Destructor (computer programming)

From Wikipedia, the free encyclopedia

In object-oriented programming, a **destructor** (sometimes shortened to **dtor**) is a method which is automatically invoked when the object is destroyed. It can happen when its lifetime is bound to scope and the execution leaves the scope, when it is embedded into another object whose lifetime ends, or when it was allocated dynamically and is released explicitly. Its main purpose is to free the resources (memory allocations, open files or sockets, database connections, resource locks, etc.) which were acquired by the object along its life cycle and/or deregister from other entities which may keep references to it.

È sempre disponibile il **distruttore standard**: invoca il “distruttore” per tutti i campi dati della classe, nell’ordine inverso alla loro dichiarazione

È sempre disponibile il **distruttore standard**: invoca il "distruttore" per tutti i campi dati della classe, nell'ordine inverso alla loro dichiarazione

È accettabile il distruttore standard della classe bolletta?

È sempre disponibile il **distruttore standard**: invoca il "distruttore" per tutti i campi dati della classe, nell'ordine inverso alla loro dichiarazione

È accettabile il distruttore standard della classe bolletta?

NO!

Distruttore della classe **`bolletta`**:

```
class bolletta {
public:
   ....
   ~bolletta();
   ....
};
```

Distruttore della classe **`bolletta`**:

```
class bolletta {
public:
   ....
   ~bolletta();
   ....
};
```

```
//first punta alla testa della lista
bolletta::~bolletta() { distruggi(first); }
```

Le *regole di invocazione dei distruttori* sono le seguenti:

- per gli **oggetti di classe statica**, al termine del programma (all'uscita dalla funzione principale `main`())
- per gli **oggetti di classe automatica** definiti in un blocco (ad esempio una funzione), all'uscita dal blocco in cui sono definiti. In particolare, ciò vale per i parametri formali di una funzione
- per gli **oggetti dinamici** (allocati sullo heap), quando viene invocato l'operatore di **`delete`** su un puntatore ad essi.
- per gli oggetti che sono **campi dati** di un oggetto **`x`**, quando **`x`** viene distrutto.
- gli **oggetti con lo stesso tempo di vita**, tipicamente oggetti definiti nello stesso blocco oppure oggetti statici di una classe, vengono distrutti nell'ordine inverso a quello in cui sono stati creati

In particolare, il distruttore viene invocato:

1. sui **parametri** di una funzione **passati per valore** all'uscita dalla funzione
2. sulle **variabili locali** di una funzione all'uscita dalla funzione
3. **sull'oggetto anonimo ritornato per valore** da una funzione non appena esso sia stato usato (se qualche ottimizzazione prolunga il tempo di vita dell'oggetto anonimo ritornato significa che non viene invocato il distruttore)

Ordine di distruzione per g++ e clang: [2], [3], [1]

Comportamento del distruttore

Supponiamo che la lista ordinata dei campi dati di una classe **C** sia x1,…,xn. Quando viene distrutto un oggetto di tipo **C**, viene invocato automaticamente il distruttore della classe **C**, standard oppure ridefinito, con il seguente comportamento:

1. innanzitutto viene eseguito il corpo del distruttore della classe **C**, se questo esiste
2. vengono quindi richiamati i distruttori per i campi dati nell'ordine inverso alla loro lista di dichiarazione, cioè xn,…,x1. Per un campo dati di tipo non classe (cioè primitivo o derivato) viene semplicemente rilasciata la memoria (è il "distruttore standard" dei tipi non classe) mentre per i tipi classe viene invocato il distruttore, standard oppure ridefinito.

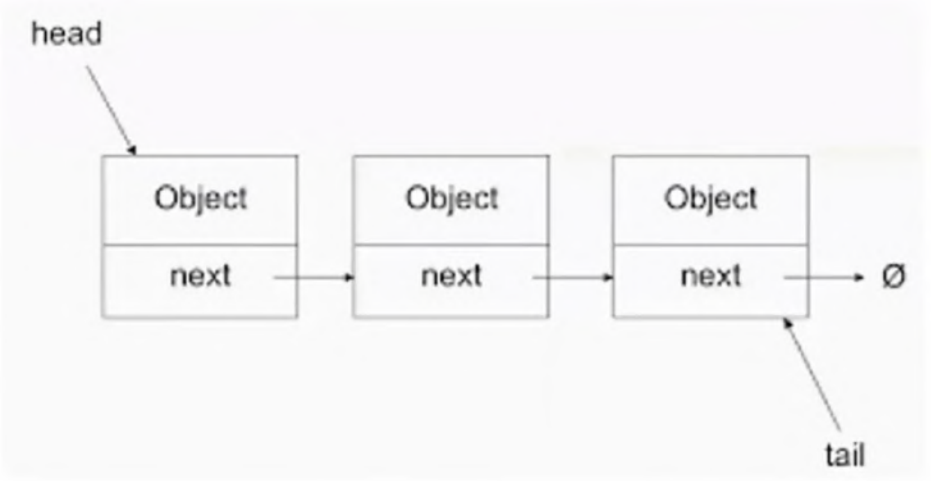
Distruttore standard: semplicemente ha il corpo vuoto. Quindi il distruttore standard di una classe **C** si limita a richiamare i distruttori per i campi dati di **C** in ordine inverso di dichiarazione

Il distruttore di **`bolletta`** richiama il metodo statico privato

```
bolletta::distruggi(nodo* p) {
  if (p) {
    distruggi(p->next);
    delete p;
  }
}
```

che a sua volta esegue l'operatore **`delete`** su tutti gli elementi della lista.


What's
Happening


head
Object
next
Object
next
Object
next
Ø
tail

[1] L'istruzione "`delete p;`" provoca l'invocazione del distruttore della classe `nodo`, e poichè esso non è stato ridefinito viene richiamato il distruttore standard di `nodo`.

[2] Il distruttore standard della classe `nodo` richiama a sua volta per ogni campo dati di `nodo`, cioè `info` e `next`, i rispettivi distruttori. Il campo puntatore `next` viene semplicemente deallocato, mentre per il campo `info` viene invocato il distruttore della classe `telefonata`, e siccome neanch'esso è stato ridefinito verrà richiamato il distruttore standard di `telefonata`.

[3] Il distruttore standard della classe `telefonata` dealloca l'intero `numero` e richiama il distruttore della classe `orario` per i campi `inizio` e `fine`, che sarà quindi il distruttore standard di `orario`.

[4] Il distruttore standard della classe `orario` dealloca il campo dati intero `sec`.

Modo più elegante: ridefinire il distruttore della classe `nodo` in modo che esso provveda a distruggere l'oggetto puntato da `next`

Modo più elegante: ridefinire il distruttore della classe **nodo** in modo che esso provveda a distruggere l'oggetto puntato da **next**

```
bolletta::nodo::~nodo()
// invocazione automatica distruttore di telefonata
// deallocazione automatica puntatore next
{
  if (next != 0)
    delete next; // chiamata ricorsiva
}
```

In questo modo il distruttore di **bolletta** diventa:

```
bolletta::~bolletta() { if(first) delete first; }
```

```
bolletta::nodo::~nodo()
// invocazione automatica distruttore di telefonata
// deallocazione automatica puntatore next
{
  if (next != 0)
    delete next; // chiamata ricorsiva
}
```

```
bolletta::~bolletta() { if(first) delete first; }
```

From the C++0x draft Standard.

> $5.3.5/2 - "[...]In either alternative, the value of the operand of delete may be a null pointer value. [..."

Of course, no one would ever do 'delete' of a pointer with NULL value, but it is safe to do. Ideally one should not have code that does deletion of a NULL pointer. But it is sometimes useful when deletion of pointers (e.g. in a container) happens in a loop. Since delete of a NULL pointer value is safe, one can really write the deletion logic without explicit checks for NULL operand to delete.

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

// funzione esterna, pass. per valore
C F(C p) { return p; }

C w("3"); // variabile globale

class D {
public:
  static C c; // campo dati statico
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

Esercizio: cosa stampa il seguente programma compilato con l'opzione `-fno-elide-constructors`?

Attenzione: senza l'opzione produce un output diverso!

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

// funzione esterna, pass. per valore
C F(C p) { return p; }

C w("3"); // variabile globale

class D {
public:
  static C c; // campo dati statico
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

Esercizio: cosa stampa il seguente programma compilato con l'opzione `-fno-elide-constructors`?

Attenzione: senza l'opzione produce un output diverso!

```
5~C 5~C UNO
5~C 5~C DUE
5~C 5~C 5~C 4~C 3~C
```

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

C  F(C& p) { return p; } // passaggio per riferimento

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

C  F(C& p) { return p; } // passaggio per riferimento

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

```
5~C UNO
5~C DUE
5~C 5~C 5~C 4~C 3~C
```

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

C& F(C& p) { return p; } //passaggio e ritorno per riferimento

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

```
UNO
DUE
5~C 5~C 5~C 4~C 3~C
```

Rule of three (C++ programming)

From Wikipedia, the free encyclopedia

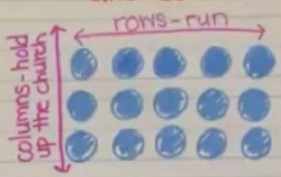
Rule of Three [edit]

The **rule of three** (also known as the Law of The Big Three or The Big Three) is a rule of thumb in C++ (prior to C++11) that claims that if a class defines one (or more) of the following it should probably explicitly define all three:[1]

- destructor
- copy constructor
- copy assignment operator

arrays

a set that shows equal groups in rows and columns



3 rows

5 columns

repeated addition: 5+5+5=15

or: 3+3+3+3+3=15

multiplication equation: 5×3=15

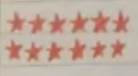
or: 3×5=15

4+4=8 or 2+2+2+2=8

4×2=8 or 2×4=8

2+2+2=6 or 3+3=6

2×3=6 or 3×2=6



6+6=12 or 2+2+2+2+2+2=12

6×2=12 or 2×6=12

3+3+3=9 or 3+3+3=9

3×3=9 or 3×3=9

5+5+5+5=20 or 4+4+4+4+4=20

5×4=20 or 4×5=20

```
int arrayStatico[5] = {3,2,-3};
int* arrayDinamico = new int[5];
arrayDinamico[0]=3;
*(arrayDinamico+1) = 2;
delete[] arrayDinamico;
```

```
int arrayStatico[5] = {3,2,-3};
int* arrayDinamico = new int[5];
arrayDinamico[0]=3;
*(arrayDinamico+1) = 2;
delete[] arrayDinamico;
```

Array di oggetti

- Array statici e dinamici
- Costruzione di array
- Distruzione di array


```
class C {
public:
  int i;
  C(int x=3): i(x) {}
  ~C() {cout << i << "~C ";}
};

int main() {
  C a[4] = {C(1), C(), C(8)};
  // distruzione oggetti anonimi, stampa: 8~C 3~C 1~C
  cout << a[0].i << a[1].i << a[2].i << a[3].i << endl;
  // stampa: 1383
}
// all'uscita stampa: 3~C 8~C 3~C 1~C
```