Tabla de Contenidos

if, else

Las decisiones en un programa sirven para tomar distintos caminos dependiendo de una condición.

if, else

Las decisiones en un programa sirven para tomar distintos caminos dependiendo de una condición.

```
if(speed == 88)
  std::cout << "Great Scott!";
```

La sentencia después de `if` sólo será ejecutada si la condición es verdadera.

if, else

Las decisiones en un programa sirven para tomar distintos caminos dependiendo de una condición.

```
if(speed == 88)
  std::cout << "Great Scott!";
```

La sentencia después de `if` sólo será ejecutada si la condición es verdadera.Para ejecutar más de una sentencia es necesario abrir un bloque { }:

```
if(power >= 1.21)
{
  std::cout << "1.21 gigawatts!";
  std::cout << "1.21 gigawatts. Great Scott!\n";
}
```

if, else

Las decisiones en un programa sirven para tomar distintos caminos dependiendo de una condición.

```
if(speed == 88)
  std::cout << "Great Scott!";
```

La sentencia después de `if` sólo será ejecutada si la condición es verdadera.Para ejecutar más de una sentencia es necesario abrir un bloque { }:

```
if(power >= 1.21)
{
  std::cout << "1.21 gigawatts!";
  std::cout << "1.21 gigawatts. Great Scott!\n";
}
```

También es posible especificar el caso contrario:

```
bool haveEnoughPower;
if(power >= 1.21)
  haveEnoughPower = true;
else
  haveEnoughPower = false;
```

if, else

Tambien es posible encadenar una serie de condicionales:

```
int width, height;
std::cin >> width >> height;

if(width > height)
  std::cout << "Fat rectangle\n";
else if(height > width)
  std::cout << "Tall rectangle\n";
else if(width == height)
  std::cout << "Square\n";
else
  std::cout << "IMPOSSIBLE!!";
```

Los cuales son evaluados uno a uno hasta que la sentecia `else` sea ejecutada.

if, else

Tambien es posible encadenar una serie de condicionales:

```
int width, height;
std::cin >> width >> height;

if(width > height)
  std::cout << "Fat rectangle\n";
else if(height > width)
  std::cout << "Tall rectangle\n";
else if(width == height)
  std::cout << "Square\n";
else
  std::cout << "IMPOSSIBLE!!";
```

Los cuales son evaluados uno a uno hasta que la sentecia `else` sea ejecutada.

Do

Es recomendable mantener las condiciones simples, de manera de evitar errores de programación

Loops

El propósito de un ciclo (loop) es ejecutar un conjnto de sentencias hasta que se cumpla una condición.

Loops

El propósito de un ciclo (loop) es ejecutar un conjnto de sentencias hasta que se cumpla una condición.

while loop

```
while(expression) statement
```

Ejemplo:

```
int t = 10;
while(t != 0)
{
  std::cout << t << ", ";
  --t;
}
std::cout << "Blastoff!";
```

Loops

Go on, go on, go on, go on, go on...

do-while loop

```
do statement while(expression);
```

```
#include <iostream>

int main()
{
  bool wantACupOfTea;
  do
  {
    std::cout << "Cup of tea father? ";
    std::cin >> wantACupOfTea; // enter 0 or 1
    if(!wantACupOfTea)
      std::cout << "Go on\n";
  } while(!wantACupOfTea);

  return 0;
}
```

Loops

for loop

```
for(inicio; condicion; incremento) sentencia;
```

Loops

for loop

```
for(inicio; condicion; incremento) sentencia;
```

1. `inicio` Usado para inicializar el contador

Loops

for loop

```
for(inicio; condicion; incremento) sentencia;
```

1. `inicio` Usado para inicializar el contador
2. `condicion` se usa para chequear la condición de parada.

Loops

for loop

```
for(inicio; condicion; incremento) sentencia;
```

1. `inicio` Usado para inicializar el contador
2. `condicion` se usa para chequear la condición de parada.
3. `sentencia` se ejecuta.

Loops

for loop

```
for(inicio; condicion; incremento) sentencia;
```

1. `inicio` Usado para inicializar el contador
2. `condicion` se usa para chequear la condición de parada.
3. `sentencia` se ejecuta.
4. `incremento` se ejecuta y vuelve al paso 2.

Loops

for loop

```
for(inicio; condicion; incremento) sentencia;
```

1. `inicio` Usado para inicializar el contador
2. `condicion` se usa para chequear la condición de parada.
3. `sentencia` se ejecuta.
4. `incremento` se ejecuta y vuelve al paso 2.

Here's our blastoff example with a for loop:

```
for(int t = 10; t != 0; --t)
{
  std::cout << t << ", ";
}
std::cout << "Blastoff!";
```

Funciones

Una función es un mecanismo de agrupar sentencias de manera que pueda ser invocada desde distintas secciones de un programa. Por ejemplo, el siguiente código:

```
forceEarthSun = G * massEarth * massSun /
  (rEarthSun * rEarthSun);
forceEarthMoon = G * massEarth * massMoon /
  (rEarthMoon * rEarthMoon);
forceEarthMars = G * massEarth * massMars /
  (rEarthMars * rEarthMars);
```

Funciones

Una función es un mecanismo de agrupar sentencias de manera que pueda ser invocada desde distintas secciones de un programa. Por ejemplo, el siguiente código:

```
forceEarthSun = G * massEarth * massSun /
  (rEarthSun * rEarthSun);
forceEarthMoon = G * massEarth * massMoon /
  (rEarthMoon * rEarthMoon);
forceEarthMars = G * massEarth * massMars /
  (rEarthMars * rEarthMars);
```

sería posible reescribirlo como:

```
forceEarthSun = force(massEarth,  massSun, rEarthSun);
forceEarthMoon = force(massEarth, massMoon, rEarthMoon);
forceEarthMars = force(massEarth, massMars, rEarthMars);
```

Funciones

El formato de una función:

```
tipo nombre(parametro1, parametro2, ...) { sentencias }
```

donde:

Funciones

El formato de una función:

```
tipo nombre(parametro1, parametro2, ...) { sentencias }
```

donde:

- `tipo` es el tipo de dato que retorna la función (también es posible usar el tipo `void`).

Funciones

El formato de una función:

```
tipo nombre(parametro1, parametro2, ...) { sentencias }
```

donde:

- `tipo` es el tipo de dato que retorna la función (también es posible usar el tipo `void`).
- `nombre` el identificador de la función.

Funciones

El formato de una función:

```
tipo nombre(parametro1, parametro2, ...) { sentencias }
```

donde:

- `tipo` es el tipo de dato que retorna la función (también es posible usar el tipo `void`).
- `nombre` el identificador de la función.
- `parametro` son los argumentos de la función.

Funciones

El formato de una función:

```
tipo nombre(parametro1, parametro2, ...) { sentencias }
```

donde:

- `tipo` es el tipo de dato que retorna la función (también es posible usar el tipo `void`).
- `nombre` el identificador de la función.
- `parametro` son los argumentos de la función.
- `sentencias` cuerpo de la función. Estas senetencias se ejecutan cuando se llame la función.

Funciones

El formato de una función:

```
tipo nombre(parametro1, parametro2, ...) { sentencias }
```

donde:

- `tipo` es el tipo de dato que retorna la función (también es posible usar el tipo `void`).
- `nombre` el identificador de la función.
- `parametro` son los argumentos de la función.
- `sentencias` cuerpo de la función. Estas senetencias se ejecutan cuando se llame la función.

Por ejemplo:

```
double force(const double mass1, const double mass2,
  const double r)
{
  return G * mass1 * mass2 / (r * r);
}
```

Funciones

```
#include <iostream>

double force(const double mass1, const double mass2, const double r)
{
  const double G = 3.96402e-14;
  return G * mass1 * mass2 / (r * r);
}

int main() {
  // Astronomical units
  const double massSun = 1.0, massEarth = 3.003e-6, massMars = 0.323e-6;

  const double rSunEarth = 1.0, rSunMars = 1.523;

  const double forceSunEarth = force(massSun, massEarth, rSunEarth);
  const double forceSunMars = force(massSun, massMars, rSunMars);

  std::cout << "Forces:\n";
  std::cout << "Sun-Earth:  " << forceSunEarth << "\n";
  std::cout << "Sun-Mars: " << forceSunMars << "\n";
}
```

Nomenclatura

Definition

llamado corresponde a la salida de una función cuando finaliza la ejecución.

Functions sin tipo

Qué pasa si no necesitamos retornar un valor?

Functions sin tipo

Qué pasa si no necesitamos retornar un valor?

Usar tipo `void`:

```
void printForce(const std::string object1,
  const std::string object2,
  const double force)
{
  std::cout << object1 << " " << object2 << ":" << force <<
      "\n";
}
```

En C++ la palabra clave `void` denota la ausencia de un tipo de dato.

Declarando funciones

Todos los identificadores deben ser declarados antes de su uso. Por ejemplo el siguiente código falla al compilar:

```
const bool ON = true, OFF = false
const bool OPEN = true, CLOSED = false;

void setMicrowaveState(const bool newState) {
  if(newState == ON)
  {
    setMicrowaveDoor(CLOSED); // <- ERROR: Don't know about
    state = ON;               // setMicrowaveDoor
  }
  else
        state = OFF;
}
void setMicrowaveDoor(const bool newState) {
  if(newState == OPEN)
    setMicrowaveState(OFF);

  doorState = newState;
}
```

Declarando funciones

Es necesario declarar la función `setMicrowaveDoor` antes que `setMicrowaveState`, de la siguiente forma:

```
void setMicrowaveDoor(const bool newState);

void setMicrowaveState(const bool newState) {
  if(newState == ON)
  {
    setMicrowaveDoor(CLOSED); // Happy: you've told me about
    state = ON;               // setMicrowaveDoor
  }
  ...
}
void setMicrowaveDoor(const bool newState) { /*as before*/ }
```

La primera linea le dice al compilador que se espera ver una función con el nombre `setMicrowaveDoor`.

Declarando funciones

Definition

Prototipos de Funciones El encabezado de una función sin el cuerpo. Se usa para declara funciones:

```
void setMicrowaveDoor(const bool newState);
```

Valores por Defecto

Definition

valor por defecto Un valor que se usa como parámetro en el caso que no se entregue ninguno

ejemplo:

```
const int MONDAY = 0;
const int TEA = 0, COFFEE = 1;
void dispenseDrink(
  const int drinkType = COFFEE)
{
  std::cout << "Dispensing: ";
  if(drinkType == COFFEE)
    std::cout << "coffee...\n";
  else
    std::cout << "tea...\n";
}
```

```
int main()
{
  unsigned int dayOfWeek;
  // Enter number from 0 to 6
  std::cin >> dayOfWeek;

  if(dayOfWeek == MONDAY)
    dispenseDrink();
  else
    dispenseDrink(TEA);
}
```

Valores por Defecto

Deben ser ingresados al final de la lista de parámetros, por ejemplo.:

```
void dispenseDrink(int size,
  int drinkType = COFFEE,
  bool withMilk = false) { ... } // Good, can call:

dispenseDrink(1);          // or ...
dispenseDrink(3, TEA); // or ...
dispenseDrink(2, COFFEE, true)
```

Valores por Defecto

Deben ser ingresados al final de la lista de parámetros, por ejemplo.:

```
void dispenseDrink(int size,
  int drinkType = COFFEE,
  bool withMilk = false) { ... } // Good, can call:

dispenseDrink(1);          // or ...
dispenseDrink(3, TEA); // or ...
dispenseDrink(2, COFFEE, true)
```

En cambio:

```
void dispenseDrink(int drinkType = COFFEE,
  int size,
  bool withMilk = false) { ... }

dispenseDrink(/*what goes here?*/, 2); // Error!
```

Valores por Defecto

Do

- Se usan para automatizar los valores comúnmente usados

Valores por Defecto

Do

- Se usan para automatizar los valores comúnmente usados
- Para proveer un indicador de cuáles serían los valores razonables.

Sobrecarga de Funciones

Qué sucede si queremos crear una función que devuelva el producto punto para enteros y para dobles?

```
int dotInt(int x0, int y0, int x1, int y1)
{ return x0 * x1 + y0 * y1; }
double dotDouble(double x0, double y0, double x1, double y1)
{ return x0 * x1 + y0 * y1; }

dotInt(fromX, fromY, toX, toY);
```

Las dos funciones tienen el mismo cuerpo pero los tipos de datos de los argumentos son distintos.

Sobrecarga de Funciones

Qué sucede si queremos crear una función que devuelva el producto punto para enteros y para dobles?

```
int dotInt(int x0, int y0, int x1, int y1)
{ return x0 * x1 + y0 * y1; }
double dotDouble(double x0, double y0, double x1, double y1)
{ return x0 * x1 + y0 * y1; }

dotInt(fromX, fromY, toX, toY);
```

Las dos funciones tienen el mismo cuerpo pero los tipos de datos de los argumentos son distintos.

```
int dot(int x0, int y0, int x1, int y1)
{ return x0 * x1 + y0 * y1; }
double dot(double x0, double y0, double x1, double y1)
{ return x0 * x1 + y0 * y1; }

dot(fromX, fromY, toX, toY); // Compiler will choose
                             // correct version based on
                             // from/to number types
```

Sobrecarga de Funciones

Definition

funciones sobrecargadas dos o más funciones con diferentes tipos o cantidad de parámetros.

Sobrecarga de Funciones

Definition

funciones sobrecargadas dos o más funciones con diferentes tipos o cantidad de parámetros.

Ejemplos:

```
// Sum two or three integers
int sum(int n1, int n2);
int sum(int n1, int n2, int n3);

// Add together any integer/double
int add(int n1, int n2);
double add(int n1, double n2);
double add(double n1, int n2);
double add(double n1, double n2);
```

Funciones Matemáticas

Las funciones matemáticas más comunes se encuentran en el encabezado `cmath`:

`abs`	absolute value
`sin, cos, tan`	Warning: take angle in radians!
`exp, log, log10`	raise e to power, natural log and base 10 log
`sqrt`	
`pow(double base, double exp)`	raise based to power `exp`

See [1] for a full list.

[1] `http://www.cplusplus.com/reference/clibrary/cmath/`

Funciones Matemáticas

Las funciones matemáticas más comunes se encuentran en el encabezado cmath:

`abs`	absolute value
`sin, cos, tan`	Warning: take angle in radians!
`exp, log, log10`	raise *e* to power, natural log and base 10 log
`sqrt`	
`pow(double base, double exp)`	raise based to power exp

See [1] for a full list.

Se debe incluir el encabezado:

```
#include <cmath>
```

al comienzo del programa.

[1] http://www.cplusplus.com/reference/clibrary/cmath/

Recursividad

Definition

funciones recursivas una función que se llama a si misma.

Es similar a una relación de recurrencia, por ejemplo:

$$b_n = nb_{(n-1)}, b_0 = 1$$

la cual entrega el factorial $n!$ de un número n.

Recursividad

Definition

funciones recursivas una función que se llama a si misma.

Es similar a una relación de recurrencia, por ejemplo:

$$b_n = nb_{(n-1)}, b_0 = 1$$

la cual entrega el factorial $n!$ de un número n. En C++:

```
double factorial(const unsigned int n)
{
  if(n > 1)
    return (n * factorial(n - 1));
  else
    return 1;
}
```