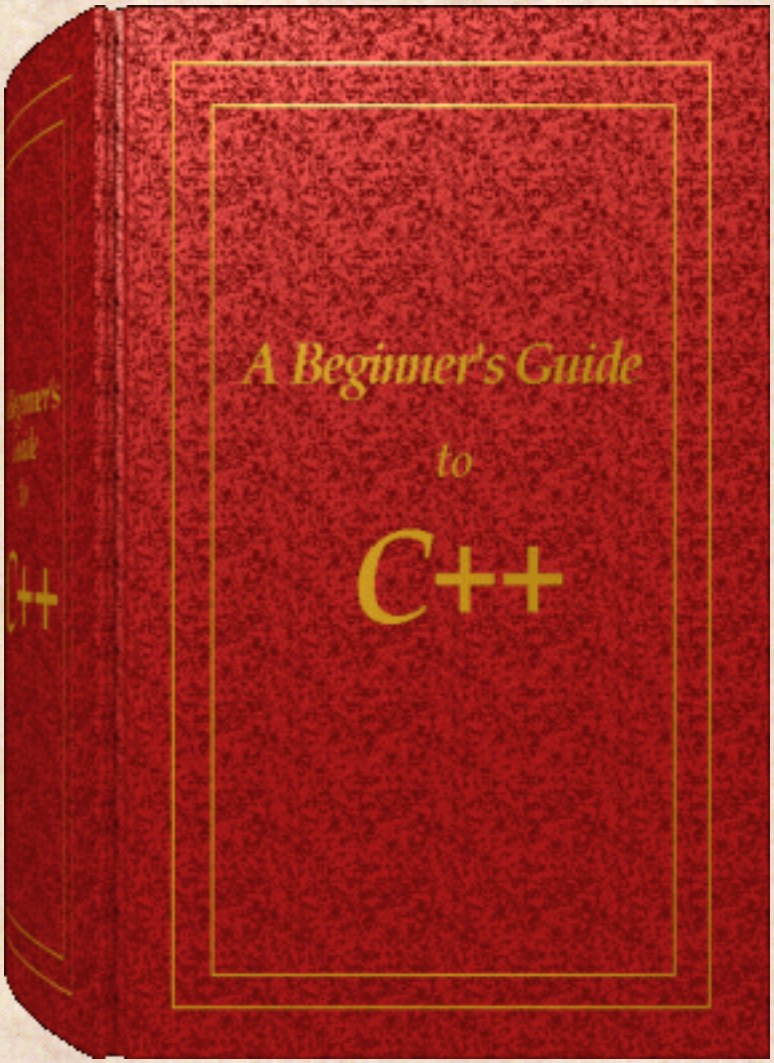

A Beginner's Guide to C++

C++ è un linguaggio orientato agli oggetti basato sul C. Questo tutorial descrive l'evoluzione del C in C++. Quindi, il lettore dovrebbe avere familiarità con i fondamenti del C. Aver già esperienza nella costruzione di software orientato agli oggetti non è richiesto, ma è comunque utile.
Il programmatore C++ già esperto potrà invece usare queste pagine come una guida rapida.

Clicca sul libro...

traduzione italiana di Antonio Bonifati
la versione originale in inglese è qui

CYBERSOFT.ORG

---

traduzione italiana di
Antonio Bonifati

Beginner's guide to C++

Introduzione

C++ è un linguaggio orientato agli oggetti basato sul C. Allo scopo di imparare questa "cosa", penso che il lettore dovrebbe prima essere familiare con la programmazione procedurale, per riuscire a vedere la grande differenza, e inoltre con il C. L'obiettivo di questa pagina non è di spiegare il C, ma di spiegare gradualmente le differenze tra C e C++ che rendono il C++ un linguaggio così potente.

Requisiti Software

Da un punto di vista tecnico, qualsiasi cosa che riesci a compilare con un compilatore C **dovrebbe** funzionare anche con un compilatore C++. Devi solo attivare le giuste opzioni e ogni cosa sarà a posto nel codice: occorre passare al modo C++ nei compilatori Borland C++ (di solito l'estensione dei file in C++ è CPP), o usare g++ oppure c++ al posto di gcc o cc su un sistema basato su Unix, ...

C++ viene usato su molti computer. Io personalmente lavoro su un PC Pentium sotto Linux. Ho gcc/g++ 2.7.2, che è quasi uno standard adesso, e ognuno degli esempi che darò si compilerà e (spero) funzionerà sicuramente con gcc. Penso comunque che sarà facile far funzionare ciascun esempio anche su altri sistemi. Comunque, se hai un problema e pensi ci sia un bug nel mio codice, non esitare, scrivimi !!!
[click qui per scrivere in inglese all'autore originale]

Ndt: esiste anche una "port" di gcc per i sistemi DOS/Windows. Si chiama DJGPP e si può scaricare gratuitamente sul sito di DJ Delorie (usate lo zip-picker).

Supporto Tecnico

Spero che questi discorsi sul C++ siano abbastanza utili per iniziare. Comunque, ricorda che queste pagine non coprono tutto quello che c'è da dire sul C++. Quindi ti chiedo di mandarmi commenti, suggerimenti o correzioni.
[scrivi all'autore originario in inglese...]

Ringraziamenti speciali a

Gherardo De Riu: primo lettore italiano.

PiLP: primo lettore inglese. Ha corretto molti errori.

Jean-Philippe Cottin: per i molti esempi che ha fornito.

Download

cxxguide.tar.gz [132K]

Lo zip contiene *tutto* il tutorial per visione offline (*identico* a quello che c'è online). In più i sorgenti C++ di una classe per usare i numeri complessi come i double, basata su tutto quello che è stato spiegato nel tutorial.

Beginner's guide to C++

Introduzione

C++ è un linguaggio orientato agli oggetti basato sul C. Allo scopo di imparare questa "cosa", penso che il lettore dovrebbe prima essere familiare con la programmazione procedurale, per riuscire a vedere la grande differenza, e inoltre con il C. L'obiettivo di questa pagina non è di spiegare il C, ma di spiegare gradualmente le differenze tra C e C++ che rendono il C++ un linguaggio così potente.

Requisiti Software

Da un punto di vista tecnico, qualsiasi cosa che riesci a compilare con un compilatore C **dovrebbe** funzionare anche con un compilatore C++. Devi solo attivare le giuste opzioni e ogni cosa sarà a posto nel codice: occorre passare al modo C++ nei compilatori Borland C++ (di solito l'estensione dei file in C++ è CPP), o usare g++ oppure c++ al posto di gcc o cc su un sistema basato su Unix, ...

C++ viene usato su molti computer. Io personalmente lavoro su un PC Pentium sotto Linux. Ho gcc/g++ 2.7.2, che è quasi uno standard adesso, e ognuno degli esempi che darò si compilerà e (spero) funzionerà sicuramente con gcc. Penso comunque che sarà facile far funzionare ciascun esempio anche su altri sistemi. Comunque, se hai un problema e pensi ci sia un bug nel mio codice, non esitare, scrivimi !!! [click qui per scrivere in inglese all'autore originale]

Ndt: esiste anche una "port" di gcc per i sistemi DOS/Windows. Si chiama DJGPP e si può scaricare gratuitamente sul sito di DJ Delorie (usate lo zip-picker).

Supporto Tecnico

Spero che questi discorsi sul C++ siano abbastanza utili per iniziare. Comunque, ricorda che queste pagine non coprono tutto quello che c'è da dire sul C++. Quindi ti chiedo di mandarmi commenti, suggerimenti o correzioni.
[scrivi all'autore originario in inglese...]

Ringraziamenti speciali a

Gherardo De Riu: primo lettore italiano.

PiLP: primo lettore inglese. Ha corretto molti errori.

Jean-Philippe Cottin: per i molti esempi che ha fornito.


C++

Beginner's guide to C++

Parti principali

Ci saranno 3 parti principali in questa guida. Mi raccomando, leggile in questo ordine a meno che non stai cercando qualcosa di particolare. Eccole:

- Differenze tra C e C++: Quelle piccole differenze che hanno poco o niente a che fare con la programmazione orientata agli oggetti, ma risolvono molti problemi del C standard.
- Programmazione ad Oggetti: La parte più succosa del nostro discorso. Questa parte include argomenti come classi, oggetti e questioni connesse. Dopo averla letta, non sarai più lo stesso programmatore...
- Argomenti complessi: Template, stream, eccezioni... Impara questi per diventare un C++ super-user.

Differenze tra il C e il C++

Semplicemente aumentano l'incredibile piacere che provi ogni volta che devi scrivere codice. Diamoci un'occhiata...

- Commenti: Adesso puoi inserire dei commenti tra // e la fine della linea.
- Usare le librerie C: In C++ puoi anche chiamare una funzione della libreria standard del C o di altre. È molto facile linkare insieme codice C e C++.
- Definire le funzioni: una funzione è definita con il suo nome e tutti i tipi dei suoi parametri, il che permette di definire più funzioni sotto lo stesso nome.
- Estensioni nella definizione di costanti: const adesso è meglio di una #define.
- Il compilatore C++ non fa la conversione implicita da un puntatore a void ad un puntatore di un altro tipo.
- Funzioni Inline: Un modo elegante ed efficace per definire le macro.
- Le Variabili possono essere definite ovunque, non solo all'inizio, e cose tipo `for(int i=1;i<5;i++)` adesso sono permesse.
- Riferimenti (Reference): I parametri adesso possono essere passati per riferimento nelle chiamate di funzione in modo simile al Pascal.
- Namespace: Come usare più variabili e tipi con lo stesso nome.

Nuovo stile dei commenti del C++

Il C++ introduce un nuovo tipo di commenti. Iniziano con // e proseguono fino alla fine della linea. Sono molto convenienti per scrivere commenti di una sola linea.

Quando il compilatore C++ incontra l'inizio di un commento, ignora qualsiasi cosa fino alla corrispondente fine del commento, che è marcata con un */ se l'inizio del commento era marcato da /*, e dal finelinea stesso se l'inizio del commento era //.

Esempio: questo codice è corretto:

```
code    /* comment */
code    // comment
```

questo no:

```
code    /* comment
code    // comment */ comment
code
```

perchè l'ultimo "comment" dopo il */ è considerato come fosse codice. Questo perchè // si trova dentro il commento stile C, perciò è ignorato.

Il principale vantaggio che deriva dalla disponibilità di questo nuovo tipo di commento è quello di poter "commentare-due volte" alcune parti del codice:

```
     code
/* inizio della parte commentata
     code  // comment
     code  // comment
fine della parte commentata */
     code
```

Linkare codice C e C++

Usare funzioni C in moduli separati (standard o non) nel codice C++

Semplicemente usa le tue funzioni in C come se stessi programmando in C: includi il file header, e poi chiama la funzione. Ad es. in C++, questo programma si compilerà senza problemi:

```
#include <stdio.h>
#include <stdlib.h>

char buffer[1];

void foo (char *name)
{
  FILE *f;
  if ((f = fopen (name, "r")) == NULL) return;
  fread (buffer, 1, 1, f);
  fclose (f);
}
```

Dichiarare proprie funzioni C in moduli per poter essere usate col C++

Puoi scrivere alcuni moduli di una applicazione in C, compilarli, e poi linkarli con moduli C++. Per poter chiamare le tue funzioni C precompilate dal codice C++, devi dichiararle in un file header del tipo seguente, e poi usare lo stesso header per compilare il modulo C e il modulo utilizzatore in C++. Ecco come deve apparire il tuo file header del modulo C:

```
...
#ifdef __cplusplus
extern "C" {
#endif

/* definizioni C */

#ifdef __cplusplus
}
#endif
...
```

Quella frase extern "C" per l'appunto dice al compilatore che le dichiarazioni seguenti sono di tipo C.

Dopo aver indicato questo al compilatore, le definizioni C del modulo nei programmi in C++ saranno utilizzabili come se fossimo in C.

Ndt: mancare quelle direttive nel file header del C in gcc provocherà errori di riferimenti indefiniti nella fase di link. È buona norma, anche quando si scrivono librerie o programmi in C che si vuole possano essere utilizzabili anche in C++ includere quelle linee, che non hanno effetto sui programmi in C. Le `#ifdef __cplusplus` servono per evitare errori da parte dei compilatori che sono solo per il C e non conoscono la direttiva `extern "C"`

Definizione delle funzioni

Questa pagina contiene i seguenti argomenti:

La definizione di una funzione è cambiata in C++. Mentre nel C è obbligatorio solo il nome, in C++ devi specificare tutti i parametri e il tipo di ogni parametro per definire una funzione. Questo porta alle seguenti conseguenze:

Prototipi di funzione

Devi sempre definire un prototipo della funzione prima di usarla. Questo prototipo è definito dal tipo del valore ritornato (o void se è nessun valore), dal nome della funzione, e dalla lista dei tipi dei parametri.

Esempio: corretto:

```
void hello();              // questa funzione non ha alcun parametro
int about (int, char);
foo (int);                 // il tipo di ritorno di default è int
```

errato:

```
void hello();
...
void foo()
{
   hello (10);
}
```

quest'ultimo genererà l'errore: "chiamata alla funzione indeterminata (cioè non definita) `hello (int)`" oppure "troppi argomenti alla funzione `void hello()`" e simili.

Ndt: notare che in C non si aveva quell'errore. Per saperne di più vedere il Kernighan-Ritchie, paragrafo 1.7

Nota che il nome di un parametro nel prototipo non è obbligatorio, solo il suo tipo è richiesto. Se il tipo di ritorno viene omesso, viene utilizzato int come default.

Non c'è bisogno di usare un prototipo nel codice sorgente se la funzione è chiamata dopo la sua definizione. Esempio:

```
// void hello();          Linea non obbligatoria

void hello()
{
  printf ("hello world.\n");
}

void foo()
{
  hello();
}
```

Overloading delle funzioni

I cambiamenti nella definizione delle funzioni sono coerenti con una nuova funzionalità permessa dal C++ detta "sovraccaricamento" (overloading) di un nome di funzione. In C++ è lecito avere due (o più) funzioni con lo stesso nome, ammesso però che la lista dei loro parametri sia differente (cosa che permette al compilatore di distinguerle). Sovraccaricare un nome di funzione è facile: basta dichiarare e utilizzare le funzioni con lo stesso nome come al solito.

Esempio:

```
int foo (int a, int b)
{
  return a * b;
}

int foo (int a)
{
  return a * 2;
}
...
{
a = foo (5, 5) + foo (5);     // ritorna    5 * 5 (risultato della prima foo)
}                             //          + 5 * 2 (risultato della seconda foo)
```

Ma possono sorgere dei problemi. Ad es. puoi definire due funzioni in questo modo:

```
int add (int a, int b);
float add (float a, float b);
```

e poi fare una chiamata ad add. Questo tipo di definizioni vanno evitate. I compilatori si confondono e danno warning o errori a seconda dei casi, corrispondenti a perdite di informazioni o ambiguità della situazione.

Ndt: vi invito a sperimentare la situazione col vostro compilatore. Provate a definire entrambe le funzioni in modo che restituiscono a+b. Se si compila provate poi a chiamare add con due interi, due float, con un intero e un float e viceversa. Potete stabilire quale funzione add viene chiamata tracciando il programma o facendo stampare da ciascuna delle funzioni un messaggio indicativo con una printf.

Quella che abbiamo visto era una situazione di ambiguità pericolosa. Ma che ne dite di questa?

```
int add (int a, float b);
float add (float a, int b);
...
a = add (2, 2);
```

Fortunatamente, il compilatore rifiuterà sicuramente quel codice, spiegando che non può scegliere tra le due funzioni funzioni add. Perciò fai attenzione!

Valori di default dei parametri

Puoi specificare un valore di default per gli ultimi parametri di una funzione. Se un tale parametro viene omesso nella chiamata, il compilatore gli assegnerà appunto il suo valore di default.

Esempio:

```
int foo (int a, int b = 5);
...
int foo (int a, int b)
{
  return a * b;
}

{
  a = foo (2);        // Dà lo stesso risultato di
  a = foo (2, 5);
}
```

Nota che puoi assegnare dei valori di default solo ad un ultimo gruppo di parametri, altrimenti il compilatore non saprebbe a quale parametro assegnare il valore di default...

```
int func1 (int = 0, char);   // Non permesso
void func2 (char, int = 0);  // Ok
```

Un valore di default deve essere specificato nel prototipo, e non necessariamente ripetuto nella definizione della funzione. Infatti, quando specifici dei valori di default, in realtà è come se implicitamente stessi definendo molte funzioni con lo stesso nome. In un esempio precedente, quando abbiamo scritto:

```
int foo (int a, int b = 5);
```

`foo (int)` e `foo (int, int)` sono state prototipate e quindi implicitamente definite.

Pertanto notiamo che combinando valori di default dei parametri con l'overloading delle funzioni, possiamo produrre situazioni ambigue vietate, come questa:

```
int foo (int a, int b = 5);    //   Definizione con valore di default per b
int foo (float a);             //   Altra definizione di foo: produce ambiguità
int foo (int a);               // Errore: conflitto con la prima definizione
```

Definizioni di Constanti

Per dichiarare una variabile costante (cioè a sola lettura), semplicemente aggiungi la parola "const" prima della sua definizione.

```
const int a = 10;
```

Quando dichiari una variabile const con lo stesso nome in più file in C++, puoi dare a questa variabile un valore differente per ciascun file. La cosa non era permessa dall' ANSI C, dovevi dichiarare esplicitamente ogni costante come static.

Un'altra differenza col C è che il compilatore C++ calcola i valori costanti a tempo di compilazione. Questo significa che cose di questo tipo adesso sono permesse:

```
const int number = 10;   // Dichiarazione di costante
int array1[number];      // Normale codice C++ (non permesso dall' ANSI-C)
int array2[number * 2];  // Non permesso dall' ANSI C.
```

Ndt: le variabili costanti sono utili per due motivi: possono consentire una certa ottimizzazione ed evitano eventuali errori logici che si hanno quando normali variabili da considerare come costanti vengono erroneamente modificate. Infatti un'altra differenza fondamentale con il C è che mentre in C un tentativo di modificare un oggetto const produce tuttalpiù un warning, in C++ si ha sempre un errore di compilazione vero e proprio. In C le variabili costanti sono semplicemente delle variabili: a parte l'emissione del warning non hanno alcun trattamento speciale a compile-time. L'effetto del qualificatore const è solo quello di far emettere un warning in caso di modifica. In C++ invece esse sono valutate come costanti vere e proprie a compile time.

Puntatori a Void

Nel modo C++, non puoi convertire implicitamente un puntatore a void (`void *`) in un puntatore di un altro tipo. Devi esplicitare l'operazione usando un operatore di cast.

Esempio:

```
int *p_int;
void *p_void;

p_void = p_int;           // Ok alla conversione implicita

p_int = p_void;           // Conversione implicita non permessa in C++ (in C sì)

p_int = (int *)p_void;    // Conversione esplicita -> OK
```

Questa regola ha un certo senso: un puntatore a `void` è fatto per puntare a qualsiasi cosa, perciò puoi metterci qualsiasi valore (indirizzo) senza troppe storie. Ma un puntatore ad `int` per esempio è usato di solito solo puntare ad un valore `int`, perciò non puoi direttamente farlo puntare a qualsiasi altro valore (che sarebbe visto come int, col risultato che l'intero ottenuto dipende dall'implementazione - la cosa è di rara utilità).

Funzioni Inline

In C, non era facile definire macro-funzioni. L'uso della direttiva #define non è semplice. C++ fornisce un modo migliore di creare delle funzioni macro.

Se dichiari una funzione "inline", allora il codice sarà riprodotto nel punto di chiamata ognivolta che chiami la funzione. Attenzione che è necessario definire il codice della funzione prima della sua prima chiamata, altrimenti un compilatore potrebbe non essere in grado di espanderlo.

Ndt: Infatti il codice delle funzioni inline deve essere espanso, non deve essere generata solo una chiamata (questa è la differenza!)

Questo è il motivo per cui il codice delle funzioni inline è generalmente messo nel file header che dichiara le funzioni inline stesse.

```
inline int addition (int a, int b)
{
  return a + b;
}
```

Ma puoi anche definire la funzione in due passi:

```
inline int addition (int a, int b);   // qui la keyword inline è opzionale
... // qui però addition non deve ancora essere usata
inline int addition (int a, int b)
{
  return a + b;
}
```

Usare le funzioni inline probabilmente sarà più efficiente, perchè la maggior parte dei compilatori C++ prima sostituirà il codice, e poi lo ottimizzerà. Ma la dimensione del codice può aumentare di parecchio. Perciò è raccomandabile usare `inline` solo per piccole funzioni (costruttori e operatori di classe sono degli ottimi esempi che vedremo).

Ecco un esempio di codice per quelli che non vedono la differenza tra le funzioni inline e una `#define`:

```
#define SQUARE(x) ((x) * (x))
int a = 2;
int b = SQUARE (a++);
   // alla fine: a = 4 e b = 2 * 3 = 6. Sorpreso ?!

inline int square (int x)
{
  return x * x;
}

int c = 2;
int d = square (c++);
   // alla fine: c = 3 e d = 2 * 2 = 4. Risultati corretti come ci aspettavamo !!
```

Nota: con il compilatore C della GNU (GCC) succede una cosa curiosa quando si chiama la macro SQUARE(a++): il risultato in b è corretto (virtù di GCC che farà felice tutti i programmatori C), però il valore finale di a è ancora 4 e non 3. Tuttavia è possibile risolvere una volta per tutte anche questo problema usando un paio di estensioni del GNU C, che sono i Naming Types e le Statement Exprs. Ecco la soluzione:

```
/* questo è GNU-specific :) */
#define SQUARE(x) \
  ({typedef _tx = (x); \
    _tx _x=(x); _x*_x; })
```

In questo modo x viene valutato solo una volta. Per saperne di più sulle estensioni C e C++ di GCC digitate al prompt:

```
info gcc "C Extensions"
```

oppure

```
info gcc "C++ Extensions"
```

Dichiarazioni di Variabili

La dichiarazione di una variabile adesso è considerata come una istruzione. Questo significa che puoi crearne una in qualsiasi parte del tuo codice.

Esempio:

```
void foo()
{
  int i = 1;
  printf ("%d\n", i);             // modalità C
  int j = 1;                      // Non permesso in C
  printf ("%d\n", j);
  for (int k = 1; k < 10; k++)    // Davvero non permesso in C
    printf ("%d\n", k);
}
```

Appena dichiari una variabile, essa è visibile nel blocco di codice in cui è stata dichiarata (un blocco di codice inizia con una "{" e finisce con una "}"). Ma dopo la fine del blocco, o prima della dichiarazione, la variabile non sarà utilizzabile. Per chiarire vediamo degli esempi sbagliati:

```
{
  // Dopo questa linea (quella del for), i sarà visibile (per convenzione)
  // solo nel blocco di codice seguente. (Nota: nelle prime versioni dello
  // standard del C++, era visibile anche per tutto il resto della funzione.
  // Ma adesso questa regola non è più valida, e le cose sono un po' più logiche).
  for (int i = 0; i < 5; i++) // Dichiarazione di i per l'intero ciclo
  {                           // Inizio del blocco dentro cui i è "viva" e rimane
dichiarata
    int j;
  }                           // Dopo questo }, i e j sono sconosciute

  // Poichè i è morta, non è più dichiarata, perciò questo è permesso:
  // in altri termini occorre dichiarare una nuova variabile i.
  // scrivendo for (int i = 0; i < 5; i++)
  for (i = 0; i < 5; i++)
  {
  }

  j = 2;                      // errore: j non dichiarata
  do
  {
    FILE *f = fopen ("my_file", "r");
  }
  while (f == NULL);          // f sconosciuta, ERRORE

}
```

Riferimenti

Adesso puoi trasmettere per riferimento i parametri ad una funzione. Significa che quando modifichi il parametro all'interno della funzione, anche la variabile trasmessa sarà modificata. Quello che avviene in realtà è sempre un passaggio di puntatori, ma con i reference non è più necessario referenziare il puntatore passato per ottenerne il valore: il referenziamento è "automatico". Per specificare che un parametro sarà referenziato implicitamente, occorre solo mettere un & tra il tipo e il nome del parametro:

```
void A (int number)      // passaggio per valore. Dentro la funzione,
{                        // number è un'altra variabile
  number *= 2;
}
void B (int & number)    // per riferimento. La variabile stessa sarà modificata
{
  number *= 2;
}
int a = 5;
A (a);              // a non modificata
B (a);              // a modificata
```

Possiamo confrontare il nuovo metodo con il passaggio di puntatori (usato in C). Ma è più potente. Il compilatore lo ottimizzerà meglio (forse). Il codice è molto più facile da scrivere, basta solo aggiungere un & per passare per riferimento invece che per valore.

Mentre se vuoi passare puntatori di puntatori, non puoi usare riferimenti di riferimenti, che non ha senso.

I riferimenti sono sempre usati nei costruttori di copia per ottenere un riferimento all'oggetto base (la sorgente della copia).

i Namespace

Questo concetto è la risposta al comune problema delle definizioni multiple. Immagina di avere due file header che descrivono due tipi differenti, ma che hanno lo stesso nome. Come facciamo ad utilizzarli entrambi ?

La risposta in C++ sta nei namespace. Si definiscono tipi e funzioni in un namespace e dopo li si usano. I namespace sono definiti usando la parola riservata `namespace`, seguita da un identificatore. Ma vediamo subito un esempio...

```
namespace Geometry
{
  struct Vector
  {
    float x, y;
  };

  float Norm (Vector v);
}
```

In questo esempio, quello che c'è dentro le `{}` è normale codice C++. Per usare questo codice, devi specificare al compilatore che adesso stai per usare il namespace `geometry`. Ci sono due modi di fare questo: specificando per una singola variabile il suo namespace (la sintassi è `namespace::variabile`) o usando la sequenza di parole riservate `using namespace`.

```
// Esempio con ::
void Foo1()
{
  Geometry::Vector v = {1.0, 2.0};
  float f = Geometry::Norm (v);
}

// Con l'uso delle parole chiave "using namespace"
using namespace Geometry;
void Foo2()
{
  Vector v = {1.0, 2.0};
  float f = Norm (v);
}
```

Quando usi la sequenza `using namespace`, potresti avere dei problemi se due o più namespace definiscono lo stesso tipo. Se c'è del codice ambiguo, il compilatore non lo compilerà. Per esempio...

```
namespace Geometry
{
```

```
  struct Vector
  {
    float x, y;
  };
}

namespace Transport
{
  struct Vector
  {
    float speed;
  };
}

using namespace Geometry;
using namespace Transport;

void foo()
{
  Vector v0;              // Errore: ambiguità
  Geometry::Vector v1;    // OK
  Transport::Vector v2;   // OK
}
```

Un ultima cosa (ovvia ?): non puoi usare un namespace se non è ancora stato definito...

Programmazione ad Oggetti in C++

La programmazione ad oggetti non è meramente qualcosa che aggiungi ad un programma. È un altro modo di pensare un'applicazione. Per introdurla in modo graduale, qui c'è una breve descrizione.

Classi e loro componenti

Qui troverai tutto quello che hai bisogno di sapere per dichiarare classi ed ogni tipo di membri che puoi usare all'interno delle classi.

- Introduzione alle dichiarazioni di classe: Come dichiarare una classe di un oggetto e come instanziarla.
- Costruttori e distruttori di un oggetto. Particolari funzioni chiamate all'inizio e alla fine della vita di un oggetto.
- Classi e funzioni Friend: Un modo per accedere ai membri privati di un oggetto.
- Ridefinizione degli operatori: per poter usare gli operatori standard con i propri oggetti.
- Membri Statici: Membri che esistono anche senza alcuna istanziazione.
- Funzioni Const: Funzioni membro che hanno il permesso di trattare anche oggetti costanti.

Ereditarietà (Inheritance)

L'ereditarietà rappresenta un grande passo avanti nell'obiettivo della riduzione del tempo necessario allo sviluppo del software. È descritta qui.

- Ereditarietà di Classe: Classi Riusabili.
- Funzioni Virtual: Per avere ancora più vantaggi con l'ereditarietà.
- Ereditarietà Multipla: Come riusare le proprietà di più oggetti.

Come dichiarare una classe

Questa pagina tratta di:

Una classe è davvero una componente software. Quindi un buon modo di implementarla è di mettere la dichiarazione della classe in un file header e il codice isolato in un file C++.

Dichiarazioni di Classi

La dichiarazione di una classe è davvero semplice. La parola chiave `class` deve essere seguita dal nome delle classe e dalla dichiarazione di variabili e metodi interni alla classe, come mostrato nell'esempio:

```
class Complex
{
  float r, i;
public:
  float GetRe();
  float GetIm();
  float Modulus();
};
```

Puoi notare che:

- Le variabili sono dichiarate allo stesso modo in cui si dichiarano nelle classiche struct del C.
- Vengono dichiarate anche le funzioni, come se fossero prototipi.
- Non dimenticare il ";" finale: al compilatore non piacerà!

`public` è una parola chiave che indica che le funzioni GetRe, GetIm e Modulus possono essere chiamate anche al di fuori dell'oggetto. Non c'è alcuna parola chiave prima di r e i, perciò essi sono dichiarati privati per default, e non sono accessibili dall'esterno.

Infatti, ci sono tre tipi di funzioni membro:

- private (private): sono accessibili solo all'interno della classe.
- protected (protette): accessibili dentro la classe e dalle sue classi eredi.
- public (pubbliche): accessibili ovunque nel codice.

Quando dichiari una classe con la parola riservata `class`, ogni membro è "private" per default. Puoi anche dichiarare una classe con la keyword `struct`; in tal caso ogni membro sarà "public" per default. Questa è l'unica differenza tra `struct` e `class`.

```
class Complex
{
  float r, i;       // Private per default
};
struct Complex
{
```

```
  float r, i;       // Pubbliche per default
};
```

Implementazione delle funzioni membro

Dopo la dichiarazione della classe, devi implementare le funzioni membro. Basta dire al compilatore di quale classe le funzioni fanno parte, mettendo il nome della classe seguito da :: prima del nome della funzione.

Col nostro esempio precedente:

```
float Complex::Modulus()
{
  return sqrt(r*r + i*i);           // Notare l'uso di r e i
}
```

In queste funzioni membro, puoi usare ogni metodo o variabile dell'oggetto come se fosse dichiarato globalmente. L'uso fatto sopra di r ed i dà l'idea.

Instanziazione di una classe come una variabile

Una classe può essere usata come ogni altro tipo di variabile. Per dichiarare un oggetto appartenente alla classe (ndt: o possiamo dire "al tipo della classe"), basta mettere il nome della classe seguito dal nome dell'oggetto che vogliamo creare.

Per esempio, per usare la classe complex possiamo scrivere:

```
Complex x, y;         // Dichiarazione
Complex vector[10];   // Array di Complex (di numeri complessi)
...
x = y;                // Utilizzo di oggetti dichiarati
```

Allocazione Dinamica

I puntatori ad oggetti si dichiarano esattamente come in C, con un "*". Si può anche ottenere l'indirizzo di un oggetto con l'operatore "&".

```
Complex x;          // Dichiarazione di un oggetto Complex
Complex *y;         // Dichiarazione di un puntatore ad (un oggetto) Complex
y = &x;             // y punta ad x
```

La chiamata C++ corrispondente alla `malloc()` del C è l'operatore `new`. Esso alloca spazio in memoria per un oggetto e ritorna un puntatore a questa area appena allocata. Per disallocare l'oggetto e liberare la memoria associata, usa l'operatore `delete`. Dopo una chiamata a delete lo spazio di memoria è stato restituito allo heap e **non** va più utilizzato direttamente tramite il puntatore.

```
Complex *p, *q;      // Dichiarazione di due puntatori
p = new Complex;     // Allocazione dinamica di un singolo Complex
q = new Complex[10]; // Allocazione dinamica di un array di 10 Complex
...                  // Codice che usa p e q
delete p;            // Libera memoria per un singolo oggetto
delete [] q;         // Libera la memoria di un array
```

Notare l'uso di [] per dire al compilatore che deve essere eliminato un array dinamico. Se non specifichi questo, nella maggior parte dei casi (dipende dal compilatore) sarà liberato solo il primo oggetto dell'array, non l'intera memoria associata. Inoltre anche l'effetto di usare `delete []` con un singolo oggetto è indefinito.

Puoi usare gli operatori `new` e `delete` ovunque nel codice, proprio come `malloc`. Non dimenticare di liberare la memoria associata ad un oggetto quando esso non serve più. Vedi la sezione construttori e distruttori per maggiori informazioni.

Usare i membri di un oggetto

Una volta che hai dichiarato un nuovo oggetto, puoi accedere ai suoi membri pubblici come faresti con una struttura in C. Basta usare il separatore . con un oggetto dichiarato staticamente, e -> con uno dichiarato dinamicamente.

```
Complex x;                    // Oggetto Statico
Complex *y = new Complex;     // Oggetto Dinamico
float a;
a = x.Modulus();               // Chiamata della funzione membro Modulus dell'oggetto
Complex
a = y->Modulus();
delete y;

Class foo
{
  int a;            // membro privato
  int Calc();       // funzione privata
public:
  int b;
  int GetA();
};
```

Auto-riferimento ad un oggetto: `this`

Finora, siamo stati in grado di usare qualsiasi membro di una classe all'interno di essa, considerandoli come variabili dichiarate. Ma potresti voler ottenere l'indirizzo di un oggetto all'interno di una sua funzione membro, per esempio per inserire un oggetto in una lista concatenata.

La soluzione è la parola riservata `this`. `this` è un puntatore del tipo (`object *const`), e punta all'oggetto corrente. È una variabile membro implicita, dichiarata come `private`. Esempio:

```
class A
{
  int a, b;
public:
  void Display();
};

void A::Display()
{
  printf ("Object at [%p]: %d, %d.\n", this, a, b);
}
```

Notate che il tipo di this è "puntatore costante ad un oggetto object". Questo significa che il valore del puntatore this non può essere alterato, mentre l'oggetto puntato ovviamente sì.

Funzioni e Classi Friend

Funzioni "Amiche" (Friend)

Come abbiamo visto non possiamo accedere a membri privati di una classe dall'esterno della classe. Ma a volte abbiamo bisogno di farlo. Per esempio, una funzione di callback ha un header prestabilito, e non può essere il membro di nessuna classe. In questi casi, puoi definire la funzione come "friend" della classe. Così dall'interno della funzione sarai in grado di accedere a qualsiasi membro privato senza ottenere errori. Vediamo un esempio:

```
class A
{
  int a;
public:
  A (int aa) : a (aa) {}     // Costruttore che inizializza a
  friend void foo();
};

void foo()
{
  A a_obj (5);       // Normale chiamata
  a_obj.a = 10;      // Non sarebbe permesso in una funzione non-friend
}
```

Puoi mettere le dichiarazioni friend da qualsiasi parte nella definizione della classe, perchè una ad una funzione friend non si applicano gli attributi di pubblico o privato. Notare che è la classe che sceglie quali funzioni avranno il permesso di accedere direttamente ai suoi membri privati. Dopo questa dichiarazione nella classe, non è necessario alcun altro prototipo quando definiamo la funzione.

Le funzioni Friend sono spesso usate ridefinire gli operatori.

Una funzione friend non può accedere al puntatore `this`, perchè non viene invocata su un oggetto.

Puoi usare anche una funzione membro di un'altra classe come funzione friend di una classe.

```
class B;     // dichiarazione necessaria per evitare problemi di riferimento
circolare
class A
{
  void uses_class_B (B &);             // Normale funzione membro
};
class B
{
  friend void A::uses_class_B (B &);   // dichiarazione di una funzione friend
};
```

In questo esempio la funzione `uses_class_B` della classe A, che accetta per parametro un riferimento ad un oggetto della classe B, è friend della classe B.

Classi Friend (o "Amiche")

Come abbiamo fatto per le funzioni, possiamo anche dichiarare delle classe friend. Questo è piuttosto semplice da fare:

```
class A
{
```

```
  int a;

  friend class B;
};
class B
{
public:
  void foo();
};
B::foo()
{
  A a_obj;
  a_obj.a = 10;          // Non sarebbe permesso se la classe B non fosse friend di A
}
```

Nota bene che è la classe A che sceglie quali classi hanno il diritto di accedere ai suoi membri privati e non quest'ultime.

Una conseguenza di questo fatto è che classi amiche di classi amiche di una classe **non** sono classi amiche anche di quest'ultima. Vediamo un esempio per chiarire il gioco di parole:

```
class A
{
  int a;
  friend class B;
};
class B
{
  int b;
  friend class C;
};
class C
{
  int c;
  int foo()
  {
    B b_obj;
    b_obj.b = 10;         // OK
    A a_obj;
    a_obj.a = 10;         // Rifiutata dal compilatore. Dovrebbe esserci
  }                       // una dichiarazione di C come class friend in A.
};
```

In altre parole la relazione X è friend di Y non è una relazione transitiva.

C è friend di B, B è friend di A **non** implica che C sia friend di A

Ridefinire gli operatori

Contenuti:

In C++, gli operatori (come quello di addizione o di moltiplicazione) sono considerati come delle vere e proprie funzioni. Essi sono già definiti per i tipi di parametri standard (i tipi "nativi"), e possiamo farne l'overloading, cioè possiamo sovraccaricarne il nome di significati, come facciamo per le normali funzioni.

Se ancora non conosci cos'è l'overloading di funzione fai un salto qui.

Riflettete che proprio questa è la conseguenza logica interessante del fatto che gli operatori non sono nient'altro che delle funzioni, che del resto è una banale particolarizzazione già nota a tutti i matematici.
Ripeto: se gli operatori sono funzioni, allora possiamo fare l'overloading degli operatori per dargli nuovi ulteriori definizioni che si adattino alla nostre proprie classi, proprio come facciamo l'overloading di funzioni.

Ma non dobbiamo dimenticarci che gli operatori hanno tutti un significato "naturale", e, anche se ovviamente possibile, non è una buona idea ad esempio definire l'operatore di addizione per moltiplicare vettori!

Funzioni friend oppure funzioni membro?

Si dichiara un operatore usando la keyword `operator` seguita dall' operatore stesso. Quando sovraccarichiamo un operatore, possiamo scegliere tra due strategie: possiamo dichiarare gli operatori come funzioni friend oppure come funzioni membro. Non c'è una soluzione migliore in assoluto, dipende dall'operatore.

dichiarare gli operatori come Funzioni Friend

Quando dichiari un operatore come una funzione friend, devi mettere nel prototipo della funzione i parametri che saranno utilizzati dall'operatore (quelli prima e dopo il segno + per esempio), e un valore di ritorno, cioè il risultato restituito dall'operatore.

```
class Complex
{
  float re, im;
public:
  Complex (float r = 0, float i = 0) { re = r; im = i; }
  friend Complex operator+ (Complex &, Complex &);
};

Complex operator+ (Complex &a, Complex &b)
{
  return Complex (a.re + b.re, a.im + b.im);
}
```

dichiarare gli operatori come Funzioni Membro

Per una funzione membro, il primo parametro è sempre la classe stessa (che è, come sappiamo, mai dichiarato e implicitamente trasmesso), e poi vanno indicati gli altri parametri.

```
class Complex
{
  float re, im;
public:
  Complex (float r = 0, float i = 0) { re = r; im = i; }
  Complex operator+ (Complex &);
};

Complex Complex::operator+ (Complex &a)
{
  return Complex (re + a.re, im + a.im);
}
```

In entrambi i casi, gli operatori possono poi essere utilizzati nella stessa maniera con cui si usano con i tipi standard:

```
{
  Complex a (1, 2);
  Complex b (3, 4);
  Complex c;
  ...
  c = a + b;           // Chiama la funzione di addizione
}
```

Lista degli operatori

Quando fate l'overloading di un operatore, dovete indicare lo stesso numero di argomenti dell'operatore "originale" se usate una funzione friend, e questo numero meno uno per una funzione membro. Segue una lista di operatori, presentati in ordine di priorità discendente (una versione per i browser che non supportano le tabelle è qui):

Operatore	Numero di parametri (nel conto va incluso l'oggetto nel caso di funzioni membro)	Nota
()	indeterminato	Devi usare una funzione membro.
[]	2	Devi usare una funzione membro.
->	2	Devi usare una funzione membro. Fanne l'Overload solo se ne hai davvero bisogno.
new delete	1	Per una classe, definiscili come funzioni membro (in tal caso sono automaticamente definite static). Possono anche essere definiti globalmente.
++ --	1	Pre-incremento (o pre-decremento).
++ --	2 (2º inutilizzato)	Post-in/decremento (il 2º argomento è di tipo int e non viene usato, serve solo per distinguere le forme postfisse e prefisse).
&	1	Pre-definito (ritorna "this"). Fai l'Overload solo se c'è davvero bisogno.
+ - ! ~ *	1	
(cast)	1	Vedi la descrizione seguente.

* / % + -	2	
<< >>	2	
< <= > == !=	2	
& ^ || && |	2	
=	2	Pre-definito (fa la copia dei membri). Devi usare una funzione membro.
+= -= *= /= %=	2	
&= ^= |= <<= >>=	2	
,	2	

Gli operatori () e []

L'operatore () è speciale perchè il numero dei suoi argomenti può essere variabile: possiamo definire molti operatori (), ciascuno con parametri diversi.

```
class Matrix
{
  int array[10][10];
public:
  // ritorna un elemento della matrice (per riferimento, in modo da essere in grado
  // di modificarlo; ad esempio posso scrivere Matrix matrix; ... matrix(2, 3) = 5;)
  int &operator()(int x, int y) { return array[x][y]; }

  // ritorna la somma degli elementi di una colonna
  // Ndt: si fa la convenzione che il primo indice sia quello di colonna e quindi il
secondo è quello di riga
  int operator()(int x)
  { int s=0; for (int i=0; i<10; i++) s+=array[x][i]; return s; }
};

{
  Matrix matrix;
  ...
  int a = matrix (2, 2);  // elemento (2,2)
  int b = matrix (2);     // somma della seconda colonna
}
```

Quando usi l'operatore [], puoi ritornare un riferimento all'oggetto. Così sarai in grado non solo di leggere l'elemento, ma anche di scriverlo:

```
class Vector
{
  int array[10];
public:
  int & operator[](int x) { return array[x]; }
};
{
  Vector v;
  int a = v[2];   // ottiene il valore
  v[5] = a;       // imposta il valore
}
```

Conversioni con l'operatore =

L'operatore = deve essere definito per forza come una funzione membro. Quando i suoi argomenti sono di un tipo classe diverso o di un tipo predefinito, può essere usato per fare delle conversioni di oggetti da una classe all'altra o da un tipo nativo ad una classe, ma solo durante l'assegnamento.

```
class Complex
{
  float re, im;
public:
  Complex (float r = 0, float i = 0) { re = r; im = i; }
  Complex & operator= (float f) { re = f; im = 0; return *this; }
};
{
  Complex a (10, 12);
  a = 12;               // Chiamata dell'operatore=
}
```

Infatti l'operatore = come convertitore di tipo ha queste limitazioni: non può essere usato ovunque, ad esempio non può essere usato per convertire argomenti prima della chiamata di una funzione o per convertire da tipo classe a tipo nativo, proprio perchè non abbiamo accesso alle classi astratte rappresentate dai tipi nativi per poterlo definire lì dentro!

Un modo migliore per effettuare le conversioni di tipo da classe a classe e per poter fare anche quelle da classe a tipo nativo è usare gli operatori di cast descritti nel prossimo paragrafo.

Ndt: Sperimentate con questo esempio:

```
class test
{
public:
//  test(int) {}        // togli il commento iniziale per vedere che succede
  test & operator= (int)
  {
    // ...
    return *this;
  }
};

void foo(const test & c)        // oppure test c
{
}

int  main()
{
  foo(2);                       // qui una temporanea di tipo test viene creata
                                // e subito distrutta dopo il ritorno
  return 0;
}
```

Questo esempio mostra anche come i costruttori possono essere usati (anche implicitamente) come convertitori di tipo (da tipo nativo a classe o anche da classe a classe). Notare che nonostante venga creata una variabile temporanea e la si inizializzi, l'operatore=(int) non viene usato, in quanto non viene effettuato un assegnamento della variabile temporanea, perciò avrete un errore se il costruttore `test(int)` non c'è. Insisto su questo punto: se ad es. scrivete `test t=3;` la variabile t non viene creata e poi assegnata ma bensì creata e inizializzata e viene chiamato il costruttore `test(int)` e non prima il costruttore vuoto e poi l'`operator=(int)`. `test t=3;` infatti è lo stesso di `test t(3);`. Notate quindi che in

C++ c'è differenza tra inizializzazione e assegnamento. Sono due concetti un po' diversi.

Ndt: da notare che facciamo restituire all'operatore= un riferimento all'oggetto appena assegnato. Notare anche l'istruzione return *this; che deve esserci, a meno che non definite void il tipo di ritorno (questo però non permette di concatenare gli operatori di assegnamento come si fa in C con i tipi nativi; es.: a=b=c; a=b=8;).
Qual è la ragione per cui abbiamo preferito far restituire un riferimento, anzichè un valore come nel prototipo `Complex operator= (float f);` ? Giusto per evitare la chiamata del costruttore di copia di default per costruire un oggetto temporaneo da ritornare. Questo programma vi aiuterà a capire la differenza e a spiegare quello che avviene nel caso di assegnamenti multipli. Vi invito a sperimentare ulteriormente col vostro compilatore, per risolvere eventuali altri dubbi. Come potete vedere, un modo semplice per capire quali funzioni vengono chiamate è quello di far stampare delle stringhe.

```
#include <stdio.h>

class Complex
{
  float re, im;
public:
  Complex (float r = 0, float i = 0)
  {
    re = r;
    im = i;
  }
  Complex (const Complex& complex)
  {
     printf("costruttore di copia chiamato\n");
     // fa il lavoro di quello di default
     re = complex.re;
     im = complex.im;
  }
  Complex operator=(const Complex& complex)
  {
     printf("operator=(Complex&) chiamato\n");
     // fa il lavoro di quello di default
     re = complex.re;
     im = complex.im;
     return *this;   // oppure complex
  }
  Complex /*&*/ operator= (float f)     // provare a uncommentare il &
  {
    printf("operator=(float) chiamato\n");
    re = f;
    im = 0;
    return *this;
  }
};

int main()
{
  Complex a (100, 200);
  Complex b (10, 20);
  Complex c (1, 2);

  c = b = a = 12;
```

```
  return  0;
}
```

Operatori di cast

Gli operatori di cast consentono di convertire da classi in tipi standard. Per esempio, se dichiariamo un `operator int()` dentro una classe, saremo in grado di convertire un oggetto di questa classe in un valore intero. Questo operatore sarà chiamato anche implicitamente, dal compilatore.

Devono essere delle funzioni membro. Il tipo di ritorno non va mai indicato, perchè è implicitamente noto (ovviamente è lo stesso del nome dell'operatore).

```
class Complex
{
  float re, im;
public:
  Complex (float r = 0, float i = 0) { re = r; im = i; }
  operator float() { return re; }
};
void foo (float f)
{
  printf ("%f\n", f);
}
{
  Complex a (10, 12);
  float b = (float) a;  // Chiamata esplicita dell'operatore float()
  float c = a;          // Chiamata implicita dell'operatore float()
  foo ((float) a);      // Chiamata esplicita dell'operatore float()
  foo (a);              // Chiamata implicita dell'operatore float()
}
```

Si possono usare gli operatori di cast anche per convertire oggetti in altri oggetti. La sintassi è la stessa, basta usare il nome della classe a cui convertire invece di `float` nell'esempio precedente.

Gli operatori `new` e `delete`

Quando fai l'overload degli operatori `new` e `delete`, devi preoccuparti di realizzare l'allocazione di memoria. Questa è la ragione per cui non dovresti farlo, a meno che non sai esattamente quello che vuoi.

La loro dichiarazione deve essere:

```
#include <sys/types.h>
#include <new.h>

static void * operator new (size_t);
static void operator delete (void *, size_t);
```

Costruttori e distruttori

Questa sezione contiene:

Costruttori

Un membro costruttore è una funzione chiamata quando l'oggetto viene dichiarato o allocato dinamicamente. Il suo nome deve essere lo stesso della classe a cui appartiene, ma ci possono essere diversi costruttori sovraccarichi (vedi overloading delle funzioni), a patto che la loro lista di parametri sia differente. Un costruttore **non** ha valore di ritorno (non si deve specificare nemmeno void, il lato sinistro del nome nella definizione deve essere lasciato vuoto). Esso non può essere dichiarato static.

Esempio:

```
class Complex
{
  float i, r;
public:
  Complex();                  // Primo semplice costruttore
  Complex (float, float);     // Secondo costruttore sovraccaricato
};
Complex::Complex()
{
  i = 0;
  r = 0;
}
Complex::Complex (float ii, float rr)
{
  i = ii;
  r = rr;
}
```

Ma come facciamo a chiamare un costruttore? Questo dipende se stai usando l'operatore "new" o no. Senza "new":

```
Complex x();          // O per farla breve:
Complex x;            // Chiamata del primo costruttore
Complex y (2, 2);     // Chiamata del secondo costruttore
```

Devi solo mettere i parametri dopo il nome della variabile, e il corrispondente costruttore sarà chiamato. Con l'operatore "new", si fa così:

```
Complex *x = new Complex();         // O più brevemente
Complex *y = new Complex;           // Chiamata del primo costruttore
Complex *z = new Complex (2, 2);    // Chiamata del secondo costruttore
...
```

```
delete x;
delete y;
delete z;
```

Il compilatore genera del codice che fa le seguenti cose:

- Alloca memoria sullo stack (allocazione statica) o sullo heap (allocazione dinamica).
- Inizializza l'oggetto (chiama i costruttori delle classi padre (vedi qui) o delle classi degli oggetti contenuti nell'oggetto (vedi più avanti).
- Chiama il costruttore con i parametri giusti.

Se gli oggetti sono dichiarati al di fuori di ogni funzione come variabili globali, i loro costruttori sono chiamati prima che la funzione "main" inizi. Ci sono due costruttori particolari (di default e di copia) che sono descritti in seguito.

Distruttori

Dopo aver visto la definizione di un costruttore, l'utilità di un distruttore appare ovvia: è chiamato quando l'oggetto viene rilasciato dalla memoria. Il suo nome è il nome della classe preceduto da ~ (tilde). Non ha nessun valore di ritorno (neanche void). **Non** prende parametri, dal momento che il programmatore non ha mai bisogno di chiamarlo direttamente.

Esempio:

```
class Complex
{
  float i, r;
public:
  Complex();                  // Primo semplice costruttore
  Complex (float, float);     // Secondo costruttore "overloaded"
  ~Complex();                 // Distruttore
};
Complex::~Complex()
{
  printf ("Destructor called.\n");
}
...
{
  Complex *x = new Complex (5, 5);
  Complex y;
  ...
  delete x;                   // Chiamata del distruttore per x
}                             // Chiamata implicita del distruttore per y
```

Quando "cancelliamo" un oggetto con delete, ecco cosa facciamo:

- Chiamiamo il distruttore
- Liberiamo la memoria occupata dall'oggetto

Con un oggetto statico, non possiamo determinare quando il distruttore sarà chiamato, perchè le regole del C++ non lo indicano. Sappiamo solo che teoricamente esso sarà chiamato.

Costruttore e distruttore di default

Quando non implementi il distruttore, viene usato un distruttore di default, che non fa nulla. Ma non appena fornisci codice per il distruttore, questo codice sarà utilizzato.

C'è anche un costruttore di default, e anche questo diventa inutilizzabile quando un qualsiasi altro costruttore è dichiarato.

Puoi allora usare solo i costruttori che hai fornito.

```
class Complex
{
public:
  Complex (float, float);   // È dichiarato un solo costruttore
};
```

Dopo questa, l'unico modo per costruire un oggetto Complex è di dare 2 parametri al costruttore.

```
Complex x;           // Il compilatore rifiuterà questa riga
Complex y (2, 2);    // è ok chiamare il giusto costruttore
Complex z[10];       // Non permesso (continua a leggere)
```

Con un array di oggetti, dovremmo fornire gli argomenti del costruttore per ciascun elemento del vettore. Ma a causa della sintassi che deve uniformarsi con quella dei tipi nativi, un array di oggetti può essere dichiarato soltanto se l'oggetto ha un costruttore che non prende nessun argomento, o un solo argomento, ma non più di uno:

```
class Complex
{
public:
  Complex (float = 0, float = 0);
};

Complex v1[5] = { 0, 1, 2, 3, 4 };   // per ogni elemento il costruttore è
                                     // chiamato con gli argomenti (x, 0)
void Create (int n)
{
  Complex v2[n] = { 0, 1, 2};  // I primi tre elementi sono inizializzati
                               // con (x, 0), e gli altri con (0,0)
}
```

Ndt: notare che nel caso del vettore v1 i valori di inizializzazione 0, 1, 2, 3, 4 sono le parti immaginarie dei numeri complessi, perchè il primo argomento del costruttore `Complex (float, float)` (vedi sopra) era usato come parte immaginaria. Se volete che accada il contrario, dovete definire il costruttore così, invertendo l'ordine degli argomenti:

```
Complex::Complex (float rr, float ii)
{
  r = rr;
  i = ii;
}
```

Fate come preferite: infatti anche in matematica alcuni preferiscono scrivere i numeri complessi come `2j+1` altri come `1+2j` (che è lo stesso). Altri in entrambe le forme, senza adottare una sintassi precisa. Sopra quando è scritto `(x, 0)` per il primo numero (x) si intende la parte immaginaria e per il secondo (0) quella reale, perciò la convenzione adottata è quella di passare i numeri complessi al costruttore nella forma `(parte imm, parte real)` del tutto simile a `(parte imm)j + parte real`.

Costruttore di Copia

Il costruttore di copia è l'altro costruttore di default. Come argomento prende solo un riferimento ad un altro oggetto della stessa classe. Quando nessun costruttore di copia è dichiarato, il compilatore ne usa uno di default che copia ogni campo dell'oggetto sorgente nell'oggetto destinazione e non fa nient'altro. Se ne dichiari uno, dovrai copiare tutti i campi che vuoi manualmente. Da notare l'uso della keyword "const", necessaria al compilatore per riconoscere un costruttore di copia.

```
class Vector
{
  int n;
  float *v;
public:
  Vector();
  Vector (const Vector &);
};
Vector::Vector()
{
  v = new float[100];
  n = 100;
}
Vector::Vector (const Vector &vector)
{
  n = vector.n;                          // Copia del campo n
  v = new float[100];                    // Crea un nuovo array
  for (int i = 0; i < 100; i++)
    v[i] = vector.v[i];                  // Copia l'array
}
```

I costruttori di copia sono necessari se effettui allocazione di memoria all'interno di un oggetto, come nell'esempio visto. In questo caso, non puoi semplicemente limitarti a copiare il puntatore (come fa il costruttore di copia di default), ma hai davvero bisogno all'atto della copia di allocare una parte della memoria, in modo che i valori associati ai due oggetti possano essere differenti. Quindi i costruttori di copia risolvono il problema della condivisione di memoria indesiderata che si avrebbe all'atto dell'assegnamento di un oggetto con parti dinamiche (cioè con membri puntatori).

Questo costruttore di copia è chiamato ognivolta che dichiari un oggetto di una classe e gli assegni un valore con una sola istruzione.

```
Vector a;                    // Costruttore vuoto
Vector b (a);                // Costruttore di copia
Vector c = a;                // Costruttore di copia (equivalente alla precedente
istruzione)
Vector *d = new vector (a);  // Costruttore di copia
```

Puoi utilizzare il costruttore di copia anche con i tipi nativi, come int o char:

```
int a = 2;            // Assegna un valore
int b (2);            // Assegna un valore
```

Includere oggetti in altri oggetti

Quello che vogliamo fare è semplice. Vogliamo includere una classe come membro di unl'altra. La dichiarazione è piuttosto semplice:

```
class A
{
  int a;
public:
  A (int);
};
class B
{
```

```
  int b;
  A a_element;
public:
  B (int, int);
};
```

In questo esempio, stiamo includendo un oggetto della classe A in un oggetto della classe B. La cosa interessante è quello che succede quando chiamiamo il costruttore di B. Dopo che la memoria è stata allocata, il linguaggio stabilisce che deve essere chiamato il costruttore per ogni oggetto incluso, e solo dopo di ciò il costruttore dell'oggetto contenitore o figlio (cioè B). Ma quale costruttore di A verrà chiamato (nel caso ve ne sia più di uno), e come specificare altrimenti? Quando definisci il costruttore di B, devi specificare quale costruttore di A deve essere chiamato, e con quali argomenti. Ecco come si fa:

```
B::B (int aa, int bb) : a_element (aa)
{
  // Codice per il costruttore
  b = bb;
}
```

Ndt: in questo esempio non c'è un costruttore senza argomenti, perciò se non specifichiamo nella definizione di B quale costruttore chiamare, il compilatore tenterà di inserire una chiamata ad un costruttore senza argomenti per la classe A, cioè a `A::A ()` ma poichè questo non è definito ci darà un errore. Ricordate infatti che abbiamo detto che il costruttore di default (che è senza argomenti) viene interdetto quando viene definito un altro costruttore.
Sperimentate questa situazione col vostro compilatore.

Note sintattiche: notare l'uso di ":" prima della chiamata al costruttore di A. Se molti costruttori devono essere chiamati, basta separare le loro chiamate con delle virgole (come nell'esempio sotto).
Nota che la sintassi per inizializzare l'oggetto contenuto nell'oggetto è all'atto della chiamata di un costruttore è la stessa usata nelle normali dichiarazioni (tipo `int b (2);`); solo il tipo della variabile da inizializzare viene omesso, in quanto già noto dalla definizione della classe.
Un altro modo più semplice per scrivere il costruttore di B è di usare i costruttori dei tipi nativi:

```
B::B (int aa, int bb) : a_element (aa), b (bb)
{
  // Codice per il costruttore
}
```

Quando viene chiamato il costruttore di copia di default di un oggetto contenente altri oggetti, sarà implicitamente chiamato anche il costruttore di copia di ciascuno degli oggetti contenuti. Nel caso abbiamo bisogno di sovrapporre il costruttore di copia dell'oggetto contenitore, dobbiamo chiamare noi esplicitamente il costruttore di copia per ognuno degli oggetti contenuti: la sintassi per chiamare questi costruttori è la stessa usata nell'esempio precedente.

Ndt: ok, ok eccovi un esempio chiarificatore:

```
#include <stdio.h>

class A
{
public:
  A()
  {
    printf("costruttore vuoto di A chiamato\n");
  }
  A (const A&)
  {
    printf("costruttore di copia A chiamato\n");
```

```
  }
};

class B
{
  A aA;
public:
  B()
  {
    printf("costruttore vuoto di B chiamato\n");
  }
  B (const B& b) : aA(b.aA)     // provare ad eliminare : aA(b.aA)
  {
    printf("costruttore di copia B chiamato\n");
  }
};

main()
{
  B aB;
  B anotherB(aB);
}
```

senza `: aA(b.aA)` il costruttore di copia di A non viene chiamato e nel membro aA dell'oggetto `anotherB` non viene copiato l'oggetto membro aA dell'oggetto `aB`. Piuttosto viene richiamato il costruttore vuoto A() sul membro aA di anotherB (provare per credere):

output con la riga `B (const B& b) : aA(b.aA)`

```
costruttore vuoto di A chiamato
costruttore vuoto di B chiamato
costruttore di copia A chiamato
costruttore di copia B chiamato
```

output con la riga `B (const B& b)`

```
costruttore vuoto di A chiamato
costruttore vuoto di B chiamato
costruttore vuoto di A chiamato
costruttore di copia B chiamato
```

Elementi Statici

Gli elementi statici esistono per una intera classe, che sia instanziata o no. Perciò le variabili static sono uniche, sia che ci siano oggetti di quella classe che no, e le funzioni static possono essere richiamate senza nessun oggetto.

Variabili Static

Per dichiarare una variabile statica, basta aggiungere la keyword `static` all'inizio della dichiarazione. Dopo averla dichiarata, occorre anche che ci sia una sua definizione in un file di codice. Ogni istanza della classe farà riferimento alla stessa variabile static.

```
// A.h
class A
{
  static int number;
public:
  A() { number++; }
  ~A() { number--; }
};

// A.C
#include "A.h"
int A::number = 0;
```

In questo esempio, la variabile interna `number` viene usata per contare il numero di oggetti della classe A che sono attualmente in memoria. Le stesse regole di protezione che si applicano ai normali membri si applicano anche a quelli static. Un membro `private` può essere solo inizializzato, come nell'esempio precedente, una variabile `public` invece può essere anche modificata.

Funzioni Static

Le funzioni Static esistono e possono essere chiamate senza alcun instanziazione della classe che le contiene. Quindi, esse possono modificare solo membri statici. Per dichiarare una funzione static occorre mettere la parola chiave `static` prima della sua dichiarazione.

```
class A
{
  static int number;
public:
  A() { number++; }
  ~A() { number--; }
  static int GetNumber();
```

```
};
int A::GetNumber()
{
  return number;
}
```

Dopodichè puoi chiamare anche normalmente una funzione statica, usando un oggetto, cioè invocandola su un oggetto. Ma si può chiamare una funzione static anche senza invocarla su alcun oggetto. Basta mettere il nome della classe seguito da un "::" e dal nome della funzione:

```
A a_obj; int n;
n = a_obj.GetNumber();     // Normale chiamata
n = A::GetNumber();        // Non occorre passare alcun oggetto
```

Funzioni costanti

Le funzioni costanti sono funzioni membro di una classe che non modificano nessun valore delle variabili interne. Esse possono quindi essere chiamate anche su un oggetto const. Per dichiararle, occorre solo mettere la keyword "const" dopo la dichiarazione **e ripeterla** dopo la definizione della funzione:

```
#include <stdio.h>
class A
{
  int a;
public:
  A (int aa = 1) { a = aa; }
  void Display() const;
};
void A::Display() const
{
  printf ("%d", a);
}
const A a_obj (2);
int main (int, char **)
{
  a_obj.Display();      // Non sarebbe permessa se Display non fosse const
}                       // con gcc avrei:
                        // Error: passing `const A' as `this' argument of `void
A::Display()' discards qualifiers
```

Ovviamente puoi chiamare funzioni const anche su normali oggetti.

Ereditarietà di Classe

Questo documento contiene:

Introduzione

L'ereditarietà è un meccanismo che consente ad una classe di ereditare ogni membro della classe-padre, e poi aggiungere e modificare le cose secondo le nuove funzionalità da realizzare.

Quando vuoi usare tutti i membri di una classe padre senza doverli dichiarare di nuovo uno alla volta, basta mettere una sola dichiarazione nell'header della classe "figlia":

```
class father
{
public:
  int a;
};

class A : public father
{
public:
  int b;
};
```

In questo esempio, la classe A avrà un membro pubblico chiamato `b`, come al solito. Ma poichè è dichiarata come figlia della classe `father`, avrà anche un membro `a` dichiarato implicitamente.

Quando una classe eredita da un'altra, ne eredita tutti i membri. Questo porta a due problemi:

1. proteggere i membri di una classe in modo che i suoi potenziali figli non siano in grado di accederli.
2. far sì che una classe figlia impedisca l'accesso da parte del mondo esterno alla parte ereditata dalla sua classe padre.

Nelle prossime sezioni vedremo come è semplice ottenere ciò.

Dichiarazioni `private`, `protected` e `public`

Come abbiamo visto una classe può dichiarare membri pubblici. In questo modo, ognuno può accederli. Questo però non è un concetto che riguarda la programmazione orientata agli oggetti. Facilita il debugging, ma non usarlo troppo. Invece, usa delle funzioni membro inline di accesso (in inglese dette "accessors") che ritornano il valore dei componenti, quando richiesto. Così facendo si può rendere una variabile a sola-lettura in modo pulito.

```
class A
{
  int a;
```

```
public:
  int GetA() { return a; }
};
```

Infatti, puoi dichiarare ogni membro (variabile) privato. Nessuno sarà in grado di utilizzarli direttamente. Questa è una buona pratica nella programmazione orientata agli oggetti.

Ndt: qui ovviamente si consiglia di non permettere l'accesso diretto ai membri dati, tranne che in casi rari. Infatti nella maggior parte dei casi i membri dati devono essere gestiti esclusivamente tramite i metodi (le funzioni) dell'oggetto e una loro gestione esterna impropria può portare a delle situazioni di incoerenza nell'oggetto e quindi di errori, oltre che rendere i programmi molto meno "mantenibili".
I membri funzione ovviamente devono essere public se si vuole poter mandare i corrispondenti "messaggi" all'oggetto dal mondo esterno.
Se una funzione invece serve solo ad uso interno dell'oggetto ed è pericoloso e/o inutile metterla a disposizione del mondo esterno (non corrisponde ad alcun messaggio) meglio non renderla public, ma `private` o `protected` a seconda dei casi.

Un'altra cosa: ho provato questo codice con tutti i miei compilatori e lo hanno accettato senza problemi, senza manco sputar fuori un warning:

```
class A
{
  int a;
public:
  int & GetA() { return a; }
};


int main()
{
  A unA;

  unA.GetA() = 5;

  return 0;
}
```

Come potete vedere tracciando il programma e guardando i contenuti dell'oggetto A, la chiamata `unA.GetA()` ritorna il precedente valore di a (che è indefinito all'inizio); per la verità essa ritorna un riferimento a quel valore. Un riferimento è qualcosa di analogo ad un puntatore C che viene però automaticamente referenziato (vedi riferimenti), perciò l'assegnazione è possibile. Ebbene in questo caso si modifica un membro `private`, contravvenendo alle regole della OOP (abbreviazione di "programmazione orientata agli oggetti"). Un buon compilatore dovrebbe impedirlo, o almeno tirar fuori un avvertimento, ma con DJGPP e Visual C++ non ho ottenuto niente! Probabilmente i programnmatori dei compilatori si sono scordati di tener conto del problema!

In alcuni casi potresti volere che solo i figli di una classe abbiano il permesso di accedere ad alcuni suoi membri. Quindi, non ci sarà modo di accedere a quei membri dall'esterno, ma all'interno sia della classe padre che figlia, ogni modifica sarà permessa. Questo tipo di dichiarazioni sono dette protette e precedute dalla keyword `protected`.

Ndt: notare la differenza con le dichiarazioni `private`: esse non sono accessibili nè all'esterno della classe, nè dai suoi discendenti.

```
class father
{
protected:              // provate ad commentarla
  int a;
};
```

```
class A : public father
{
  int GetA() { return a; }    // può accedere ad a
};
...
{
  father f;
  int x = f.a;                    // rifiutato dal compilatore
}
```

In altre parole, con questo tipo di ereditarietà, cioè l'ereditarietà pubblica, le classi figlie hanno gli stessi diritti di accesso ai membri `public` e `private` degli utilizzatori esterni: i `public` sono manipolabili direttamente dalle classi figlie, i `private` no. Invece i membri `protected` sono speciali: le classi figlie possono usarli direttamente, ma non il resto del mondo (cioè altre classi non figlie, normali funzioni, il main). Tuttavia ci sono altri tipi di ereditarietà, il cui scopo è cambiare alcuni specificatori di accesso per i membri ereditati; la cosa più comune è far sì che membri pubblici siano ereditati come protetti o privati, anzichè come pubblici. Continuate a leggere...

Ereditarietà privata, protetta o pubblica

Un figlio di una classe può scegliere i permessi di accesso ai membri pubblici ereditati da suo padre. Può scegliere di mantenerli pubblici, rispettando totalmente le scelte della classe padre. In tal caso occorre inserire la parola `public` tra ":" e il nome della classe padre:

```
class father
{
public:
  int a;
};

class A : public father
{
  int GetA() { return a; }    // può accedere ad a
}
...
{
  A a_obj;
  x = a_obj.a;                // si possono accedere membri pubblici ereditati come
tali
}
```

Una classe figlia può anche proteggere i membri che ha ereditato da suo padre, facendo sì che le proprietà ereditate da suo padre siano "interne". La ereditarietà deve essere dichiarata di tipo protetto (tutti i membri pubblici della classe padre diventeranno protetti) o di tipo privato (se invece si vuole che tutti i membri pubblici diventino privati). La differenza tra derivazione protetta e privata si manifesta solo nei figli dei figli della classe padre, poichè per il resto del mondo (main, normali funzioni o altri classi che usano la classe figlia e nipote) i campi pubblici della classe padre ereditati come protected o private sono in entrambi i casi inaccessibili.

```
class father
{
public:
  int a;
};
```

```
class A : private father
{
  int GetA() { return a; }    // a si può accedere qui (a è diventato un membro
privato)
}
...
{
  A a_obj;
  x = a_obj.a;                // rifiutato dal compilatore: non si può accedere al
membro privato a
}
```

Nota che puoi omettere la keyword `public`, `protected` o `private`. In tal caso si intende `private` per default.

La tabella seguente riassume come cambiano i livelli di accesso in una classe figlia a seconda dei 3 tipi di derivazione possibili:

classe Padre	**classe Figlia**		
	derivazione `public`	*derivazione* `protected`	*derivazione* `private`
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`non accessibili`	`non accessibili`	`non accessibili`

Costruttori e distruttori

Come per gli oggetti membri, devi trasmettere i parametri al costruttore della classe padre, che è chiamato prima del costruttore della classe figlia.

La sintassi è proprio la stessa:

```
class Father
{
  int a;
public:
  Father (int aa) { a = aa; }
};

class Child : public Father
{
  int b;
public:
  Child (int aa) : Father (aa) {}   // Ecco un costruttore inline
  Child (int, int);                 // ed uno che è una normale funzione
};

Child::Child (int aa, int bb) : Father (aa)
{
  b = bb;
}

/* alternativa:
Child::Child (int aa, int bb) : Father (aa), b(bb)
```

```
{
}*/
```

Provate ad omettere la parte di linea `: Father (aa)` nella definizione del costruttore `Child (int aa)`. Cosa succede? abbiamo già fatto vedere una situazione simile qui.

Overloading di funzioni membro

Quando una classe eredita da un'altra, eredita tutte le variabili membro (statiche o non). Non può rifiutare l'eredità :). Tuttavia nel caso di una funzione, una classe figlia può scegliere se rimpiazzare una funzione della classe padre o meno.

```
class Father
{
public:
  void MakeAThing();
};
class Child : public Father
{
public:
  void MakeAThing();
};
...
{
  Father father;
  father.MakeAThing();    // metodo MakeAThing di Father chiamato
  Child child;
  child.MakeAThing();     // metodo MakeAThing di Child chiamato
}
```

Nella nuova versione della funzione, potresti aver bisogno di chiamare la versione della funzione della classe padre, o persino una funzione globale con lo stesso nome. Come fare? Una chiamata a MakeAThing() della classe Child all'interno di MakeAThing() della classe Child stessa è una chiamata ricorsiva e non corrisponde a nessuna delle due chiamate precedenti. La cosa è possibile specificando più informazioni al compilatore all'atto della chiamata, usando :: e il nome della classe a cui appartiene la funzione che vuoi chiamare. Ecco un esempio che chiarisce tutto:

```
void MakeAThing();
class Father
{
public:
  void MakeAThing();
};
class Child : public Father
{
public:
  void MakeAThing()
  {
    Father::MakeAThing();  // viene chiamato il metodo di Father
    ::MakeAThing();        // viene chiamata la funzione globale
    //MakeAThing();        // questa è una chiamata ricorsiva!
  }
};
...
{
  Father father;
```

```
  father.MakeAThing();    // metodo MakeAThing di Father chiamato
  Child child;
  child.MakeAThing();     // metodo MakeAThing di Child chiamato
  MakeAThing();           // chiamata una funzione globale
}
```

Polimorfismo

Uno dei principali vantaggi dell'ereditarietà è che una classe figlia può sempre prendere il posto della sua classe padre, ad esempio nell'argomento di una funzione. La classe figlia può essere vista come una classe con due (o anche più!) identità. Ecco perchè questa proprietà è detta polimorfismo, parola che deriva dal greco e significa appunto "dalle molte forme".

```
class Father
{
...
};
class Child : public Father
{
...
};
...
void ExampleFunction (Father &);
...
{
  Father father;
  ExampleFunction (father);   // Normale chiamata
  Child child;
  ExampleFunction (child);    // un oggetto child è considerato come uno di tipo
father
}
```

Questo proprietà vale per gli oggetti, i puntatori agli oggetti e i riferimenti agli oggetti. Così definendo una classe figlia di una classe, puoi apportare dei miglioramenti rispetto a quest'ultima e nello stesso tempo essere in grado di usarne tutte le caratteristiche e le funzionalità. Questo può essere fatto anche se non hai il codice sorgente della classe padre! È questa la ragione principale del successo della programmazione orientata agli oggetti.

Funzioni Virtuali

Definizione

Date prima un'occhiata a questo esempio:

```
class Father
{
public:
  void MakeAThing();
};
class Child : public Father
{
public:
  void MakeAThing();
};
...
{
  Father *obj;
  obj = new Child();
  obj->MakeAThing();    // Quale metodo verrà chiamato ?
}
```

Una spiegazione semplice (non stiamo ancora parlando di funzioni virtuali però questo esempio serve ad introdurre il discorso): `obj` è di tipo `Father *`. La classe `Child` è figlia della classe `Father` (come consuetudine... Ndt: in inglese Child significa proprio "figlio/figlia" e Father significa "padre"). Pertanto è possibile scrivere un valore di tipo `Child *` (quello ritornato dall'operatore new) nella variabile `obj` (sì come è possibile scrivere un valore di tipo `Child` in una variabile di tipo `Father`).

Allora che succede? Quasi sicuramente penserai: siccome `obj` è dichiarato dalla classe `Child`, sarà il metodo `MakeAThing` della classe `Child` ad essere chiamato. Ma **non** è così. Rifletti: il compilatore sa solo che `obj` è di tipo `Father *`, perciò chiamerà il metodo di `Father`. Infatti non c'è niente che dica al compilatore che la funzione membro da chiamare dipende dalla classe.

Ebbene, esiste un modo per indicare questo al compilatore. Se dichiari le funzioni "overloaded" come virtuali, tramite la keyword `virtual`, indichi al compilatore, che sa che questa funzione cambia da una classe ai suoi figli, di scegliere dinamicamente la funzione da chiamare, in modo che corrisponda al tipo dell'oggetto. Confrontate questo esempio con quello precedente:

```
class Father
{
public:
  virtual void MakeAThing();
};
class Child : public Father
{
public:
  virtual void MakeAThing();
};
...
```

```
{
  Father *obj;
  obj = new Child();
  obj->MakeAThing();    // ora viene chiamato il metodo di Child !
}
```

Classi Astratte

In C++ si possono definire delle funzioni virtuali pure. Una classe che contenga questo tipo di funzioni è essa stessa solo virtuale e non può essere instanziata. Ma ciascuna delle sue classi figlie dovrà dichiarare quella funzione (vedi overloading di funzioni membro), e inoltre dovrà poi ridefinirla.

Per dichiare una funzione come solo virtuale, aggiungi "= 0" alla fine della sua dichiarazione.

```
#include <stdio.h>

class Father
{
public:
  virtual void Display() = 0;
};

class Child : public Father
{
  void Display() { printf ("Sono una figlia della Classe Father.\n"); }
};

int main()
{
  Father *f = new Father();  // NON PERMESSO (la classe Father è solo-virtuale)
  Child *c = new Child();    // OK. Child è una normale classe (non virtuale)
  Father *f = new Child();   // Puoi dichiarare dei puntatore a Father

  return 0;
}
```

Notare che si può definire un puntatore ad una classe astratta, ma non si può instanziarla.

Le classi astratte sono usate per dichiarare componenti comuni per le classi che ne derivano. Quando erediti da una classe astratta in una classe, dopo aver definito un suo membro dati (ehi, non definitelo `private`!), puoi usarlo in tutte le funzioni definite nelle classe figlie.

```
class Component
{
public:
  virtual void Display() = 0;
};

class Rectangle : public Component
{
public:
```

```
  virtual void Display()
  { ... }
};

class Square : public Component
{
public:
  virtual void Display()
  { ... }
};

// Dichiara una lista di puntatori a Component
Component *list[10];

...

{
  // Qual'è il bisogno di dichiarare un oggetto Component ?
  // Nessuno, sono i suoi figli che interessano
  list[0] = new Square;
  list[1] = new Rectangle;
  ...
  for (int i=0; i<10; i++)
    list[i]->Display();         // chiamata di funzione "dinamica"
}
```

Ndt: In questo caso la classe `Component` è una classe basa astratta, che serve solo a riunire le proprietà (in tal caso un solo metodo, `Display()` ma potrebbero esserci più metodi e campi dati) comuni all'intera gerarchia di classi che da essa deriva. Il meccanismo delle funzioni virtuali, come mostrato nell'esempio precedente, consente di avere una azione (come `Display()`) che ha lo stesso nome su ogni oggetto della gerarchia di classi su cui agisce, ma è implementata in modo diverso e il compilatore sceglierà automaticamente la versione giusta da chiamare. In virtù del polimorfismo è infatti possibile implementare liste eterogenee di oggetti (il vettore `list` ne è un semplice esempio: è una lista statica di oggetti dinamici). Nell'esempio viene chiamata prima la funzione `Display()` di `Square`, poi quella di `Rectangle`, ecc...

Ereditarietà multipla

Una estensione naturale del concetto di ereditarietà finora visto è quella di avere più di una classe padre. Per esempio, considerate un aeroplano: esso è un oggetto volante ma è anche un veicolo, perciò deve avere entrambe le caratteristiche.

Dichiarazione

Per indicare la relazione di ereditarietà multipla, basta aggiungere alla lista dei padri quelli che vuoi, ciascuno preceduto da una delle keyword `public`, `protected` o `private` (per la spiegazione dei vari tipi di ereditarietà ti rimando qui:

```
class father1
{
  int a;
};
class father2
{
  int b;
};
class Child : public father1, private father2
{
  int c;
};
```

Notare che l'ereditarietà semplice è semplicemente un caso particolare dell'ereditarietà multipla.

Problemi con l'ereditarietà multipla

Date un'occhiata a questo codice:

```
class GrandFather
{
protected:
  int a;
};
class Father1 : public GrandFather
{
protected:
  int b;
};
class Father2 : public GrandFather
{
protected:
  int c;
};
class Child : public Father1,
```

```
              public Father2
{
protected:
  int d;
  void foo();
};
```

La variabile `a` della classe `GrandFather` è ereditata da entrambe le classi `Father1` e `Father2`. Quando viene creata la classe `Child`, avrà due distinte variabili `a` ereditate o una sola? Per default ne avrà due. Per indicare l'una o all'altra devi aggiungere `Father1::` o `Father2::` prima di riferirti alla variabile `a`.

```
void Child::foo()
{
  d = Father1::a;
// Oppure
  d = Father2::a;
}
```

Ma puoi anche far sì che nella classe `Child` le variabili `a` ereditate da `Father1` e `Father2` siano le stesse. Basta mettere la keyword `virtual` dopo la parola chiave `public`, `protected` o `private`, e nessun membro verrà duplicato. In questo modo puoi ridurre la quantità di memoria necessaria all'applicazione.

```
class GrandFather
{
protected:
  int a;
};

class Father1 : public GrandFather
{
protected:
  int b;
};

class Father2 : public GrandFather
{
protected:
  int c;
};

class Child : public virtual Father1,
              public virtual Father2
{
protected:
  int d;
  void foo();
};

void Child::foo()
{
```

```
  d = a;             // OK
  d = Father1::a;    // In questo caso è la stessa dell'istruzione precedente
  d = Father2::a;    // In questo caso è la stessa dell'istruzione precedente
}
```

Strumenti avanzati del C++

Qui presentiamo alcuni strumenti di programmazione avanzata in C++. Hai bisogno di avere una buona comprensione di quello che è stato fatto prima.

Ogni sezioni descrive un punto particolare a sè stante:

- Stream: Un modo comodo di accedere all'input/output standard.
- Template: Come rendere generici i propri oggetti o normale codice C/C++.
- Eccezioni: Come gestire i problemi e gli errori (occorre un compilatore C++ recente).

Stream

Gli stream sono un modo di accedere all'input/output standard, sia alla consolle che ai files. Il meccanismo uniforme è tutto basato sull'overloading degli operatori << e >>. Occorre includere il file **iostream.h** per poter utilizzare gli stream.

Stream di Output

Uno stream di output è del tipo classe `ostream`. Per i tipi standard, questa classe ridefinisce l'operatore <<:

```
ostream & operator << (base_type);
```

Notare che l'operatore restituisce ancora uno stream (anzi per risparmiare tempo e memoria un riferimento ad uno stream), e questo fatto permette di concatenare l'output usando appunto più operatori << concatenati (come si fa quando si scrive 1+2+3, ecc..!):

```
stream << "Time is" << hour << ":" << minute << ":" << second;
```

Ndt: la restituzione di un riferimento non è comunque solo una questione di ottimizzazione; è necessaria affinchè quando più operatori << son concatenati come nell'esempio precedente, le modifiche fatte all'oggetto stream dal secondo operatore in poi siano conservate, cioè non agiscano solo una copia temporanea, ma sull'oggetto stream stesso specificato all'inizio della espressione.

Gli stream standard (potete pensarli come istanze dei tipi classe `ostream`, `istream` o di alcuni loro derivati) sono `cout` per lo standard output, `cerr` per lo standard error bufferizzato, e `clog` per lo standard error senza buffer. Per esempio per far scrivere l'ora, basta digitare:

```
cout << "Time is" << hour << ":" << minute << ":" << second;
```

Un'altro modo di scrivere negli stream è di usare la funzione membro `put`. Essa è dichiarata come segue:

```
ostream & put (char);
```

ed è una funzione membro della classe `ofstream`. Si usa in modo molto semplice:

```
#include <iostream.h>

char Message[] = "This is a message.\n";
//...

int main()
{
  int i=0;
  while (Message[i])
    cout.put (Message[i++]);
```

```
  return(0);
}
```

Stream di Input

L'operatore >> è così sovrapposto:

```
istream & operator >> (& base_type);
```

Lo Standard Input è l'oggetto `cin`, perciò si usa in questo modo:

```
{
  int hh;
  int mm;

  cin >> hh >> mm;   // legge due interi da standard input
}
```

Puoi anche usare la funzione membro `get` se vuoi leggere un solo carattere o un insieme di caratteri:

```
istream & get (char &c);
istream & get (char *p, int max, char separator);
```

Overloading degli operatori << e >>

Puoi ridefinire gli operatori << e >> per permettere ai tuoi oggetti di uniformarsi all'architettura standard per l'i/o descritta in questa pagina. In questo modo aggiungi il supporto per l'input e/o l'output dei tuoi oggetti al sistema standard per l'i/o! Basta dichiararli così:

```
#include <iostream.h>
...
class Example
{
  friend ostream & operator << (ostream &st, Example &ex);
  friend istream & operator >> (istream &st, Example &ex);
};

ostream & operator << (ostream &st, Example &ex)
{
...
}

istream & operator >> (istream &st, Example &ex)
{
```

```
...
}
```

Per esempio consideriamo la classe Complex:

```
#include <iostream.h>
class Complex
{
  float r, i;

  friend ostream & operator << (ostream &out, Complex &ex);
  friend istream & operator >> (istream &in, Complex &ex);
};

ostream & operator << (ostream &out, Complex &ex)
{
  out << '(' << ex.r << ',' << ex.i << ')';
  return out;
}

istream & operator >> (istream &in, Complex &ex)
{
  in >> ex.r >> ex.i;
  return in;
}

int main()
{
  Complex c;
  cin >> c;
  cout << c << endl;

  return 0;
}
```

Connettere uno stream con un file

Quando vuoi connettere uno stream ad un file, usa le classi base `fstream`, `ifstream` e `ofstream`. I loro costruttori sono di questo tipo:

```
fstream (char *filename, open_mode mode);
ifstream (char *filename, open_mode mode = 0);
ofstream (char *filename, open_mode mode = 0);
```

dove `filename` è il nome del file e `mode` è un tipo enumerato. Questo enum è dichiarato nella classe `ios`, e può assumere uno o più di questi valori:

```
in          apri per l'input
out         apri in modalità di output
ate         apri e posizionati (seek) alla fine
app         apri in modo append
trunc       cancella prima di scrivere (cioè sovrascrivi)
nocreate    esci se il file non esiste
noreplace   esci se il file esiste
```

Ma vediamo un esempio:

```
// Copia un file in un altro

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void Fatal (const char *message1, const char *message2 = "")
{
  cerr << message1 << ' ' << message2 << '\n';
  exit (-1);
}

int main (int argc, char *argv[])
{
  if (argc != 3)
    Fatal ("Not enough parameters.");

  ifstream source (argv[1]);
  if (!source) Fatal ("Cannot open file : ", argv[1]);

  ofstream dest (argv[2]);
  if (!dest) Fatal ("Cannot open file : ", argv[2]);

  char c;
  while (source.get(c))
    dest.put (c);

  if (!source.eof() || dest.bad())
    Fatal ("Reading or writing error.");

  exit (0);
}
```


C++

Template

Lo scopo dei template è quello di permettere di scrivere il codice una sola volta e di utilizzarlo molte volte, in quei casi in cui si vuole lavorare con tipi in qualche modo parametrizzati. Per esempio, supponiamo che tu abbia bisogno di array di int, float o oggetti. Basta che tu scriva una classe `Array` ed usi in essa un tipo non-determinato. La sintassi è un po' complessa, ma i risultati hanno molto effetto:

```
// Dichiara la classe
template <int N, class A>
class Array
{
  A *array;
public:
  Array();
  ~Array();
  A& operator[] (int i);
};

// Dichiara il costruttore
template <int N, class A>
Array <N, A>::Array()
{
  array = new A[N];
}

// Dichiara il distruttore
template <int N, class A>
Array <N, A>::~Array()
{
  delete [] array;
}

// Dichiara la funzione membro
template <int N, class A>
A &Array <N, A>::operator[] (int i)
{
  return array[i];
}

class ExampleClass
{
public:
  int a;
};
```

```
int main (int, char **)
{
  Array <5, float> real_vector;
  Array <200, char> string;
  Array <10, ExampleClass> object_vector;

  int i;
  for (i = 0; i < 5; i++) real_vector[i] = 1.0;
  for (i = 0; i < 200; i++) string[i] = 'A';
  for (i = 0; i < 10; i++) object_vector[i].a = 0;

  return 0;
}
```

Ogni volta che dichiari un oggetto da una classe template, tutto il codice scritto (comprese le funzioni membro) deve esserci compilato nello stesso file prima della della dichiarazione dell'oggetto. Inoltre i template sono spesso usati per piccole funzioni generiche, quindi sono spesso dichiarati come funzioni inline:

```
template <int N, class A>
class Array
{
  A *array;
public:
  Array() { array = new A[N]; }
  ~Array() { delete [] array; }
  A& operator[] (int i) { return array[i]; }
};
```

questa è una definizione oltre che più semplice, più efficiente della stessa classe `Array` vista prima.

Gestione delle Eccezioni

Dichiarare un gestore delle eccezioni

Questo è un modo potente di gestire ogni tipo di errore che può accadere in un programma in modo semplice ed elegante. Basta dichiarare una o più eccezioni in una classe usando la keyword `class` in questo modo:

```
class exception_name {};
```

Poi se decidi di lanciare una eccezione usa la keyword `throw`:

```
throw exception_name();
```

In C++, si può tentare (`try`) di eseguire un pezzo di codice, e catturare (`catch`) ciascuna eccezione che può capitare durante l'esecuzione:

```
try
{
  // codice
}
catch (class_name::exception_1)
{
  // gestione dell'eccezione
}
```

Esempio

```
// Esempio di una classe Vettore con gestione delle eccezioni

#include <sys/types.h>
#include <iostream.h>
#include <stdlib.h>

class Vector
{
private:
  size_t si;
  int *value;
public:
  class BadIndex {};
  class BadSize {};
  class BadAllocation {};
```

```
  Vector (size_t);
  int& operator[](size_t);
};

Vector::Vector (size_t s)
{
  if (s <= 0) throw BadSize();
  si = s;
  value = new int[si];
  if (value == 0) throw BadAllocation();
}

int& Vector::operator[] (size_t i)
{
  if ((i < 0) || (i >= si)) throw BadIndex();
  return value[i];
}

int main (int, char **)
{
  Vector *v;

  try
  {
    int si, index;
    cout << "Give the size of the array: ";
    cin >> si;
    v = new Vector (si);
    cout << "Give an index in the array: ";
    cin >> index;
    cout << "Give its value: ";
    cin >> (*v)[index];
  }
  catch (Vector::BadSize)
  {
    cerr << "The size of an array must be greater than 0.\n";
    exit(1);
  }
  catch (Vector::BadAllocation)
  {
    cerr << "Memory allocation error.\n";
    exit(2);
  }
  catch (Vector::BadIndex)
  {
```

```
    cerr << "Index out of range.\n";
    exit(3);
  }
  //...
  exit(0);
}
```

Altro materiale...

Imparare il C

Se proprio vuoi imparare bene il C (per tuo proprio piacere, o perchè pensi che sia meglio imparare bene il C prima del C++ come credo io), dai un'occhiata alla "Programming in C" Guide di David Marshall.

Alcuni programmi utili da avere (e da studiare), gli esercizi del famosissimo Kernighan-Ritchie ed altre cose li trovate in questa mia pagina sul C.

La mia nuova homepage ufficiale è qui

Altri tutorial

Ci sono anche molti altri tutorial e pagine web sul C++ o che hanno a che fare col C++ sulla rete. Eccone alcuni...

- C++ Warriors (sito Italiano)
- Learn C/C++ Today (Una lista di risorse/tutorial)
- The C++ Virtual Library...
- ...e il suo sito FTP
- Manuel hypertexte des langages C, C++ et SyncC++ [solo in Francese]
- DOC++, The Documentation Tool.