Shock the monkey (Un'introduzione a C++)

Federico Terraneo

18 marzo 2012

Introduzione: caratteristiche del linguaggio

C++ è un linguaggio:

- multiparadigma (OO, procedurale, metaprogrammativo)
- compilato (niente interpreti, niente virtual machine)
- con una sintassi C-like (infatti deriva dal C ed quasi pienamente retrocompatibile)

Compilatori e ambienti di sviluppo sono disponibili per la maggior parte delle architetture

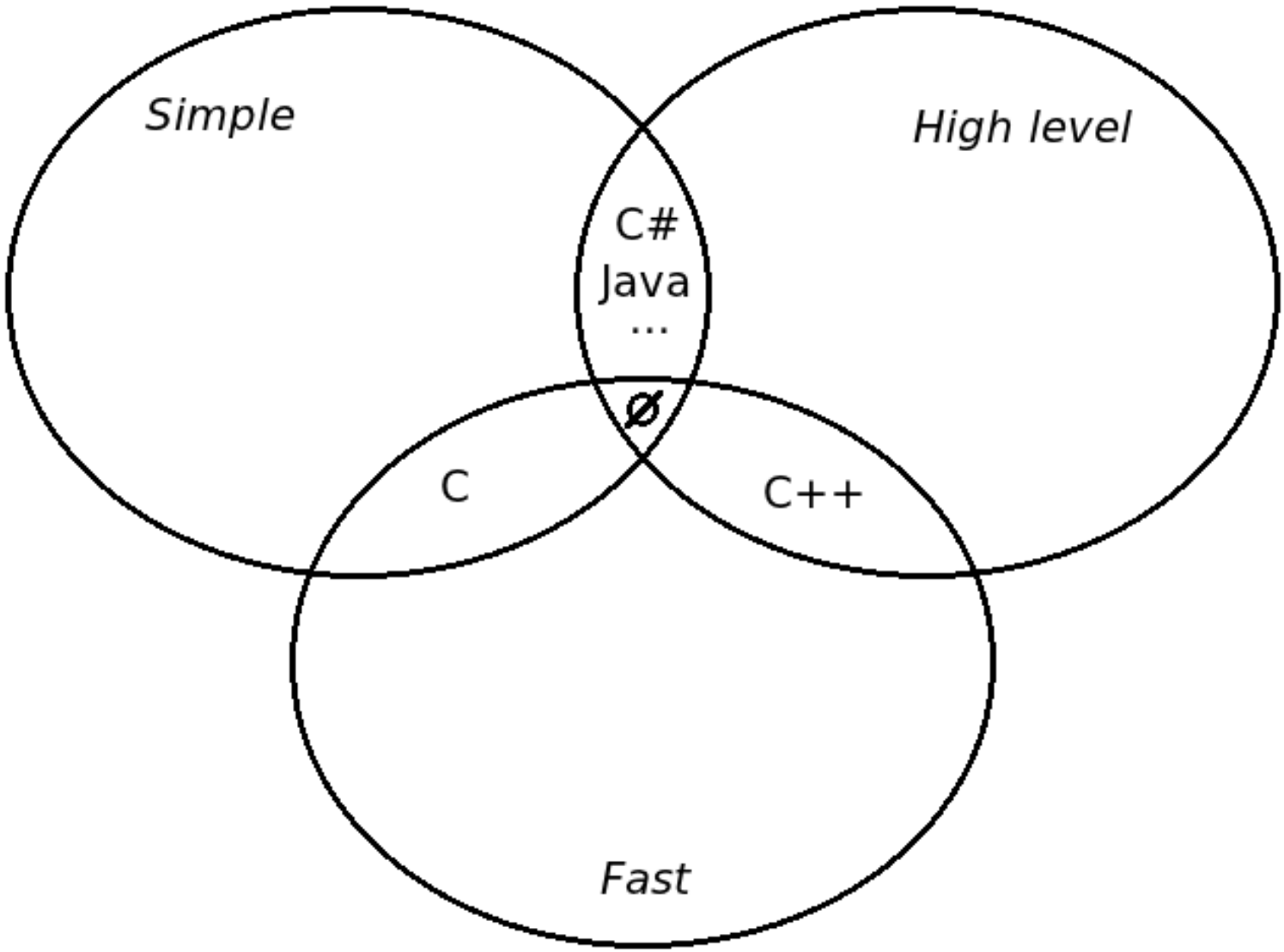
- desktop/server (Linux, Windows, Mac, ...)
- mobile (Android, iOS, MeeGo, Symbiam, ...)
- embedded/microcontrollori (Arduino, ...)
- e per molti altri ancora, quali game consoles, ...

Introduzione: caratteristiche del linguaggio

Come scegliere un linguaggio di programmazione?

Fast, simple, high level, pick any two.

Introduzione: caratteristiche del linguaggio



Introduzione: due parole su queste slide

Non è possibile insegnare un qualunque linguaggio di programmazione in due ore, a maggior ragione C++.L'obbiettivo di queste slide è quindi di mettervi in grado di guardare un file .cpp capendo più o meno cosa c'è scritto.

Queste slide partono dal presupposto che si conosca già il C e la programmazione procedurale.

Verranno introdotte le principali nuove feature presenti in C++ e assenti in C

Ampio spazio verrà dato al paradigma di programmazione a oggetti

In un certo senso, queste slide sono l'output di un

```
diff -ruN c c++
```

Sommario

- una parentesi sul C: suddivisione di un programma in più file
- da C a C++
  - hello world
  - input/output stream
  - namespace
  - stringhe
  - new/delete
  - references
  - eccezioni
  - template
  - STL
- programmazione object oriented
  - abstract data type
  - evoluzione delle struct
  - costruttori e distruttori
  - allocazione nello stack e nell'heap
  - public e private

Una parentesi sul C: suddivisione di un programma in più file

Spesso, quando si scrive un semplice programma in C lo si scrive tutto in un solo file.

Quest'approccio però non scala al crescere della dimensione dei programmi, perchè:

- Programmi di grandi dimensioni vengono di solito divisi logicamente in moduli, che svolgono una ben precisa funzione, come gestire l'interfaccia utente, la logica dell'applicazione, la lettura di file di configurazione ecc. Fare in modo che a questa suddivisione logica (moduli) corrisponda una suddivisione fisica (ogni modulo in un file) semplifica la comprensione e la manutenibilità del codice.
- Dividere un progetto in file consente di ridurre la memoria RAM occupata dal compilatore, questo è particolarmente importante per progetti di grosse dimensioni
- Dividere un progetto in file semplifica la scrittura di codice da parte di più persone, anche in presenza di sistemi di version control (come git, svn, ...)

Una parentesi sul C: suddivisione di un programma in più file

Suddividere un progetto in più file richiede di usare due tipi di file, con estensione .c e .h

File .c "file sorgente c"
Questi file contengono

- funzioni chiamabili da qualunque file sorgente del progetto
- funzioni chiamabili solo all'interno del file (dichiarate static)
- dichiarazione di struct, enum, typedef e macro "private", ossia usabili solo all'interno del file

File .h "header file"

- prototipi delle funzioni chiamabili da qualunque file sorgente del progetto
- dichiarazione di struct, enum, typedef e macro "pubbliche", ossia chiamabili da qualunque file sorgente del progetto

Una parentesi sul C: suddivisione di un programma in più file

Un semplice esempio di suddivisione di un programma in più file

main.c

```c

#include "test.h"

int main()
{
    struct Test t;
    t.a=10;
    t.b=20;
    testFunc(t);
}

```

test.h

```c

#ifndef TEST_H
#define TEST_H

struct Test
{
    int a, b;
};

void testFunc(Test t);

#endif //TEST_H

```

test.c

```c

#include "test.h"
#include <stdio.h>

void testFunc(struct Test t)
{
    printf("%d %d\n",t.a,t.b);
}

```

Compilare con

```
gcc -O2 -c main.c              # Compila main.c e genera main.o
gcc -O2 -c test.c              # Compila test.c e genera test.o
gcc -o prog main.o test.o      # Linka i file.o generando l'eseguibile prog
```

da C a C++: hello world

L'hello world in C++ confrontato con l'equivalente in C

example.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout<<"Hello world"<<endl;
}
```

example.c

```c
#include <stdio.h>

int main()
{
    printf("Hello world\n");
}
```

Compilare con

```
g++ -O2 example.cpp -o example-cpp   # g++ è il compilatore C++ su Linux
gcc -O2 example.c   -o example-c     # gcc è il compilatore C su Linux
```

Nota: poichè C++ è retrocompatibile con il C, il contenuto del file example.c è anche codice C++ valido (in un programma C++ si può usare printf, etc.).

da C a C++: input/output stream

C++ introduce il concetto di stream, un'astrazione di un device in cui si può scrivere e/o da cui si può leggere.
Il concetto di stream consente di modellizzare

- la console: cout, cin (sostituisce printf/scanf del C)
- i file: ifstream, ofstream (sostituisce fopen/fclose del C)
- la memoria, per effettuare operazioni su stringhe: stringstream (sostituisce sprintf/sscanf del C)

Uso degli stream

- per scrivere in uno stream stringhe, interi, float, etc... si usa l'operatore «
- per andare a capo quando si sta scrivendo si usa `endl`
- per leggere interi, float, etc... l'operatore »
- per leggere una linea di testo in una stringa si usa la funzione `getline()`.

da C a C++: input/output stream

Esempio di uso degli stream

```
1
2   #include <iostream>
3
4   using namespace std;
5
6   int main()
7   {
8       int i;
9       float f;
10      string s;
11      cin>>i>>f;
12      getline(cin,s);
13      cout<<"Intero"<<i<<endl;
14      cout<<"Float"<<f<<endl;
15      cout<<"Stringa"<<s<<endl;
16  }
17
```

Nota: contrariamente al C non è necessario specificare il tipo di quello che si sta per stampare o leggere (come %d per gli interi in printf). Il compilatore lo riconosce automaticamente.

da C a C++: namespace

I namespace servono a risolvere i conflitti nei nomi, consentendo a due funzioni identiche, con lo stesso nome e gli stessi parametri di coesistere nello stesso programma.

```
1
2   #include <iostream>
3
4   namespace nam1 {
5
6   void f()
7   {
8       std::cout<<"f() nel namespace 1"<<std::endl;
9   }
10
11  }
12
13  namespace nam2 {
14
15  void f()
16  {
17      std::cout<<"f() nel namespace 2"<<std::endl;
18  }
19
20  }
21
22  int main()
23  {
24      nam1::f();
25      nam2::f();
26  }
27
```

da C a C++: namespace

- Un namespace si dichiara con `namespace <nome> { [...] }`
- All'interno di un namespace si possono dichiarare funzioni, classi, enum, typedef, costanti...
- Per accedere a un elemento `foo` in un namespace `bar` occorre aggiungere a `foo` il prefisso `bar::`...
- ...oppure aggiungere al file corrente un `using namespace bar;`
- Le funzioni/classi della libreria standard del C++ si trovano nel namespace `std`
- E' considerata una cattiva pratica di programmazione mettere un `using namespace` in un header file (.h), in quanto i programmatori che includeranno quel file sono costretti a tenersi un using namespace che magari non vogliono.

da C a C++: stringhe

Contrariamente al C dove le stringhe sono array di caratteri, in C++ esiste il tipo stringa

stringhe.cpp

```cpp

#include <iostream>

using namespace std;

int main()
{
    string a,b;
    a="Hello";     //Inizializzazione
    a+=" world!"; //Concatenamento
    b=a;           //Assrgnamento
    cout<<a<<endl;
}
```

stringhe.c

```c

#include <stdio.h>
#include <string.h>

int main()
{
    char a[1024], b[1024];
    strcpy(a,"Hello");      //Inizializzazione
    strcat(a," world!");    //Concatenamento
    strcpy(b,a);            //Assegnamento
    printf("%s\n",b);
}
```

Notate come in C è necessario definire una dimensione massima per le stringhe, mentre in C++ le stringhe si ridimensionano automaticamente. Le stringhe non sono tipi base come int e float, sono delle classi. Come vedremo in seguito, è possibile scrivere le proprie classi così come in C si possono scrivere funzioni.

Quindi in C++ esistono altri tipi di stringhe oltre a quelle standard, però queste classi conservano la stessa interfaccia di quelle standard, come l'uso di = per l'assegnamento e + per il concatenamento.

da C a C++: new/delete

In C si usa `malloc()` per allocare memoria nello heap, e `free()` per deallocarla.
In C++ si usa `new` per allocare memoria e `delete` per deallocarla.

heap.cpp

```cpp
int main()
{
    //Allocazione di un intero
    int *a=new int;
    delete a;

    //Allocazione di un array di 10 interi
    int *b=new int[10];
    delete[] b;
}
```

heap.c

```c
#include <stdlib.h>

int main()
{
    //Allocazione di un intero
    int *a=(int*)malloc(sizeof(int));
    free(a);

    //Allocazione di un array di 10 interi
    int *b=(int*)malloc(10*sizeof(int));
    free(b);
}
```

da C a C++: reference

In C gli argomenti vengono passati alle funzioni per copia, quindi per modificare un argmento passato occorre passarlo per puntatore.
In C++ esiste una sintassi più elegante, le reference.

pass-by-reference.cpp

```cpp
#include <iostream>

using namespace std;

void incrementa(int& intero)
{
    intero++;
}

int main()
{
    int i=0;
    incrementa(i);
    cout<<i<<endl;
}
```

pass-by-reference.c

```c
#include <stdio.h>

void incrementa(int *intero)
{
    (*intero)++;
}

int main()
{
    int i=0;
    incrementa(&i);
    printf("%d\n",i);
}
```

da C a C++: reference

- Le reference si possono usare anche al di fuori dei parametri di una funzione (vedi esempio)
- Una volta che una reference `j` "punta" a una variabile `i` non è possibile farla puntare a nessun altro oggetto, contrariamente ai puntatori
- Non esistono reference a `NULL`, una reference punta sempre a un vero oggetto

```
1
2   #include <iostream>
3
4   using namespace std;
5
6   int main()
7   {
8       int i=0;
9       int& j=i;
10      cout<<j<<endl;
11      i=10;
12      cout<<j<<endl;
13  }
14
```

da C a C++: eccezioni

C non ha dei costrutti del linguaggio specifici per la gestione degli errori.
Solitamente le funzioni di libreria ritornano uno specifico valore in caso di errore.
Esempio: `malloc()` ritorna `NULL` se non c'è memoria libera allocabile.

I problemi di questo approccio sono:

- E' facile *dimenticarsi* di controllare i codici di errore. Per esempio, `printf()` ritorna un numero negativo in caso di errore, ma chi controlla mai il valore di ritorno di printf?
- Se effettivamente ci si volesse mettere d'impegno e controllare tutti i codici d'errore occorrerebbe scrivere *molte* più linee di codice in un programma.

In C++ esiste un costrutto, le eccezioni, che consente di separare la parte di codice che tratta il caso in cui va tutto bene da quello in cui capita un errore.

da C a C++: eccezioni

Esempio di gestione degli errori in C e C++. Si noti come non è necessario gestire esplicitamente l'errore nella funzione intermedia, ma ciònonostante l'errore non venga ignorato.

errorhandling.cpp

```cpp
#include <iostream>
#include <stdexcept>

using namespace std;

int g(int i)
{
   static const int table[]={10,20,30};
   if(i<0 || i>=3) throw range_error("index out of bounds");
   return table[i];
}

void f(int i)
{
    cout<<"Risultato:"<<g(i)<<endl;
}

int main()
{
    try {
        int i;
        cin>>i;
        f(i);
    } catch(exceprion& e)
    {
        cout<<"Errore:"<<e.what();<<endl;
    }
}
```

errorhandling.c

```c
#include <stdio.h>

int g(int i)
{
   static const int table[]={10,20,30};
   if(i<0 || i>=3) return -1; //-1 = Codice di errore
   return table[i];
}

int f(int i)
{
    int result=g(i);
    if(result>=0)
    {
        printf("Risultato:%d\n",result);
        return 0; //0 = Tutto OK
    } else {
        return -1; //-1 = Codice di errore
    }
}

int main()
{
    int i;
    scanf("%d",&i);
    int result=f(i);
    if(result<0)
    {
        printf("Errore\n");
    }
}
```

da C a C++: template

I template sono la feature di C++ che consente la metaprogrammazione. Con i template è possibile scrivere codice (funzioni, classi) "parametriche" in uno o più tipi di dati, in grado di essere "istanziate" quando vengono usate con un tipo dati concreto.

```
1
2   #include <iostream>
3
4   using namespace std;
5
6   template<typename T> bool max(T a, T b)
7   {
8       if(a>b) return a;
9       else return b;
10  }
11
12  int main()
13  {
14      int x=10, y=20;
15      cout<<max(x,y)<<endl; //max() viene istanziata per T=int
16
17      float v=1.0f, w=-1.0f;
18      cout<<max(v,w)<<endl; //max() viene istanziata per T=float
19  }
20
```

da C a C++: template

I template sono uno degli argomenti più complessi di C++, ma consentono di generare codice molto ottimizzato in quanto sono risolti interamente a `compile time`.

Nella loro forma più generale i template rappresentano un linguaggio a sè stante, turing completo, che viene interpretato dal compilatore C++ e il cui output è a sua volta codice C++.

da C a C++: STL

La STL, o standard template library, è una libreria di `contenitori`, *algoritmi* e `iteratori` interamente a template.

- Contenitori: sono strutture dati comunemente usate nei programmi, quali liste, vettori a dimensione dinamica, hashtable, etc...
- Algoritmi: una collezione di algoritmi comuni da applicare ai contenitori quali sorting, etc...
- Iteratori: rappresentano modi di iterare gli elementi dei contenitori. Sono anche usati dagli algoritmi in modo da riusare un algoritmo per più contenitori

da C a C++: STL

Esempio di codice che stampa un file al contrario.

```
1
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5
6  using namespace std;
7
8  int main(int argc, char *argv[])
9  {
10     if(argc<2) return 1; //Nessun argomento passato al programma
11     ifstream in(argv[1]);
12     if(!in.good()) return 1; //File non trovato
13
14     vector<string> v;
15     string line;
16     while(getline(in,line)) v.push_back(line);
17
18     for(int i=v.size();i>=0;i--) cout<<v[i]<<endl;
19 }
20
```

Si noti l'uso di `vector<>` per memorizzare le linee di testo del file. `vector<>` è una classe template, che viene specializzata in `vector<string>` in modo da avere un vettore di stringhe. Contariamente a un array, un `vector<>` non ha una dimensione prefissata, si ingrandisce da solo man mano che viene chiamato il metodo `push_back()`.

programmazione object oriented: abstract data type

Più volte in queste slide si è parlato di *classi*, senza spiegarne il significato. Ma cosa sono queste classi?

Le classi sono la base di un *paradigma di programmazione*, detto *Object Oriented*.

Il C è si basa sul paradigma *procedurale*, in cui la logica del codice viene partizionata in *funzioni*.

Nella programmazione OO invece, la logica viene partizionata in *classi*

programmazione object oriented: abstract data type

Ok, ma cos'è una classe? Una classe è un *abstract data type*, una unità base per lo sviluppo di software composta da

- Dati, equivalenti alle variabili in C
- Metodi, all'incirca come le funzioni in C

La differenza rispetto alla programmazione procedurale è

- che i dati e i metodi fano parte di un unica entità, la classe invece di essere separati in struct e funzioni
- il concetto di *incapsulamento*: i dati di una classe sono accessibili solo tramite i suoi metodi.
- i concetti di ereditarietà e polimorfismo, che consentono di specializzare in modo incrementale i tipi dati, evitando di duplicare il codice relativo a parti comuni

programmazione object oriented: evoluzione delle struct

Per introdurre la programmazione object oriented "by examples", consideriamo una applicazione C che necessita di un buffer circolare di interi.

Il buffer ha una dimensione fissata durante l'inizializzazione.

L'aggiunta e la rimozione degli elementi dal buffer viene fatta secondo la politica fifo.

Una possibile implementazione in C è la seguente:

programmazione object oriented: evoluzione delle struct

queue.c

```c
#include <stdlib.h>
#include "queue.h"

int *data;
int max_size, size, put_pos, get_pos;

void queue_init(int length)
{
    data=malloc(length*sizeof(int));
    max_size=length;
    put_pos=get_pos=size=0;
}

int queue_put(int value)
{
    if(size==max_size) return -1; //Error code
    data[put_pos]=value;
    if(++put_pos==max_size) put_pos=0;
    size++;
    return 0;
}

int queue_get(int *value)
{
    if(size==0) return -1; //Error code
    *value=data[get_pos];
    if(++get_pos==max_size) get_pos=0;
    size--;
    return 0;
}

void queue_done()
{
    free(data);
}
```

queue.h

```c
#ifndef QUEUE_H
#define QUEUE_H

void queue_init(int length)

int queue_put(int value);

int queue_get(int *value);

void queue_done();

#endif //QUEUE_H
```

programmazione object oriented: evoluzione delle struct

L'implementazione è stata fatta in una coppia di file .c e .h in modo da separare questo modulo dal resto del programma.

Notate come l'implementazione faccia uso di variabili globali.

Immaginiamo ora nell'applicazione serva più di un buffer circolare, come è possibile generalizzare il codice?

programmazione object oriented: evoluzione delle struct

Risposta: passando dalle variabili globali ad una struct.

queue2.c

```c
#include <stdlib.h>
#include "queue.h"

void queue_init(struct queue *q, int length)
{
    q->data=malloc(length*sizeof(int));
    q->max_size=length;
    q->put_pos=q->get_pos=q->size=0;
}

int queue_put(struct queue *q, int value)
{
    if(q->size==q->max_size) return -1; //Error code
    q->data[q->put_pos]=value;
    if(++q->put_pos==q->max_size) q->put_pos=0;
    q->size++;
    return 0;
}

int queue_get(struct queue *q, int *value)
{
    if(q->size==0) return -1; //Error code
    *value=q->data[q->get_pos];
    if(++q->get_pos==q->max_size) q->get_pos=0;
    q->size--;
    return 0;
}

void queue_done(struct queue *q)
{
    free(q->data);
}

```

queue2.h

```c
#ifndef QUEUE_H
#define QUEUE_H

struct queue
{
    int *data;
    int max_size;
    int size;
    int put_pos;
    int get_pos;
};

void queue_init(struct queue *q, int length);

int queue_put(struct queue *q, int value);

int queue_get(struct queue *q, int *value);

void queue_done(struct queue *q);

#endif //QUEUE_H

```

programmazione object oriented: evoluzione delle struct

Per quanto non siate abituati a pensarci, questo codice definisce un nuovo tipo dati, la coda.

Infatti, considerando un tipo dati come int, e la coda, si notano molte somiglianze

- Così come si possono creare molte istanze di int, la stessa cosa si può fare con la coda
- Sia int che la coda hanno delle operazioni applicabili, nel caso di int le operazioni sono $+, -, *, /, \ldots$ e sono definite dal compilatore, nel caso della coda sono `init`,`put`,`get`,`done` e sono definite dal programmatore che ha scritto queue2.c

programmazione object oriented: evoluzione delle struct

La coda è un tipo dati astratto, è una classe?

No, perchè

- I dati (la struct `queue`) sono separati dalle operazioni, le quattro funzioni
- Non esiste `incapsulamento`, in quanto il codice che usa la coda può indifferentemente accederci tramite le funzioni, direttamente tramite i campi della struct, o definirsi delle proprie funzioni

programmazione object oriented: evoluzione delle struct

Passiamo ora da C a C++, introducendo la classe coda.

queue3.cpp

```
1
2  #include <stdexcept>
3  #include "queue.h"
4
5  void Queue::init(int length)
6  {
7      this->data=new int[length];
8      this->max_size=length;
9      this->put_pos=this->get_pos=this->size=0;
10 }
11
12 void Queue::put(int value)
13 {
14     if(this->size==this->max_size) throw std::range_error("");
15     this->data[this->put_pos]=value;
16     if(++this->put_pos==this->max_size) this->put_pos=0;
17     this->size++;
18 }
19
20 int Queue::get()
21 {
22     if(this->size==0) throw std::range_error("");
23     int result=this->data[this->get_pos];
24     if(++this->get_pos==this->max_size) this->get_pos=0;
25     this->size--;
26     return result;
27 }
28
29 void Queue::done()
30 {
31     delete[] this->data;
32 }
```

queue3.h

```
1
2  #ifndef QUEUE_H
3  #define QUEUE_H
4
5  class Queue
6  {
7  public:
8      void init(int length);
9
10     void put(int value);
11
12     int get();
13
14     void done();
15
16 private:
17     int *data;
18     int max_size;
19     int size;
20     int put_pos;
21     int get_pos;
22 };
23
24 #endif //QUEUE_H
25
```

programmazione object oriented: evoluzione delle struct

Analizziamo uno ad uno i cambiamenti avvenuti nel passaggio da C a C++:
queue.h

- Queue è stata dichiarata come `class` anzichè `struct`
- Sono comparse le keyword `public` e `private` per definire l'incapsulamento, come verrà spiegato in seguito.
- le funzioni `init`,`put`,`get`,`done` sono diventati metodi, e sono stati dichiarati all'interno della classe.
- nel passaggio da funzioni a metodi, è scomparso il parametro `struct queue *q`, che è diventato *implicito*

queue.cpp

- L'implementazione dei metodi è simile a delle funzioni, ma con nome pari a `NOMECLASSE::NOMEMETODO`, esempio `Queue::put`
- L'accesso ai dati della classe (`data`,`max_size`,`...` avviene tramite l'uso di `this->`, una parola chiave di C++ per accedere ai dati di un oggetto (in realtà in molti casi si può omettere, ma è stato usato per chiarezza)
- Sono state usate le eccezioni e `new/delete` al posto dei codici di errore e `malloc/free`

programmazione object oriented: evoluzione delle struct

Come si usa la classe coda?

queue-usage.c

```c
#include <stdio.h>
#include "queue.h"

int main()
{
    queue q;
    queue_init(&q,10);

    queue_put(&q,1);
    queue_put(&q,2);
    queue_put(&q,3);

    int i;
    queue_get(&q,&i);
    printf("%d\n",i);
    queue_get(&q,&i);
    printf("%d\n",i);
    queue_get(&q,&i);
    printf("%d\n",i);

    queue_done(&q);
}
```

queue-usage.cpp

```cpp
#include <iostream>
#include "queue.h"

using namespace std;

int main()
{
    Queue q;
    q.init(10);

    q.put(1);
    q.put(2);
    q.put(3);

    cout<<q.get()<<endl;
    cout<<q.get()<<endl;
    cout<<q.get()<<endl;

    q.done();
}
```

programmazione object oriented: evoluzione delle struct

Analizziamo uno ad uno i cambiamenti avvenuti nel passaggio da C a C++:

- Le istanze di una classe si chiamano *oggetti*. Si dichiarano con `NOMCLASSE NOMEOGGETTO`, ad esempio `Queue q;`. Notare come la sintassi sia uguale a quella per dichiarare variabili come `int i;`
- I metodi si chiamano con la notazione `NOMEOGGETTO.NOMEMETODO`, ad esempio `q.init(10);`

programmazione object oriented: costruttori e distruttori

Il codice visto fin qui funziona, ma cosa succede se il programmatore si dimenticasse di inizializzare la classe con il metodo init?

- Probabilmente l'applicazione andrebbe in crash a runtime

E cosa succederebbe se il programmatore si dimenticasse di chiamare il metodo done dopo aver usato la classe?

- Ci sarebbe un *memory leak*

Questo problema esiste anche nella versione in C del programma, solo che mentre in C il problema non è eliminabile, in C++ può essere risolto, usando i costruttori e i distruttori.

programmazione object oriented: costruttori e distruttori

La classe `Queue` con in aggiunta un costruttore e il distruttore

queue4.cpp

```
1
2  #include <stdexcept>
3  #include "queue.h"
4
5  Queue::Queue(int length)
6  {
7      this->data=new int[length];
8      this->max_size=length;
9      this->put_pos=this->get_pos=this->size=0;
10 }
11
12 void Queue::put(int value)
13 {
14     if(this->size==this->max_size) throw std::range_error("");
15     this->data[this->put_pos]=value;
16     if(++this->put_pos==this->max_size) this->put_pos=0;
17     this->size++;
18 }
19
20 int Queue::get()
21 {
22     if(this->size==0) throw std::range_error("");
23     int result=this->data[this->get_pos];
24     if(++this->get_pos==this->max_size) this->get_pos=0;
25     this->size--;
26     return result;
27 }
28
29 Queue::~Queue()
30 {
31     delete[] this->data;
32 }
33
```

queue4.h

```
1
2  #ifndef QUEUE_H
3  #define QUEUE_H
4
5  class Queue
6  {
7  public:
8      Queue(int length); //Costruttore
9
10     void put(int value);
11
12     int get();
13
14     ~Queue(); //Distruttore
15
16 private:
17     int *data;
18     int max_size;
19     int size;
20     int put_pos;
21     int get_pos;
22 };
23
24 #endif //QUEUE_H
25
```

programmazione object oriented: costruttori e distruttori

I costruttori sono

- Metodi il cui nome è uguale a quello della classe
- Possono prendere parametri, ma non hanno un valore di ritorno, nemmeno void
- Una classe può avere più costruttori, con diverso numero o tipo di parametri
- L'implementazione del costruttore viene chiamata *automaticamente* quando si crea una istanza di un oggetto, quindi il programmatore può inserirci il codice atto a garantire che l'oggetto sia inizializzato

Il distruttore è

- Un metodo il cui nome è uguale al della classe preceduto da $\sim$
- Un metodo che non prende parametri e non ritorna nulla, nemmeno void
- Una classe può avere al più un distruttore
- L'implementazione del costruttore viene chiamata *automaticamente* quando si distrugge una istanza di un oggetto, quindi il programmatore può inserirci il codice atto a garantire la finalizzazione delle risorse

programmazione object oriented: costruttori e distruttori

Uso della classe `Queue` con e senza costruttore e distruttore

senza-costruttore.cpp

```
1
2  #include <iostream>
3  #include "queue.h"
4
5  using namespace std;
6
7  int main()
8  {
9       Queue q;
10      q.init(10);
11
12      q.put(1);
13      q.put(2);
14      q.put(3);
15
16      cout<<q.get()<<endl;
17      cout<<q.get()<<endl;
18      cout<<q.get()<<endl;
19
20      q.done();
21 }
22
```

con-costruttore.cpp

```
1
2  #include <iostream>
3  #include "queue.h"
4
5  using namespace std;
6
7  int main()
8  {
9       Queue q(10);
10
11      q.put(1);
12      q.put(2);
13      q.put(3);
14
15      cout<<q.get()<<endl;
16      cout<<q.get()<<endl;
17      cout<<q.get()<<endl;
18
19      //Nota: il distruttore viene
20      //chiamato automaticamente
21 }
22
```

programmazione object oriented: allocazione nello stack e nell'heap

Così come è possibile allocare un intero

- Sullo stack, semplicemente con `int i;`
- Nell'heap, con `int *i=new int; ... delete i;`

La stessa cosa si può fare con gli oggetti.

programmazione object oriented: allocazione nello stack e nell'heap

stack-alloc.cpp

```cpp
#include <iostream>
#include "queue.h"

using namespace std;

int main()
{
    Queue q(10);

    q.put(1);
    q.put(2);
    q.put(3);

    cout<<q.get()<<endl;
    cout<<q.get()<<endl;
    cout<<q.get()<<endl;

    //Nota: il distruttore viene
    //chiamato automaticamente
}
```

heap-alloc.cpp

```cpp
#include <iostream>
#include "queue.h"

using namespace std;

int main()
{
    Queue *q=new Queue(10);

    q->put(1);
    q->put(2);
    q->put(3);

    cout<<q->get()<<endl;
    cout<<q->get()<<endl;
    cout<<q->get()<<endl;

    delete q; //Questo chiama il distruttore
}
```

programmazione object oriented: public e private

Una delle caratteristiche principali della programmazione object oriented è l'*incapsulamento*, la possibilità di rendere alcuni dati e/o metodi privati, chiamabili solo dall'interno dela classe. Questo è importante perchè

- Se il resto del programma non può accedere a parti della classe, queste parti possono in futuro essere modificate senza che il resto del codice se ne accorga
- Se il resto del codice del codice non può accedere a parti della classe, non può commettere errori nell'uso di quei dati e il debugging è semplificato

C++ mette a disposizione le keyword `public` e `private` per definire quali parti di una classe sono accessibili da tutti e quali no.

Sia metodi che dati possono essere resi `public`, ma è considerata buona pratica di programmazione rendere pubblici solo metodi. L'accesso ai dati privati, se necessario, può essere implementato tramite metodi detti `getter e setter`.

programmazione object oriented: public e private

publicprivate.cpp

```
1
2   #include <iostream>
3
4   class Foo
5   {
6   public:
7       void method1();
8       int var1;
9
10  private:
11      void method2();
12      int var2;
13  };
14
15  int main()
16  {
17      Foo foo;
18      foo.method1(); //OK
19      foo.var1=0;    //OK
20
        foo.method2(); //Error
        foo.var2=0;    //Error
23  }
24
```

da C a C++: cosa manca?

Per ragioni di tempo queste slide hanno omesso altre caratteristiche di C++, quali

- Ereditarietà delle classi, ereditarietà multipla
- `protected`
- Polimorfismo e virtual functions
- Classi virtuali pure
- Il passaggio per reference to const, l'uso in generale di `const` e `mutable`
- Il costruttore di copia e l'overload dell'operator=
- L'overload di funzioni e di operatori
- inlining
- Una trattazione più esaustiva dei template e della STL
- ...

da C a C++: what's next?

Esistono molti libri sul C++, gli interssati al linguaggio sono consigliati di prenderne uno (e leggerlo)

Per imparare più approfonditamente il linguaggio, c'è "Thinking in C++" `http://mindview.net/Books/TICPP/ThinkingInCPP2e.html` (disponibile anche come free download).

Per dubbi relativi al C++, e specialmente alla sua libreria standard è possibile fare riferimento al sito `http://www.cplusplus.com`, molto utile per risolvere dubbi mentre si sta scrivendo del codice.