos un dibujo de un arreglo bidimensional:



Esta vez se podrá acceder a cada casilla atravez de dos números, uno que indica la fila y el segundo la columna. El siguiente programa usa un arreglo bidimensional para almacenar un horario académico.

matriz.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      string calendario[5][5];
7      int i,j;
8
9      calendario[0][0]="Lun";calendario[0][1]="Mar";
10     calendario[0][2]="Mié";calendario[0][3]="Jue";
11     calendario[0][4]="Vie";
12     for(j=0;j<5;j++){
13         for(i=1;i<5;i++){
14             cout<<"Ingrese las tres primeras letras del ramo"<<endl;
15             cout<<"día "<<calendario[0][j]<<" en el bloque "<<i<<endl;
16             cin>>calendario[i][j];
17         }
18     }
19     for(i=0;i<5;i++){
20         for(j=0;j<5;j++){
21             cout<<calendario[i][j]<<"    ";
22         }
23         cout<<endl;
24     }
25     return 0;
26 }
```

Acá se pueden apreciar los “for anidados”(usar un for dentro de otro for) la cual es una herramienta muy utilizada en programación y es indispensable al trabajar con matrices. La variable i por convención es usada para indicar la fila en la cual estamos mientras que la variable j la columna(puede relacionarlo con los famosos vectores unitarios paralelos a los ejes cartesianos). En el primer for anidado llenamos la matriz por columnas y en el segundo for anidado se muestran los elementos fila por fila. Esta diferencia la hace la forma que estan anidados los for(j primero por columnas/i primero por filas).

Asi concluimos con el uso básico de arreglos en programación. Estos arreglos tienen una desventaja: Una vez que los declaramos no los podemos agrandar(tienen tamaño fijo). En ocaciones necesitaremos una cantidad de variables que no conocemos a priori y si bien esto se podría solucionar creando un arreglo enorme, no es una buena opción ya que consume mucho espacio en la memoria. Una solución mas elegante a esta problemática será introducida más adelante cuando veamos como funciona la memoria RAM.

9. Funciones

Hasta el momento hemos visto que todos nuestros programas tienen la función main. Esta es la función que se ejecuta cuando ejecutamos el programa, pero no necesariamente todo el código debe estar dentro de esta función, es más, es recomendable, siempre que se pueda, modularizar con funciones para hacer el código más fácil de entender por humanos. Cuando tenemos que realizar una acción muchas veces como por ejemplo mostrar por pantalla una matriz es mejor crear una función que se llame mostrarMatriz y llamarla desde la función main cada vez que lo deseemos.

Toda las funciones tienen un tipo, un nombre, parámetros y un cuerpo. La función main que es la única que hemos visto hasta ahora es de tipo int(retorna cero), su nombre es(valga la rebundancia) main, no recibe parámetros ya que no necesita nada para ejecutarse [7] y el cuerpo es el programa en sí.

La estructura de una función es la siguiente:

```
1  tipo nombre(parámetros){
2      cuerpo
3  }
```

Para usar las funciones primero debemos declararlas y así podremos invocarlas desde la función main. Una función puede llamar funciones a su vez mientras se ejecuta. Además pueden recibir todo tipo de variables y arreglos como parámetros. Los nombres de variables que se declaran dentro de funciones diferentes pueden repetirse ya que cada vez que se entra a una funcion se crean variables nuevas que existirán mientras la función se ejecute. Cuando la función retorna algún valor se borran todas las variables declaradas incluyendo los parámetros.

Existen casos que los parámetros no son borrados al terminar la ejecución de una función. Esto ocurre cuando se pasan los parámetros por referencia(lo veremos más adelante).
En el siguiente ejemplo hacemos uso itensivo de funciones para llevar a cabo una tarea que en un comienzo parecía trivial, sin embargo terminaron siendo mas de 100 líneas de código(recuerde que los programas en este apunte no estan hechos para optimizar el uso de recursos sino para servir un fin académico).

El programa recibe las medidas de los lados y los angulos(grados) de un paralelogramo. Solo con estos datos determina si dicho paralelogramo es un cuadrado, un rectangulo, un rombo o un romboide y además calcula su perímetro y su área.

Para realizar esto se crearon funciones sobre cada procedimiento que llevaríamos a cabo en el papel si lo hicieramos a mano, por ejemplo. La función cuadrado retorna true si el paralelogramo es un cuadrado y false si no lo es. El perímetro se calcula para todos los paralelogramos de la misma forma, pero el área no. En la función área utilizamos las funciones cuadrado, rectángulo, rombo y romboide para determinar el camino a seguir para calcular el área.

La función main se usa para la entrada y salida de datos por consola. El “trabajo pesado” se hace al llamar nuestras funciones desde la función main.

[7] Puede recibir parámetros, que son los datos que son entregados en formas de opciones por la terminal

paralelogramo.cpp

```cpp
#include <iostream>
#include <math.h>

using namespace std;

double radianes(int grad){
    double rad;
    rad=(grad*3.14159265)/90;
    return rad;
}


double perimetro(double lados[]){
    int i;
    double perimetro;
    perimetro =0;
    for(i=0;i<4;i++){
        perimetro=perimetro+lados[i];
    }
    return perimetro;
}


bool cuadrado(double lados[],double angulos[]){
    if((lados[0]==lados[1])&&(lados[1]==lados[2])&&(lados[2]==lados[3])){
        if(angulos[0]==90){
            return true;
        }
    }
    return false;
}


bool rombo(double lados[], double angulos[]){
    if((lados[0]==lados[1])&&(lados[1]==lados[2])&&(lados[2]==lados[3])){
        if(angulos[0]!=90){
            return true;
        }
    }
    return false;
}

bool rectangulo(double lados[],double angulos[]){
    if((cuadrado(lados,angulos)||rombo(lados,angulos))){
        return false;
    }
    else{
        if(angulos[0]==90){
            return true;
        }
    }
    return false;
}

bool romboide(double lados[],double angulos[]){
    if((cuadrado(lados,angulos))||(rectangulo(lados,angulos))||(rombo(lados,angulos))){
        return false;
    }
    else{
        return true;
    }
}

double area(double lados[],double angulos[]){
    int i;

    if(cuadrado(lados,angulos)){
        cout<<"Es un cuadrado"<<endl;
        return (lados[0]*lados[0]);
    }
    if(rectangulo(lados,angulos)){
        cout<<"Es un rectangulo"<<endl;
        for(i=1;i<4;i++){
            if(lados[0]!=lados[i]){
                return (lados[0]*lados[i]);
            }
        }
    }
    if(rombo(lados,angulos)){
        cout<<"Es un rombo"<<endl;
        return (lados[0]*lados[0]*sin(radianes(angulos[0])));
    }
    if(romboide(lados,angulos)){
        cout<<"Es un romboide"<<endl;
        for(i=0;i<4;i++){
            if(lados[0]!=lados[i]){
                return (lados[0]*lados[i]*sin(radianes(angulos[0])));
            }
        }
    }
    return -9000;
}

int main(){
    double lados[4],angulos[4];
    cout<<"Ingrese los lados de un paralelogramo"<<endl;
    cin>>lados[0]>>lados[1]>>lados[2]>>lados[3];
    cout<<"Ingrese los ángulos del paralelogramo"<<endl;
    cin>>angulos[0]>>angulos[1]>>angulos[2]>>angulos[3];
    cout<<"perimetro:"<<perimetro(lados)<<"  area:"<<area(lados,angulos)<<endl;
    return 0;
}
```

10. Punteros

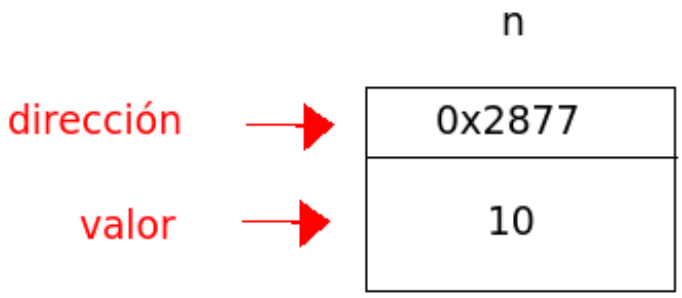
Muchos estudiantes de informática se asustan al escuchar la palabra “puntero”. Quizás se deba a los errores frecuentes que se producen por un uso incorrecto de punteros, los famosos “Segmentation Fault”.

Primero que todo los punteros no son nada sobrenatural. Un puntero es simplemente un tipo de variable. ¿Como un entero? ¡Sí! Como un entero(int), un carácter(char), un booleano(bool), etc. Pero ¿Qué representa un puntero?

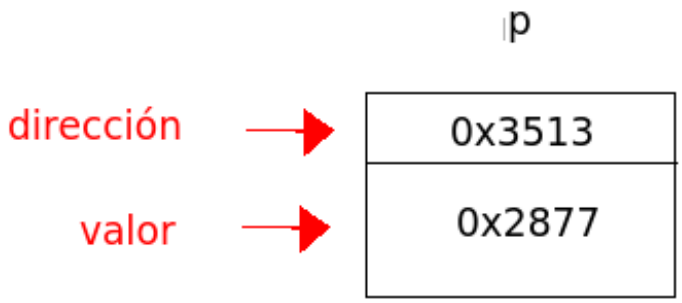
Las variables en general poseen una dirección en la memoria además de su valor. Por ejemplo si declaramos la siguiente variable:

int n = 10;

Al ejecutar el programa debería guardarse la variable “n” en la memoria RAM.



Un puntero es un tipo de variable que guarda una dirección de memoria. Por ejemplo, un puntero “p” que “apunta” a la variable “n” podemos representarlo gráficamente de la siguiente manera:



Como el puntero “p” tiene el valor de la dirección de memoria de “n” decimos que “p” es un puntero a un entero y lo declaramos de la siguiente forma:

int * p;

o también

int *p;

En general para declarar un puntero se sigue el siguiente formato:

(Tipo de variable a la cual apunta) *(nombre del puntero);

Hasta el momento solo hemos mostrado como declarar un puntero pero aún no podemos hacer nada con esto. A continuación veremos dos operadores que nos permiten trabajar con punteros. Son operadores unarios, es decir, solo trabajan con un operando, a diferencia de los operadores binarios(como el operador suma "+") que requieren dos operandos.

- Operador de dirección (&): Este operador nos entrega la dirección de memoria del operando(el operando es una variable que podría ser o no ser un puntero). ¡Con este operador podemos asignar valores a nuestros punteros!. A continuación haremos que el puntero "p" apunte al entero "n".

  p= &n;

- Operador de indirección (*): El operando en este caso es un puntero. El operador nos entrega el valor al cual apunta el operando. Si bien el mismo asterisco se usa para declarar un puntero y para utilizar el operador de indirección, la función que cumple depende del contexto. A continuación mostramos como obtener el valor al cual apunta el puntero "p" y lo mostramos por pantalla. En nuestro ejemplo el valor sería 10.

  cout<< "El valor apuntado es "<< *p << endl;

  Ahora que sabemos como declarar punteros y como trabajar con ellos mediante los operadores de dirección e indirección discutiremos en las siguientes secciones los usos más frecuentes de los punteros.

11. Paso por referencia

El paso por referencia es una de las utilidades que hace a los punteros una herramienta muy potente en la programación. Hasta el momento cuando usamos funciones, todos los parámetros que ingresamos a la función son copiados, se usan durante la ejecución de la función y luego se eliminan cuando la función retorna un valor o cuando se ejecuta la ultima línea de código.

¿Y si queremos que nuestra función modifique directamente variables de nuestro programa que están en el "main"?

En este caso en vez de pasar las variables como parámetros pasamos punteros a tales variables como parámetros. En el siguiente ejemplo las variables nota1 y nota2 son inicializadas en 100, para luego ser pasadas por referencia a una función que las modifica asignándoles un valor aleatorio.

puntero2014.cpp

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <time.h>
4
5  using namespace std;
6
7  void random(int *p){
8      sleep(1); srand (time(NULL));
9      *p=1+rand()%(101-1); /*se cambia el valor en la función main*/
10 }
11 int main(){
12     int nota1,nota2,*p1,*p2; /*declaración de variables y punteros*/
13
14     nota1=100; /*Asignación de valores*/
15     nota2=100;
16     p1=&nota1;
17     p2=&nota2;
18
19     random(p1); /*paso por referencia*/
20     random(p2);
21
22     cout<<"La nota 1 ahora es "<<nota1<<endl;
23     cout<<"La nota 2 ahora es "<<*p2<<endl;
24     return 0;
25 }
```

- La línea 8 permite un buen funcionamiento de la funcion rand().
- En las líneas 22 y 23 se muestran dos formas de obtener el valor de una variable. En la línea 22 se obtiene directamente y en la línea 23 se obtiene através de un puntero usando el operador de indirección.

12. Arreglos dinámicos

Para introducir los arreglos dinámicos hablaremos un poco sobre la memoria RAM(random access memory). Sea cual sea el sistema operativo que usemos, si el computador esta ejecutando un programa en un instante determinado(es lo que llamamos proceso), este programa se encuentra guardado en la memoria RAM.

Cuando ejecutamos nuestros programas estos ocupan un espacio en la memoria RAM. Este espacio esta dividido en tres partes:

- Memoria estática: Aquí se guardan todas las variables y arreglos que hemos visto hasta ahora. Este espacio de memoria no cambia durante la ejecución del programa.
- Stack: Es la memoria reservada para el llamado de funciones durante la ejecución de nuestro programa. Cada vés que se llama una función desde el "main" se crea un nuevo "frame" de memoria que ocupa espacio del stack.
- Heap: Es la memoria dinámica. Se puede solicitar memoria dinámica durante el tiempo de ejecución.

No es difícil imaginar alguna situación en la que necesitemos espacio adicional que no teníamos contemplado al escribir el programa. Por ejemplo, si quieremos hacer un programa para un profesor que guarde 100 notas de un curso en un arreglo podemos hacerlo, pero si ese profesor quiere prestárselo a otro que tiene cursos con 1000 notas entonces habría que modificar el código fuente de nuestro programa y volver a compilar.

Una forma de solucionar el problema anterior es escribir nuestro código fuente con arreglos de 1000000 notas ya que sabemos que ningún profesor tendrá más de un millón de alumnos. El modificar el código fuente equivale en este caso, a modificar la cantidad de memoria estática que nuestro programa ocupará al ser ejecutado. El problema es que cada vés que se ejecute nuestro programa se reservará en la memoria RAM espacio para estos arreglos enormes aunque el profesor solo tenga 50 alumnos lo que tiene como concecuencia un programa muy ineficiente en cuanto a espacio.

Otra forma de solucionar el problema es usar memoria dinámica. En nuestro código fuente podemos especificar que nuestro programa durante su ejecución pedirá memoria adicional del "heap".

- Ventajas: Eficiencia y elegancia.
- Desventajas: El programador adquiere una nueva responsabilidad(no menor) de manipular correctamente esta memoria. Errores de uso de la memoria dinámica son difíciles de depurar(Segmentation Fault).

Un arreglo dinámico es entonces un arreglo cuyo tamaño puede variar durante el tiempo de ejecución. Para usar arreglos dinámicos se siguen los siguientes pasos:

- Declarar el arreglo dinámico: Como siempre lo hemos hecho con las variables y arreglos.
- Asignarle memoria dinámica: Podemos asignarle la cantidad de memoria que necesitamos deacuerdo a nuestras necesidades.
- Liberar la memoria dinámica: Antes de finalizar nuestro programa debemos liberar toda la memoria dinámica que pedimos durante su ejecución.

A continuación mostramos la versión mejorada del programa del profesor flojo. Ahora se lo puede prestar a los demás profesores flojos sin importar cuantos alumnos tengan.

dinamic-array.cpp

```cpp
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(){

    int i;
    int size;
    int * array = NULL; /*declarar el arreglo que inicialmente no tiene
    memoria asignada */
    cout<<"¿Cuantas notas desea guardar?"<<endl;
    cin>>size;
    cout<<"Creando arreglo dinámico"<<endl;
    array = new int[size]; /*se le asigna la memoria que se necesita*/

    for(i=0;i<size;i++){
        array[i]= 1+rand()%(101-1);
    }
    for(i=0;i<size;i++){
        cout<<array[i]<<endl;
    }

    delete[] array;/*Se libera la memoria de arreglo*/
    return 0;
}
```

- Siempre que pedimos memoria dinámica usando *new* debemos liberarla con *delete*. Para liberar arreglos dinámicos se usa *delete[]*.

13. Estamos trabajando para usted.