Tabla de Contenidos

Arreglos

Definition

arreglo una serie de elementos del mismo tipo ocupando un espacio contiguo de memoria.

El formato para declarar un arreglo es:

```
type name[num_elements];
```

Donde `type` es cualquier tipo de dato y `num_elements` es una constante tipo entero positivo.

Arreglos

Definition

arreglo una serie de elementos del mismo tipo ocupando un espacio contiguo de memoria.

El formato para declarar un arreglo es:

```
type name[num_elements];
```

Donde `type` es cualquier tipo de dato y `num_elements` es una constante tipo entero positivo. Ejemplos:

```
unsigned int lotteryNumbers[7];

double planetMasses[8];

const unsigned int numParticles = 128;
double xPositions[numParticles];
double yPositions[numParticles];
```

Inicialización de arreglos

Al momento de declarar un arreglo es posible inicializar sus elementos de la siguiente forma:

```
unsigned int lotteryNumbers[7] = {16, 3, 28, 9, 24, 10, 8}
```

También es posible omitir el tamaño , por lo tanto se usa el tamaño de la lista:

```
unsigned int lotteryNumbers[] = {16, 3, 28, 9, 24, 10, 8}
```

Accediendo a los elementos

To access an element of an array the format is:

```
name[index]
```

Warning!

En C y C++ los arreglos comienzan en 0. Esto genera confusión al portar algoritmos desarrollados en otros lenguajes como Fortran o Matlab, que comienzan en 1

Accediendo a los elementos

To access an element of an array the format is:

```
name[index]
```

Warning!

En C y C++ los arreglos comienzan en 0. Esto genera confusión al portar algoritmos desarrollados en otros lenguajes como Fortran o Matlab, que comienzan en 1

Por ejemplo, para leer el 3^{rd} elemento de un arreglo:

```
unsigned int third = lotteryNumbers[2];
```

Accediendo a los elementos

To access an element of an array the format is:

```
name[index]
```

Warning!

En C y C++ los arreglos comienzan en 0. Esto genera confusión al portar algoritmos desarrollados en otros lenguajes como Fortran o Matlab, que comienzan en 1

Por ejemplo, para leer el 3^{rd} elemento de un arreglo:

```
unsigned int third = lotteryNumbers[2];
```

Para escribir el 3^{rd} elemento del arreglo:

```
lotteryNumber[2] = 23;
```

Accediendo a los elementos

To access an element of an array the format is:

```
name[index]
```

Warning!

En C y C++ los arreglos comienzan en 0. Esto genera confusión al portar algoritmos desarrollados en otros lenguajes como Fortran o Matlab, que comienzan en 1

Por ejemplo, para leer el 3^{rd} elemento de un arreglo:

```
unsigned int third = lotteryNumbers[2];
```

Para escribir el 3^{rd} elemento del arreglo:

```
lotteryNumber[2] = 23;
```

También es posible acceder a los elementos de un arreglo usando un índice:

```
for(int i = 0; i < 7; ++i)
    std::cout << lotteryNumbers[i] << " ";
```

Accediendo a los elementos

Indices fuera de los límites

Warning!

Se debe tener cuidado con acceder a elementos que no existen. Por ejemplo:

```
const unsigned int numPlanets = 8;
double masses[numPlanets];
for(int i = 0; i <= numPlanets; ++i)
  masses[i] = random();
```

compila OK?

Arreglos Multidimensionales [1]

Los arreglos multidimensionales pueden verse como "arreglos de arreglos".
Por ejemplo:

```
const unsigned int numParticles = 10;
double positions[numParticles][3];  // positions x,y,z
double masses[numParticles];
double centreOfMass[3] = {0.0, 0.0, 0.0};

// Populate arrays with random masses and positions

for(int i = 0; i < numParticles; ++i)
{
  for(int dim = 0; dim < 3; ++dim)
  {
    centreOfMass[dim] += masses[i] * positions[i][dim] /
      numParticles;
  }
}

std::cout << "Centre of mass: " << centreOfMass[0] << " "
```

[1] https://es.wikipedia.org/wiki/Centro_de_masas

Arreglos Multidimensionales

En teoría podemos usar una cantidad infinita de dimensiones. Por ejemplo un arreglo en 3D:

```
bool isingSpins[nX][nY][nZ];
```

En la practica se hace dificil manejar arreglos multidimensionales y al mismo tiempo se debe tomar en cuenta el espacio de memoria requerido

Arreglos Multidimensionales

En teoría podemos usar una cantidad infinita de dimensiones. Por ejemplo un arreglo en 3D:

```
bool isingSpins[nX][nY][nZ];
```

En la practica se hace dificil manejar arreglos multidimensionales y al mismo tiempo se debe tomar en cuenta el espacio de memoria requerido Por ejemplo:

```
std::cout << "Need: " << sizeof(bool) * nX * nY * nZ <<
  " bytes";
```

Strings

Es posible crear arreglos de caracteres de la misma forma:

```
char message[] = { 'P', 'h', 'y', 's', 'i', 'c', 's', ' ',
    'r', 'o', 'c', 'k', 's', '!', '\0'}; // see footnote 1
```

[1] El caracter \0 le dice al compilador el fin de la cadena.

Strings

Es posible crear arreglos de caracteres de la misma forma:

```
char message[] = { 'P', 'h', 'y', 's', 'i', 'c', 's', ' ',
    'r', 'o', 'c', 'k', 's', '!', '\0'}; // see footnote 1
```

C++ también permite escribir cadenas de caracteres de la siguiente forma:

```
char message[] = "Physics rocks!"
```

Otra forma es mediante strings

```
std::string message = "Physics rocks!";
```

Los strings internamente usan cadenas de caracteres pero de manera mas amigable. Para usar strings debemos incluir:

```
#include <string>  // At the top of your file
```

[1] El caracter \0 le dice al compilador el fin de la cadena.

Variables tipo String

Qué podemos hacer con strings?

Initialise

```
std::string firstName = "Bjarne";
std::string lastName("Stroustrup"); // Almost same as above
```

Variables tipo String

Qué podemos hacer con strings?

Initialise

```
std::string firstName = "Bjarne";
std::string lastName("Stroustrup"); // Almost same as above
```

Concatenar

```
std::string fullName = firstName + " " + lastName;
```

Variables tipo String

Qué podemos hacer con strings?

Initialise

```
std::string firstName = "Bjarne";
std::string lastName("Stroustrup"); // Almost same as above
```

Concatenar

```
std::string fullName = firstName + " " + lastName;
```

Leer desde la entrada estándar

```
std::cout << "Enter first name: "
std::cin >> firstName;
```

Punteros

Cada variable vive en una dirección de memoria. Un puntero es un tipo especial de dato que almacena direcciones.

Pointer declaration

La forma de declarar un puntero es:

```
type * name;
```

Le dice al compilador que la variable `name` es un puntero hacia la dirección de una variable del tipo `type`.

Punteros

Cada variable vive en una dirección de memoria. Un puntero es un tipo especial de dato que almacena direcciones.

Pointer declaration

La forma de declarar un puntero es:

```
type * name;
```

Le dice al compilador que la variable `name` es un puntero hacia la dirección de una variable del tipo `type`.
Ejemplos

```
int * pointerToInt;
std::string * pointerToString;
```

Usando Punteros

Inicialización de Punteros

Para inicar punteros podemos usar el operador de referencia: & ("dirección de memoria de").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

La línea 2 indica que:

Usando Punteros

Inicialización de Punteros

Para inicar punteros podemos usar el operador de referencia: & ("dirección de memoria de").

```
1 int upSpins = 10;
2 int * spinsPointer = &upSpins;
```

La línea 2 indica que:

1. Crea un puntero llamado `spinsPointer` que apunta a un `int`.

Usando Punteros

Inicialización de Punteros

Para inicar punteros podemos usar el operador de referencia: & ("dirección de memoria de").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

La línea 2 indica que:

1. Crea un puntero llamado `spinsPointer` que apunta a un `int`.
2. Apunta hacia la dirección de `upSpins`.

Usando Punteros

Inicialización de Punteros

Para inicar punteros podemos usar el operador de referencia: & ("dirección de memoria de").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

La línea 2 indica que:

1. Crea un puntero llamado `spinsPointer` que apunta a un `int`.
2. Apunta hacia la dirección de `upSpins`.

Qué hay en un puntero?

```
std::cout << "Address is: "
  << spinsPointer
  << "\n";
```

Usando Punteros

Inicialización de Punteros

Para inicar punteros podemos usar el operador de referencia: & ("dirección de memoria de").

```
1 int upSpins = 10;
2 int * spinsPointer = &upSpins;
```

La línea 2 indica que:

1. Crea un puntero llamado `spinsPointer` que apunta a un `int`.
2. Apunta hacia la dirección de `upSpins`.

Qué hay en un puntero?

```
std::cout << "Address is: "
  << spinsPointer
  << "\n";
```

Output: `Address is: 0x00CBF748`

Usando Punteros

Leer valores

Para acceder al valor, usar el operador de de-referencia: * ("valor apuntado por").

```
std::cout << "Value is: "
  << *spinsPointer
```

Usando Punteros

Leer valores

Para acceder al valor, usar el operador de de-referencia: * ("valor apuntado por").

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is: 10`

Seteando valores

Para setear el valor también usamos el asterisco:

```
*spinsPointer = 20;
std::cout << "New upSpins: "
  << upSpins
```

Usando Punteros

Leer valores

Para acceder al valor, usar el operador de de-referencia: * ("valor apuntado por").

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is: 10`

Seteando valores

Para setear el valor también usamos el asterisco:

```
*spinsPointer = 20;
std::cout << "New upSpins: "
  << upSpins
```

Output: `New upSpins: 20`

Usando Punteros

```
// my first pointer
#include <iostream>
using namespace std;

int main (){
        int firstvalue, secondvalue;
        int * mypointer;

        mypointer = &firstvalue;
        *mypointer = 10;
        mypointer = &secondvalue;
        *mypointer = 20;
        cout << "firstvalue is " << firstvalue << '\n';
        cout << "secondvalue is " << secondvalue << '\n';
        return 0;
}
```

Punteros

Warning!

El uso de punteros en C++ es riesgoso! Por ejemplo:

```
int * upSpinsPointer;
std::cout << *upSpinsPointer;
```

Se requiere el valor apuntado por `upSpinsPointer`. Hacia adonde apunta? podría ser una dirección válida o basura.

Punteros

Warning!

El uso de punteros en C++ es riesgoso! Por ejemplo:

```
int * upSpinsPointer;
std::cout << *upSpinsPointer;
```

Se requiere el valor apuntado por `upSpinsPointer`. Hacia adonde apunta? podría ser una dirección válida o basura.

Do

Declarar punteros iniciando con un valor de referencia `0`, indica que no está apuntando a una dirección de memoria válida:

```
int * upSpinsPointer = 0;
```

Esta forma se llama, puntero nulo (por ejemplo, el fin de una lista o arreglo de tamaño desconocido). El programa se cae si se trata de de-referenciar.

Memoria Dinámica

Hasta ahora los arreglos usando un tamaño fijo de memoria. Sin embargo, muchas veces es necesario determinar esta cantidad en tiempo de ejecución (una vez iniciado el programa).

Memoria Dinámica

Hasta ahora los arreglos usando un tamaño fijo de memoria. Sin embargo, muchas veces es necesario determinar esta cantidad en tiempo de ejecución (una vez iniciado el programa). Consider:

```
unsigned int spinChainLength;
std::cout << "Enter spin chain length: ";
std::cin >> spinChainLength;
bool spinChain[spinChainLength]; // ERROR!
```

Solamente podemos inicializar arreglos de tamaño *constante*!

Memoria Dinámica

Hasta ahora los arreglos usando un tamaño fijo de memoria. Sin embargo, muchas veces es necesario determinar esta cantidad en tiempo de ejecución (una vez iniciado el programa). Consider:

```
unsigned int spinChainLength;
std::cout << "Enter spin chain length: ";
std::cin >> spinChainLength;
bool spinChain[spinChainLength]; // ERROR!
```

Solamente podemos inicializar arreglos de tamaño *constante*!
Solución: memoria dinámica.

Operadores `new` y `new []`

Para alojar memoria de forma dinámica:

```
  pointer = new type;                  // single variable
  pointer = new type[num_elements]; // array
```

Memoria Dinámica

Hasta ahora los arreglos usando un tamaño fijo de memoria. Sin embargo, muchas veces es necesario determinar esta cantidad en tiempo de ejecución (una vez iniciado el programa). Consider:

```
unsigned int spinChainLength;
std::cout << "Enter spin chain length: ";
std::cin >> spinChainLength;
bool spinChain[spinChainLength]; // ERROR!
```

Solamente podemos inicializar arreglos de tamaño *constante*!
Solución: memoria dinámica.

Operadores `new` y `new[]`

Para alojar memoria de forma dinámica:

```
  pointer = new type;                  // single variable
  pointer = new type[num_elements]; // array
```

Por ejemplo:

```
bool * spinChain = new bool[spinChainLength]; // Good
```

Memoria Dinámica

No olvidar recojer la basura!

Operadores `delete` and `delete[]`

Para liberar memoria dinámica:

```
delete pointer;   // single variable
delete[] pointer; // array
```

Memoria Dinámica

No olvidar recojer la basura!

Operadores `delete` and `delete[]`

Para liberar memoria dinámica:

```
delete pointer;   // single variable
delete[] pointer; // array
```

Don't

No olvidar liberar la memoria dinámica una vez que se ha utilizado.

Memoria Dinámica

No olvidar recojer la basura!

Operadores `delete` and `delete[]`

Para liberar memoria dinámica:

```
delete pointer;   // single variable
delete[] pointer; // array
```

Don't

No olvidar liberar la memoria dinámica una vez que se ha utilizado.

Do

Setear el puntero a 0 despues que se ha liberado la memoria

```
delete[] spinChain;
spinChain = 0;
```

Referencias

Una referencia es similar a un puntero, más limitado y más seguro.

Decalaración e inicialización de una referencia

```
type & name = variable_name;
```

Le dice al compilador que la variable `name` es una referencia hacia una variable existente llamada `variable_name`. Las referencias *no* pueden ser declaradas sin inicialización !

Example:

```
int upSpins = 10;
int & spinsReference = upSpins;
int & downSpins; // Error: cannot be uninitialised
```

Referencias

Una referencia es similar a un puntero, más limitado y más seguro.

Decalaración e inicialización de una referencia

```
type & name = variable_name;
```

Le dice al compilador que la variable `name` es una referencia hacia una variable existente llamada `variable_name`. Las referencias *no* pueden ser declaradas sin inicialización !
Example:

```
int upSpins = 10;
int & spinsReference = upSpins;
int & downSpins; // Error: cannot be uninitialised
```

Usando referencias

Una vez que se declara una referencia, puede ser usado casi igual que una variable.

Punteros vs. referencias

```cpp
#include <iostream>

int main()
{
  int upSpins = 10;
  int downSpins = 7;
  int * spinsPointer = &upSpins;

  std::cout << "Address is: "
    << spinsPointer
    << "\n";

  std::cout << "Value is: "
    << *spinsPointer
    << "\n";

  *spinsPointer = 20;
  std::cout << "New upSpins: "
    << upSpins
    << "\n";

  spinsPointer = &downSpins;
  std::cout << "Down spins : "
    << *spinsPointer;

  return 0;
}
```

```cpp
#include <iostream>

int main()
{
  int upSpins = 10;
  int downSpins = 7;
  int & spinsReference = upSpins;

//  std::cout << "Address is: "
//    << spinsReference
//    << "\n";

  std::cout << "Value is: "
    << spinsReference
    << "\n";

  spinsReference = 20;
  std::cout << "New upSpins: "
    << upSpins
    << "\n";

//  spinsReference = downSpins;
//  std::cout << "Down spins : "
//    << spinsReference;

  return 0;
}
```

Cambiando el valor en una función

Consider:

```
void runningSum(int sum, int value)
{
  sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i < 100; ++i)
    runningSum(sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is: 0`

Paso de parámetros con punteros

```
void runningSum(int * sum, int
    value)
{
  *sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i <= 100; ++i)
    runningSum(&sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is: 5050`

Paso de parámetros con referencias

```
void runningSum(int & sum, int
    value)
{
  sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i <= 100; ++i)
    runningSum(sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is: 5050`

Paso por valor, puntero y referencia

tres formas de pasar parámetros:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

Paso por valor, puntero y referencia

tres formas de pasar parámetros:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

1. I make an exact duplicate and give it to you. Any changes you make to yours don't affect mine.

Paso por valor, puntero y referencia

tres formas de pasar parámetros:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

1. I make an exact duplicate and give it to you. Any changes you make to yours don't affect mine.
2. I give you my address and you can view and change the painting by visiting (dereferencing) my address.

Paso por valor, puntero y referencia

tres formas de pasar parámetros:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

1. I make an exact duplicate and give it to you. Any changes you make to yours don't affect mine.
2. I give you my address and you can view and change the painting by visiting (dereferencing) my address.
3. I create a second painting that is quantum entangled with mine. Any changes you make to yours affect mine instantly.

Vectores (C++11)

Definition

Vector es un arreglo que puede cambiar de tamaño

Al igual que los arreglos, los vectores almacenan elementos en espacios contiguos de memoria.

```
vector<type> name;
```

Donde `type` es cualquier tipo de dato.

Vectores (C++11)

Definition

Vector es un arreglo que puede cambiar de tamaño

Al igual que los arreglos, los vectores almacenan elementos en espacios contiguos de memoria.

```
vector<type> name;
```

Donde type es cualquier tipo de dato. Ejemplos:

```
std::vector<int> first;     // empty vector of ints
std::vector<int> second (4,100); // four ints with
    value 100
std::vector<int> third
    (second.begin(),second.end());  // iterating
    through second
std::vector<int> fourth (third); // a copy of third
```

Vectores (C++11)

```
// the iterator constructor can also be used to
    construct from arrays:
int myints[] = {16,2,77,29};
std::vector<int> fifth (myints, myints +
    sizeof(myints) / sizeof(int) );

std::cout << "The contents of fifth are:";
for (std::vector<int>::iterator it = fifth.begin();
    it != fifth.end(); ++it)
std::cout << ' ' << *it;
std::cout << '\n';
// The contents of fifth are: 16 2 77 29
```

Vectores (C++11)

```
// vector::push_back
#include <iostream>
#include <vector>

int main (){
        std::vector<int> myvector;
        int myint;
        std::cout << "Please enter some integers (enter 0
            to end):\n";
        do {
                std::cin >> myint;
                myvector.push_back (myint);
        } while (myint);
        std::cout << "myvector stores " <<
            int(myvector.size()) << " numbers.\n";
        return 0;
}
```

Paso de parámetros con referencias

```
// resizing vector
#include <iostream>
#include <vector>

int main ()
{
std::vector<int> myvector;

// set some initial content:
for (int i=1;i<10;i++)
    myvector.push_back(i);

myvector.resize(5);
myvector.resize(8,100);
myvector.resize(12);

std::cout << "myvector contains:";
for (int i=0;i<myvector.size();i++)
std::cout << ' ' << myvector[i];
std::cout << '\n';

return 0;
}
```

Output:

```
myvector contains: 1 2 3 4 5
100 100 100 0 0 0 0
```