# Costruttori e distruttori

# Costruttori

- Quando si crea un oggetto di una classe è possibile inizializzarne i membri tramite il *costruttore di classe*
- Un costruttore è una funzione membro speciale di una classe
- Il nome del costruttore e' lo stesso del nome della classe
- I costruttori non restituiscono nessun valore (nemmeno void)
- Il costruttore o e' scritto esplicitamente dal programmatore
- ..oppure il compilatore crea un costruttore di default che copia uno ad uno tutti i dati membro

# Inizializzazione nei Costruttori

- I dati membro non possono essere inizializzati esplicitamente all'interno della definizione di una classe

```
class Time {
  private:
  int hour=23;
  int min=12;
};
```

- I dati membro si possono (devono) inizializzare all'interno del costruttore
- Assicurarsi sempre di inizializzare i dati membro con valori significativi
  - in particolare i puntatori dovrebbero essere inizializzati a NULL o meglio a 0

# Costruttori di default

- Si possono avere diversi costruttori per casi differenti di inizializzazione utilizzando l'overloading
- Si possono (dovrebbero) utilizzare costruttori con argomenti di default
- Se tutti gli argomenti del costruttore hanno un default questo diventerà il *costruttore di default* ovvero il costruttore che viene invocato quando non sono specificati inizializzatori (ne può esistere solo uno)
- I valori di default dovrebbero essere dichiarati solo nei prototipi

```
class Nome {
  public:
  int f(int=2,double=3.14);
};
```

# Usare Set con i Costruttori

- Conviene creare una funzione Set per attribuire un insieme di valori ai dati membri (con controllo sui valori ammessi)
- Conviene utilizzare una chiamata a questa funzione all'interno del costruttore
- In questo modo non si duplica codice
- Per rendere il tutto più efficiente si può dichiarare la funzione Set inline (ad esempio definendola all'interno della classe)

# Esempio

```
#ifndef TIME2_H
#define TIME2_H
// Time abstract data type definition
class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();       // print military time format
   void printStandard();           // print standard time
   format
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};
#endif
```

```
// Member function definitions for Time class.
#include <iostream>
#include "time2.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour   = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}
```

# Distruttore

- Un distruttore è una funzione membro speciale di una classe
- Il nome del distruttore è composto da una tilde “~” seguito dal nome della classe (è una specie di complemento del costruttore)
- Un distruttore non riceve parametri e non restituisce alcun valore (nemmeno void)
- Una classe può avere un solo distruttore
  - non è ammesso overloading
  - non ha parametri e pertanto il meccanizmo di overloading non potrebbe disambiguare distruttori diversi

# Distruttore

- Il distruttore viene invocato quando l'esecuzione del programma olterpassa lo scope in cui l'oggetto è istanziato
- Oppure esplicitamente su oggetti in memoria dinamica tramite l'operatore delete

- Il distruttore effettua delle operazioni sulla memoria occupata dall'oggetto prima di restituire tale memoria al sistema in modo che possa essere riutilizzata per memorizzare nuovi oggetti

# Esempio:

```
#include <iostream>

class Stack{
   public:
   Stack(int);
   ~Stack();
   void push(int);

   private:
   int *array;
   int dim;
   int tos;
};
```

```
Stack::Stack(int num){
   tos=0;
   dim=num;
   array=new int[dim];
}

Stack::~Stack(){
    delete [] array;
    }

void Stack::push(int n){
   if(tos<dim) {
        array[tos]=n;
        tos++;
   }
}
int main(){
   Stack v(10);
   for(int i=0;i<10;i++)
        v.push(i);
   return 0;
}
```

# Quando sono chiamati i costruttori e i distruttori?

- Le chiamate a costruttori e distruttori (per le variabili automatiche) sono automatiche
- Il loro ordine dipende dall'ordine in cui il flusso di esecuzione entra ed esce dallo scope in cui sono istanziati gli oggetti
- In generale i distruttori sono chiamati in ordine inverso rispetto ai costruttori
  - ...ma il tipo di memorizzazione (es. static) può alterare tale ordine

# Quando sono chiamati i costruttori e i distruttori?

- Oggetti globali:
  - i costruttori degli oggetti che hanno scope globale sono chiamati prima di ogni altra funzione (main inclusa)
  - i distruttori corrispondenti sono chiamati quando main termina o viene invocata la funzione exit()

# Quando sono chiamati i costruttori e i distruttori?

- Oggetti locali:
  - i costruttori sono chiamati per gli oggetti locali nel momento in cui l'esecuzione raggiunge il punto in cui sono definiti
  - i distruttori sono chiamati quando l'esecuzione oltrepassa lo scope degli oggetti in questione, ovvero al termine del blocco in cui sono definiti

# Quando sono chiamati i costruttori e i distruttori?

- Oggetti static:
  - i costruttori sono chiamati solo una volta: la prima volta in cui l'esecuzione raggiunge il punto in cui sono definiti
  - il distruttore viene chiamato quando main termina o quando viene invocata la exit()

# Esempio

```
// Definition of class CreateAndDestroy.
// Member functions defined in create.cpp.
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};

#endif
```

```
// Member function definitions for class CreateAndDestroy
#include <iostream>
#include "create.h"

CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy(){
   cout << "Object " << data << "   destructor " << endl;
}
```

```
#include <iostream>
#include "create.h"
void create( void );   // prototype
CreateAndDestroy first( 1 );  // global object
int main()
{
   cout << "   (global created before main)" << endl;
   CreateAndDestroy second( 2 );        // local object
   cout << "   (local automatic in main)" << endl;
   static CreateAndDestroy third( 3 );  // local object
   cout << "   (local static in main)" << endl;
   create();  // call function to create objects
   CreateAndDestroy fourth( 4 );        // local object
   cout << "   (local automatic in main)" << endl;
   return 0;
}
void create( void ){
   CreateAndDestroy fifth( 5 );
   cout << "   (local automatic in create)" << endl;
   static CreateAndDestroy sixth( 6 );
   cout << "   (local static in create)" << endl;
   CreateAndDestroy seventh( 7 );
   cout << "   (local automatic in create)" << endl;
}
```

# Esempio

- Output:
  - Object 1 constructor (global created before main)
  - Object 2 constructor (local automatic in main)
  - Object 3 constructor (local static in main)
  - Object 5 constructor (local automatic in create)
  - Object 6 constructor (local static in create)
  - Object 7 constructor (local automatic in create)
  - Object 7 destructor
  - Object 5 destructor
  - Object 4 constructor (local automatic in main)
  - Object 4 destructor
  - Object 2 destructor
  - Object 6 destructor
  - Object 3 destructor
  - Object 1 destructor