Programmazione in C++

Guida di Stile

Marco Trivellato

trive@technologist.com
http://www.x-gdk.org/

Ultima modifica: 5.05.2001

1 Introduzione

Il C++ è un linguaggio di programmazione e, come tale, ha una grammatica e determinate regole in base alle quali devono essere scritti i programmi.
In realtà, un programma oltre alla correttezza possiede anche altre caratteristiche, quali la portabilità, la manutenibilità ed alcune che riguardano puramente l'aspetto estetico.

Questa guida non ha lo scopo di definire lo stile di programmazione in C++, bensì di proporne uno dei tanti. Ogni stile di programmazione è il frutto delle problematiche che sono state affrontate, e soprattutto, delle soluzioni che sono state elaborate. Infatti, uno stile di programmazione è in continua evoluzione e si perfeziona sia analizzando il codice di altre persone, sia grazie alla capacità di sperimentare nuove soluzioni a fronte dei problemi che si affrontano.
Questa guida non ha lo scopo di insegnare a programmare in C++ e tantomeno di mostrare al lettore nuove tecniche di programmazione. A questo proposito si assume che il lettore abbia già maturato una certa esperienza con il linguaggio.

L'obiettivo è quello di fornire una serie di regole che aiutino a scrivere codice in modo da migliorarne alcune caratteristiche fondamentali:

- Leggibilità
- Manutenibilità
- Portabilità

Le regole e i consigli che verranno presentati sono derivanti da un'esperienza personale di programmazione in C++ nel campo dei videogiochi in cui bisogna imparare a risolvere problemi di ogni genere tenendo sempre in constante considerazione le prestazioni. Nonostante ciò credo che questo documento posssa essere utile anche a chi si occupa di altre problematiche.

1.1 Motivazioni

A cosa serve definire uno stile di programmazione? Serve soprattutto a migliorare la manutenibilità del codice.
Ogni programmatore ha un suo stile di programmazione ed è molto probabile che due programmatori usino convenzioni molto diverse tra loro. Nel momento in cui si deve lavorare in gruppo è necessario definire una serie di regole comuni in base alle quali ogni programmatore deve fare riferimento. Se questo non avviene, è molto probabile che le varie parti del codice siano scritte in maniera differente, con conseguente difficoltà di lettura da parte di chi legge del codice che non ha scritto.
Non ci deve essere il concetto di "proprietà del codice" in modo che chiunque possa leggere ed eventualmente modificare ciò che, inizialmente, è stato scritto da un'altra persona.

In generale, quando si riscontra un problema nel codice, bisogna valutare se è più conveniente correggere il problema, oppure se è necessario riscrivere tutto da zero. Se dopo questa analisi si arriva alla conclusione che è più semplice riscrivere tutto il codice, allora in questo caso le cause potrebbero essere due. La prima è una cattiva progettazione, la seconda riguarda puramente il modo in cui è scritto il codice.

2 Regole Generali

La struttura di un progetto è importante soprattutto per quanto riguarda la sua manutenibilità. Il listato monolitico è assolutamente da evitare, infatti, è impensabile portare avanti un progetto di grosse dimensioni in cui tutte le dichiarazioni vengono fatte in un unico file.
Organizzando i file in piccoli moduli, che possono essere portati avanti separatamente è più facile individuare un problema e modificarlo senza comportare grosse modifiche a chi lo utilizza.
L'uso di tante piccole classi, ognuna con piccoli compiti e che non ha grosse quantità di codice è un buon metodo per modularizzare il progetto. Il risultato è sicuramente la frammentazione del codice ma è un prezzo che è compensato dai vantaggi che si ottengono se si fa uso dell'ereditarietà multipla.
Nel caso in cui si debba distribuire il progetto a terzi, oppure se si vuole raggruppare parti del progetti in *packages* è possibile creare un file che a sua volta include tutti gli altri.

2.1 Tipi di file

I nomi dei file dovrebbero essere sempre coerenti con il loro contenuto. Quindi è importante che il nome e l'estensione dei file siano significativi.

Per quanto riguarda le estensioni bisogna dire che non esiste uno standard per i file c++, infatti, alcuni compilatori come VisualC++ ed Borland C++ Builder usano di default i *.cpp*, CodeWarrior usa i *.cp*, mentre chi usa gcc molto spesso è abituato ad usare i *.cc*.
Inoltre, la distinzione codice c e c++ deve essere fatta anche per gli header quindi, anche in questo caso, bisogna utilizzare l'estensione appropriata.

A questo proposito è utili definirle per evitare che persone che lavorano allo stesso progetto utilizzino estensioni diverse tra loro.

Estensioni dei file:

.h	Header c
.hh	Header c++
.c	File c
.cc	File c++

Alcuni compilatori come I VisualC++ ed il C++ Builder non sono predisposti a compilare file .cc ma è possibile abilitarli a riconoscere questa estensione con delle apposite opzioni.

2.2 Inclusioni

Le inclusioni possono essere fatte in due modi, **“”** e **<>**. Apparentemente potrebbe sembrare che non ci sia differenza ma in realtà c'è e un giusto utilizzo potrebbe evitarvi alcuni problemi. I tag <> sono utilizzati per le librerie standard e di sistema, in genere quelle che sono fornite con il compilatore. Invece, i doppi apici “” per tutti gli altri file, compresi quelli del progetto a cui state lavorando. Alcuni compilatori non fanno distinzione, altri permettono di cercare i file tra parentesi angolari nei percorsi definiti dall'utente. Per evitare problemi dovuti all'utilizzo di un altro compilatore non utilizzate mai questa opzione.

L'uso della direttiva di preprocessore *#include* deve essere fatto con un certo criterio. Nel senso che bisogna includere il minor numero di file in modo da diminuire le dipendenze tra i file.
Innanzitutto, il *.hh* di una classe deve includere solo i file delle classi da cui deriva, il *.cc* invece include tutti i file che riguardano i tipi di cui ha bisogno. Nel caso in cui una classe utilizzi dei tipi di dato che non sono definiti, ovviamente il compilatore genere un errore del tipo '*undefined symbol*' o '*declaration error'*.
In questi casi, la tentazione di includere tutti i file di cui si ha bisogno direttamente nel .h è sicuramente molto forte, soprattutto per chi è alle prime armi con il C++ ma si deve cercare di sostituire queste inclusioni con delle dichiarazioni anticipate *(pre-declarations)*.
Vediamo un esempio:

```
#include “bitmap.hh”

class Button{
public:
      Bitmap*     GetBitmap(  );
      void        SetBitmap(  Bitmap* bitmap);
};
```

grazie all'inclusione di bitmap.hh, il compilatore è in grado di riconoscere il simbolo Bitmap ma questa soluzione aumenta le dipendenze tra i file e puó essere facilmente sostituita con la seguente:

```
class Bitmap; // dichiarazione anticipata

class Button{
public:
      Bitmap*     GetBitmap(  );
      void        SetBitmap(  Bitmap* bitmap);
};
```

Questo tipo di approccio comporta una diminuzione delle dipendenze e di conseguenza diminuisce anche il tempo necessario a ricompilare a seguito di una modifica che riguarda questi file.

2.3 File Header

Per evitare problemi di doppia inclusione, ogni header dovrebbe avere questa struttura:

```
#ifndef MYHEADER_H
#define MYHEADER_H
#pragma once

// dichiarazioni

#endif      // MYHEADER_H
```

La direttiva *once* serve ad indicare che il file deve essere indicato una sola volta. Quindi si puó dire che ha la stessa funzione dell'*#ifndef*. La differenza sta nel fatto che un metodo definisce un simbolo di preprocessing e l'altro è un settaggio del compilatore. Usarli insieme serve ad avere la certezza che il file non venga incluso piú di una volta.

2.4 Dichiarazioni

I programmatori C sono abituati a dichiarare tutte le variabili prima di tutte le altre istruzioni. In C++ non vi è più questa restrizione, inoltre, dichiarare una variabile al momento del primo utilizzo risulta più efficiente. Ciò è dovuto al fatto che la dichiarazione di un oggetto implica che il suo costruttore venga eseguito automaticamente.

2.5 Puntatori e References

Puntatori e reference devono essere dichiarati rispettivamente con '*' e '&' a fianco del tipo, oppure accanto al nome. E' preferibile la prima soluzione.

```
Button*    pCancel;
```

oppure

```
Button    *pCancel;
```

Nelle dichiarazioni multiple, questo tipo di dichiarazione può indurre in errore:

```
Button*    pCancel, pOk;      // pOk non è un puntatore!
```

allora è utile essere più espliciti, dichiarandone uno alla volta:

```
Button*     pCancel;
Button*     pOk;
```

questa soluzione risolve qualsiasi ambiguità ma ad alcuni potrebbe non piacere, in tal caso è possibile definire un tipo di dato da usare come puntatore:

```
class Button{
      ...
};

typedef Button*   ButtonPtr;
```

in questo modo la dichiarazione diventa:

```
ButtonPtr   pCancel,    pOk;
```

3 Documentazione

3.1 Commenti

Anche per quanto riguarda i commenti non esistono regole precise ma seguire delle convenzioni aumenta la leggibilità del codice ed aiuta il lettore a ricavarne informazioni che, altrimenti, potrebbe avere solo nel caso in cui avesse la possibilità di chiederle direttamente all'autore.
I commenti sono essenziali e possono essere considerati come parte integrante del codice.

All'inizio di ogni file ci dovrebbe sempre essere un'intestazione in cui ci sono informazioni riguardanti:

- contenuto del file
- autore o autori
- versione
- data di creazione e di ultima modifica
- note sull'utilizzo

eventualmente è possibile inserire una "storia" del file in cui vengono indicate le modifiche che sono state effettuate.

I commenti al codice devono essere sintetici e significativi. Non devono comparire a fianco di ogni istruzione, come accade per listati in assembler ma dovrebbero spiegare in modo essenziale ciò che l'autore vuole ottenere da un certo numero di istruzioni senza duplicare informazioni, infatti, i commenti devono necessariamente spiegare qualcosa che non si può capire direttamente dal codice.

Commenti del tipo:

```
int index = 5; // inizializzo index
index *= 5; // moltiplico index per 5
```

sono superflui e rischiano solo di confondere chi analizza il codice.

I commenti relativi ad una classe devono spiegare che cosa rappresenta e quali sono le sue funzionalità. Ulteriori spiegazioni riguardano i singoli metodi, anche se, in molti casi, potrebbe essere sufficiente che i loro nomi e quelli dei parametri siano abbastanza significativi.
Per alcuni programmatori, questi consigli potrebbero risultare scontati ma personalmente credo che per altri non lo siano affatto.

3.2 Generazione automatica

Ecco un esempio di intestazione con sintassi per la generazione automatica della documentazione tramite Doxygen:

```
/*!
 *  \class        Base
 *
 *  \brief        base class containing name, object and class information
 *
 *  \author       Marco Trivellato
 *
 *  \version      1.0
 *
 *  \date         05.dec.1999 - 11.jan.2001
 *
 *  \internal
 *
 *  Bugs?:        Please send this file with the problem
 *                description to trive@technologist.com
 */
```

4 Nomi

In questa guida vengono usati i seguenti prefissi:

m	membri di classi
s	membri statici
p	Puntatori
h	Handle
g	variabili globali

Ogni parola che segue la prima, deve iniziare con la lettera maiuscola.

```
class IndexBuffer{
public:
            IndexBuffer( int indexCount)
              :  mIndexCount( indexCount)
            {
                  ...
            }

      int   GetIndexCount() const { return mIndexCount; }

            ...
```

```
private:
      int   mIndexCount;
            ...
};
```

4.1 Criteri di assegnazione

Assegnare un nome ad una variabile, ad un metodo o a una classe è una delle prime operazione che un programmatore deve fare quando scrive le sue prime linee di codice. Spesso accade che questa operazione venga fatta in modo superficiale, ottenendo così dei nomi poco significativi oppure, nel peggiore dei casi, non adatti.
Sono da evitare dichiarazioni del tipo:

```
EditField edit1, edit2, edit3;

Button* p = new Button();

Button* pippo = new Button();
```

questi nomi non danno alcuna informazione aggiunta al loro tipo e rischiano di confondere anche l'autore stesso.

I nomi devono essere significativi e bisogna convincersi che il tempo speso per trovare un nome deve essere considerato come tempo risparmiato durante la manutenzione del codice. Il nome perfetto non esiste ma esistono solo nomi che calzano meglio e che quindi sono più significativi. Non stupitevi di cambiare un nome anche dopo alcuni mesi, se vi accorgete che un determinato nome non vi soddisfa e ne avete trovato uno che si adatta meglio, allora dovete cambiarlo.

I nomi delle classi identificano un'entità o un particolare comportamento (*behavior*) che può assumerne un determinato oggetto.

4.2 Variabili

Esistono diverse scuole di pensiero sui nomi da dare ad un membro di una classe o ad una semplice variabile dichiarata nello scope di un metodo. Una convenzione molto diffusa nell'ambiente Windows che è assolutamente da evitare è l'Hungarian-notation, in base alla quale ogni nome di variabile deve essere preceduto da una serie di caratteri che ne identificano il tipo. In molti caso questi nomi diventano molto lunghi e il tempo che bisogna spendere per dichiarare una variabile non è giustificato dal vantaggio di sapere in qualsiasi momento il suo tipo, soprattutto adesso che queste informazioni sono fornite direttamente dall'ambiente di sviluppo.

I nomi delle variabili devono iniziare con la lettera minuscola.

…

4.3 Puntatori

L'unico postfisso riguarda i tipi dei puntatori, i quali devono essere seguiti da *Ptr*:

```
class Bitmap{
      ...
};

typedef Bitmap*   BitmapPtr;
```

4.4 Costanti

Invece, le costanti vengono precedute dalla lettera *k* minuscola:

```
enum{
      kOpenMode_Read          = 0x01,
      kOpenMode_Write         = 0x02
};
```

…

4.5 Funzioni

Per quanto riguarda i nomi delle funzioni vi sono due convenzioni particolarmente diffuse che differenziano solo per la lettera iniziale. Una che ”vuole” la prima lettera minuscola, come ad esempio la convenzione Java, ed l’altra in cui è maiuscola come nel caso delle API di Windows.
Noi useremo la seconda convenzione, quindi le procedure avranno nomi di questo tipo:

```
Image*      CreateImage( int width, int height, int depth);
```

4.6 Classi

I nomi delle classi iniziano tutti con una lettera maiuscola. Alternativamente possono essere preceduti da un prefisso di comune a tutto il progetto, ad esempio Sprite potrebbe diventare GfxSprite. In genere il prefisso non dovrebbe superare le 3 lettere.

```
class GfxBitmap{
      ...
};

class GfxSprite{
      ...
};
```

E’ comunque preferibile inserire tutte le classi in un namespace.

```
namespace Gfx{

class Bitmap{
      ...
};

class Sprite{
      ...
};

}
```

Le variabili membro devono essere preceduti dalla lettera *m*, senza l'underscore a seguire. Ad esempio:

```
class Bitmap{
private:
      int   mWidth;
      int   mHeight;
};
```

4.7 Metodi

Per i metodi di una classe valgono le stesse regole delle funzioni.

```
class Bitmap{
public:
      ...
      int   GetWidth();
      int   GetHeight();

      void  SetWidth( int width);
      void  SetHeight( int height);
private:
      ...
};
```

4.8 Parametri

Il C++ permette di omettere il nome dei parametri in ingresso dei metodi di una classe, definendone solo il tipo. Ecco un esempio:

```
class List{
public:
      void  Insert(Node*);
};
```

E' invece preferibile utilizzare sempre la dichiarazione estesa, indicando il nome del parametro. Dare un nome significativo ai dati in ingresso può essere considerato parte della documentazione.

```
class List{
public:
      void  Insert(Node* pItem);
};
```

5 Formattazione

5.1 Indentazione

Per migliorare la leggibilità del codice, è necessario scrivere in modo che il codice possa essere letto senza troppi problemi nell'individuare l'inizio o la fine di un costrutto. Ogni riga non deve avere più di una istruzione.

5.2 Parentesi graffe

Per quanto riguarda le parentesi graffe ci sono principalmente due correnti di pensiero. Una molto diffusa tra i programmatori per sistemi Linux:

```
if (cond) {
    istruzione1
    istruzione2
}
```

una variante è quella in cui la parentesi aperta si trova allineata con la parentesi chiusa:

```
if (cond)
{
    istruzione1
    istruzione2
}
```

Lo stile di programmazione descritto in questa guida utilizza la prima convenzione.
Per tutte le persone che non condividono questa scelta, il consiglio è comunque di non scrivere codice come questo:

```
if (cond)
      {
      istruzione1
      istruzione2
      }

istruzione1
      if (cond)
      {
      istruzione2
      istruzione3
      }
```

ovviamente questo è un esempio molto semplice ma puó capitare di avere dei costrutti annidati in cui diventa difficile capire dove sia l'inizio e la fine dei vari scope.

5.3 Uso di spazi e Tab

Per indentare il codice non bisogna usare gli spazi, bensì il tab preferibilmente da 4 caratteri.
Gli spazi, invece, devono essere utilizzati altrove per migliorare la leggibilità.

- Passaggio di parametri
- assegnamenti

…

5.4 Linee lunghe

In generale non esiste una lunghezza massima ma esistono solo linee che possono risultare di difficile lettura, perciò devono essere scritte su più righe. Ecco alcuni esempi su come riscrivere alcuni costrutti:

```
for (vector<Node>::iterator iter( nodes.begin()); iter != nodes.end(); iter++){
    iter->DoSomething();
}
```

diventa:

```
for (vector<Node>::iterator iter( nodes.begin());
        iter != nodes.end();
        iter++){
    iter->DoSomething();
}
```

5.5 Condizioni

Le condizioni devono essere chiare, soprattutto per quanto riguarda i puntatori.

Ad esempio, volendo rendere più esplicita la condizione:

```
if ( pButton ) {

      pButton->Draw();
}
```

si potrebbe scrivere come:

```
if ( pButton != nil) {

      pButton->Draw();
}
```

come si può notare, questo accorgimento comporta una piccola modifica al codice ma che sicuramente rende più immediato il suo significato.
E volendo eliminare ogni pericolo dovuto ad assegnamenti non voluti è possibile scrivere:

```
if ( nil != pButton) {

      pButton->Draw();
}
```

Dove necessario, i commenti devono essere scritte su più di una riga. Ad esempio:

```
if ( cond1 && cond2 && cond3) {

    istruzione1;
    istruzione2;
}
```

diventa:

```
if ( cond1 &&
        cond2 &&
        cond3) {
```

```
    istruzione1;
    istruzione2;
}
```

5.5 if-then-else

L'if-then-else è uno dei costrutti più utilizzati, perciò anche in questo caso bisogna uniformare il modo in cui viene scritto.
Ecco come dovrebbe essere scritto in queste diverse situazioni:

```
if (condizione)
   istruzione
```

oppure

```
if (condizione) {
   …
}

if (condizione) {
    …
}
else {
    …
}
```

In situazioni in cui compiono più *if* annidati è preferibile l'uso delle parentesi graffe anche se vi è una sola istruzione da eseguire. Ad esempio:

```
if (cond)
    if (cond)
        istr1;
    else
        istr2;
```

il ramo else è riferito al primo o al secondo if? Per il compilatore si riferisce al secondo e probabilmente è proprio quello che vuole ottenere chi lo ha scritto ma è molto facile incorrere in errori che non sempre si individuano facilmente.

Riscrivendolo potrebbe diventare:

```
if (cond) {
    if (cond) {
        istr1;
    }
    else {
        istr2;
    }
}
```

In questo modo risulta molto più chiaro e non lascia spazio ad ambiguità.

Infine sono da evitare indentazioni di questo tipo:

```
if (condizione) istruzione
```

Risparmiare una riga non serve assolutamente a nulla e rende il codice di difficile lettura, soprattutto quando questo tipo di indentazione viene ripetuto più di una volta.
Un altro motivo per non scrivere l'*if* su una sola riga è la fase di debug, infatti, risulta molto scomodo usare dei *breakpoint* scrivendo in questo modo.

5.5 for

Consideriamo questo ciclo for:

```
for (int i = 0; i < 5; i++){
      ....
      if (i % 2){
            cout << “dispari” << endl;
            }
      else
            cout << “pari” << endl;
      }
```

potrebbe diventare:

```
for (int i = 0; i < 5; i++) {
      ...
      if (i % 2) {
            cout << “dispari” << endl;
      }
      else {
            cout << “pari” << endl;
      }
}
```

...

5.6 while

In generale è preferibile usare il ciclo *while* piuttosto che il *do_while*.

Cicli del tipo:

```
while (value++ < N);
```

oppure

```
while (value++ < N)
{
}
```

sono equivalenti ma nel caso in cui si voglia usare il primo tipo è consigliabile inserire un commento.

5.7 switch

Quando possibile, è sempre meglio usare lo *switch* piuttosto che una serie di *if* che testano lo stesso dato.

```
switch (value) {
	case kRed :
		...
		break;

	case kGreen:
		...
		break;

	case kBlue:
		...
		break;

	default :
		// Unknown value
		break;
}
```

Ora vediamo un altro esempio:

```
switch (value) {
	case kRed :
		...
		break;

	case kGreen:
		...

	case kBlue:
		...
		break;

	default :
		// Unknown value
		break;
}
```

Come si può notare, nel case kGreen manca il break. Ciò è perfettamente lecito ma sarebbe più chiaro se venire inserito un commento per esplicitare che l'esecuzione continua dalla prima istruzione del case kBlue:

```
switch (value) {
	case kRed :
		...
		break;

	case kGreen:
		...
		// continua

	case kBlue:
		...
		break;

	default :
		// Unknown value
		break;
}
```

L’ultimo consiglio per lo switch è quello di inserire sempre il caso *default*. In questo modo si rende il codice più esplicito.

6 Classi

Ogni classe del progetto è associata a due file: il .hh che contiene la dichiarazione ed il .cc l’implementazione.
A questa regola fanno eccezione le classi template, le quali non hanno un .cc ma solo il .hh.
Infatti, il corpo dei metodi si trova dopo la dichiarazione della classe stessa:

```
#ifndef VECTOR_H
#define VECTOR_H

template<class T>
class Vector{
public:
            Vector( unsigned int size);
      T&    operator[] ( unsigned int index);
            ...
protected:
      T*          mpItems;
unsigned int      mSize;
                  ...
};

template<class T>
T&
Vector<T>::Vector( unsigned int size)
  :  mSize( size)
{
      ...
}

template<class T>
T&
Vector<T>::operator[]( unsigned int index)
{
      return mItems[index];
}

#endif      // VECTOR_H
```

I membri di una classe dovrebbero essere sempre dichiarati con questa sequenza:

```
class MyClass{
public:
      ...
protected:
      ...
private:
      ...

}
```

Esempio di dichiarazione di una classe derivata:

```
#ifndef MYBITMAP_H
#define MYBITMAP_H


#include “Bitmap.hh”

namespace gfx{

class MyBitmap : public Bitmap {
public:
                  MyBitmap(   unsigned int width,
                              unsigned int height)
                    :  Bitmap( width,
                              height)
                  {
                  }

      virtual     ~MyBitmap(){            // distruttore virtuale
                  }

};    // fine della dichiarazione della classe Bitmap

}     // fine del namespace

#endif      // MYBITMAP_H
```

6.1 Costruttori

Ogni classe dovrebbe fornire un costruttore di default. Se ciò non avviene è il compilatore che si preoccupa di fornirlo ma in questo caso il codice potrebbe risultare poco chiaro e si dovrebbe modificare la dichiarazione della classe nel momento in cui si decidesse di implementarlo.
Un costruttore dovrebbe solo inizializzare i membri dell’oggetto senza eseguire altre operazioni che potrebbero generare eccezioni.
E’ importante che ogni membro venga inizializzato in modo da evitare qualsiasi ambiguità sul suo valore, soprattutto se si tratta di un puntatore. Questa é una regola che, se rispettata, potrebbe risparmiare parecchio tempo in fase di debugging. Infatti non vi è alcuna garanzia che il compilatore inizializzi a zero l’area di memoria in cui viene allocato un oggetto. Ad esempio, un puntatore non inizializzato potrebbe contenere un indirizzo che non corrisponde ad alcun oggetto in memoria, oppure si riferisce ad un oggetto che è giá stato deallocato.
Inoltre, si deve evitare di usare opzioni del compilatore che azzerano automaticamente i membri di un oggetto. E’ molto meglio che questa operazione sia resa esplicita all’interno del costruttore.
L’unico caso in cui è possibile omettere l’inizializzazione è quando esiste un costruttore di default.

Costruttori del tipo:

```
Bitmap::Bitmap(int width,
               int height)
{
      mWidth = width;
      mHeight = height;
}
```

dovrebbero diventare:

```
Bitmap::Bitmap(int width,
               int height) :
      mWidth( width),
      mHeight( height)
{
}
```

Attenzione a non chiamare metodi virtuali all'interno di un costruttore.

```
class Bitmap{
public:
      ...
      virtual     Build();

protected:
      ...
};

class Texture : public Bitmap{
public:
      ...
      virtual     Build();
protected:
      ...
};

Bitmap::Bitmap(int width,
               int height)
  :   mWidth( width),
      mHeight( height)
{
      Build();
}
```

Durante la creazione di un oggetto di tipo Texture viene eseguito metodo Bitmap::Build() e non Texture::Build(), infatti, all'interno di costruttori ogni meccanismo virtuale è 'disabilitato' per evitare che un metodo faccia riferimento a membri della classe derivata che non sono ancora stati inizializzati.

6.2 Distruttori

I distruttori devono essere dichiarati con il specificatore *virtual*. In questo modo abbiamo la sicurezza l'oggetto che l'oggetto sia distrutto correttamente anche se viene distrutto da un puntatore alla classe base.

Consideriamo il seguente caso:

```
class Base {
public:
            Base();
            ~Base();
};

class Derived : public Base{
```

```
public:
            Derived ();
            ~ Derived ();
};

Base* pBase = new Derived;

delete pBase;     // viene chiamato il distruttore di Base
```

Per garantire che venga invocato anche il distruttore di Derived è necessario utilizzare lo specificatore *virtual*:

```
class Base {
public:
                  Base();
      virtual     ~Base();
};

class Derived : public Base {
public:
                  Derived ();
                  ~ Derived ();
};

Base* pBase = new Derived;

delete pBase;     // viene chiamato il distruttore di Derived e successivamente
                  // il distruttore di Base
```

Infine, anche all'interno dei distruttori non dovrebbero comparire chiamate a metodi virtuali.

6.3 Accesso ai dati

Una delle peculiarità piú importanti delle classi è sicuramente l'incapsulamento dei dati. I membri di una classe dovrebbero sempre essere dichiarati *private* o *protected*.
Quindi, la cosa migliore da fare é creare un'interfaccia, ovvero, serie di metodi del tipo *Get/Set* che regoli l'accesso ai membri sia per la lettura, che per la modifica.
Molto spesso, chi arriva dal C si chiede quale sia l'utilità di dichiarare dei membri privati o protected. Questo tipo genere di approccio permette di aver maggiore controllo e sicurezza sia per quanto riguarda i diritti di accesso, sia per garantire la coerenza dei dati.
Ecco una dichiarazione che, oltre a non avere nulla di C++ a parte la parola *class*, non garantisce l'integrità dei dati e tantomeno la loro validità:

```
class Bitmap{
public:
      int   mWidth,
            mHeight;
      int   mSize;
};
```

Questa classe non fornisce alcuna garanzia sulla validità dei suoi membri, in particolare di mSize. Una soluzione più sicura potrebbe essere la seguente:

```
class Bitmap{
public:
      int   GetSize();
```

```
private:
      int   mWidth,
            mHeight;
      int   mSize;
};
```

Il metodo GetSize() restituisce la dimensione della Bitmap, l'utente non sa esattamente se la sua implementazione è:

```
int
Bitmap::GetSize()
{
      return mSize;
}
```

oppure:

```
int
Bitmap::GetSize()
{
      return mWidth*mHeigh;
}
```

I due metodi sono equivalenti ma se mSize fosse dichiarato *public*, chiunque potrebbe accedere e modificare il suo valore. Inoltre, se non ci fosse il metodo GetSize() e si decidesse di calcolare il valore invece di tenerlo come membro, ció richiederebbe una modifica all'interfaccia della classe, con conseguente modifica del codice da parte dell'utente.
Inoltre, utilizzando dei metodi per richiedere informazioni relative ad un oggetto, si nasconde l'implementazione in modo del tutto trasparente dal punto di vista dell'utilizzatore, al quale non interessa come viene svolta una determinata operazione ma solo che la esegua correttamente.

6.4 Specificatore *const*

Quando possibile bisogna sempre utilizzare lo specificatore *const*.
Nei casi in cui il metodo della classe non effettua modifiche ai membri dell'oggetto dovrebbe essere dichiarato come *const*.

Consideriamo questa dichiarazione:

```
class Bitmap{
public:
      int   GetSize();
protected:
      int   mSize;
};
```

Il metodo *GetSize* non effettua alcuna modifica ma non vi è alcuna garanzia che nella sua implementazione non vengano fatte modifiche ai membri della classe.

```
class Bitmap{
public:
      int   GetSize() const;
```

```
protected:
      int   mSize;
};
```

Se il metodo è dichiarato come *const* è il compilatore che si preoccupa di verificare che non vengano modificati i membri della classe.

Ad esempio:

```
class Bitmap{
public:
      int   GetSize() const {
                mSize = mWidth*mHeight;   // assegnazione non permessa
                return mSize;
            }
protected:
      int   mWidth;
      int   mHeight;
      int   mSize;
};
```

in questo caso, l'errore è segnalato a *compile-time*.

6.5 Membri statici

I membri statici possono essere la soluzione per molti problemi ma attenzione a farne un uso eccessivo.
Devono essere usati principalmente per condividere dati tra oggetti della stessa classe e non semplicemente come un metodo alternativo per dichiarare variabili globali!
Inoltre, devono essere limitati i casi in cui un oggetto deve accedere ai membri statici di un'altra classe. Se ció avviene molto frequente, cominciate a chiedervi se quei membri non debbano essere dichiarati dentro alla classe che li usa.

Non bisogna mai fare assunzioni sull'ordine di inizializzazione dei membri statici. Nel caso in cui non si possa fare altrimenti è possibile verificare se il compilatore permette di definire una propria procedura in cui vengono fatte le inizializzazioni dei membri statici. Non tutti i compilatori supportano questa opzione.

6.6 Overloading degli operatori

Chi viene a conoscenza per la prima volta di questa caratteristica del linguaggio, spesso viene preso dalla voglia di utilizzare gli operatori in sostituzione dei metodi tradizionali. In effetti, con gli operatori si possono scrivere espressioni complesse ma può accadere che questo vada a scapito della leggibilità.
A questo proposito non ci sono regole particolari ma un solo consiglio: l'overloading degli operatori deve essere utilizzato per migliorare la leggibilità non il contrario. Se quello che si ottiene è un codice di difficile interpretazione allora non conviene ridefinire gli operatori.

7 Portabilità

La portabilitá è una caratteristica del codice che spesso non viene considerata nelle fasi iniziali di progettazioni o che, comunque, viene tralasciata per risparmiare tempo.
Il problema è che, spesso, il tempo risparmiato inizialmente viene perso quando si decide di dare il porting per un'altra piattaforma. Bisogna considerare che è molto piú costoso fare il porting in fase avanzata, piuttosto di progettare il codice fin dall'inizio in modo da poterlo convertire facilmente per un'altra piattaforma.
Lo scopo di questa guida non è sicuramente di spiegare in dettaglio le questioni riguardanti la portabilità ma vediamo alcuni semplici consigli che possono far risparmiare molto tempo in fase avanzata di progetto.

7.1 Uso del typedef

Mettere a disposizione una serie di tipi di base che chiunque potrà utilizzare è sicuramente un metodo per rendere il codice meno indipendente dalla piattaforma:

```
typedef     signed char       SInt8;
typedef     signed short      SInt16;
typedef     signed long       SInt32;

typedef     unsigned char     UInt8;
typedef     unsigned short    UInt16;
typedef     unsigned long     UInt32;

typedef     SInt8*            SInt8Ptr;
typedef     SInt16*           SInt16Ptr;
typedef     SInt32*           SInt32Ptr;

typedef     unsigned char     Boolean;
```

in questo modo è possibile effettuare modiche ai tipi di base senza dover cambiare tutto il codice che li utilizza.
Ad esempio, se volessimo definire un tipo Real come:

```
typedef float     Real;
```

in qualsiasi momento potremmo decidere di cambiarlo in:

```
typedef double    Real;
```

senza effettuare altre modifiche al codice, evitando di fare un find & replace in tutti i sorgenti.

7.2 #define condizionali

Le definizioni condizionali sono utili a scegliere alcune proprietà che influenzano l'intera compilazione. Infatti, quando si scrive codice portabile si devono modificare le parti di codice che dipendono dalla piattaforma ed è molto utile poter utile definire alcuni parametri che dipendono da:

- processore
- sistema operativo
- compilatore

Ad esempio, è possibile definire alcuni tipi di dato in precedenza in base al compilatore:

```
#if !defined(__BORLANDC__) && !defined(__BCPLUSPLUS__)

      typedef unsigned char         Boolean;

      enum {
            False = 0,
            True  = 1
      };

#endif
```

in questo caso si evita che il tipo *Boolean* venga definito se il compilatore lo fornisce giá nelle sue librerie.

8 Ottimizzazioni

L'ottimizzazione merita un discorso particolare: è vero che le prestazioni sono da tenere in costante considerazione, soprattutto nel campo dei videogiochi, ma è anche vero che se l'ottimizzazione viene fatta nella fase iniziale dello sviluppo, il codice diventa difficile da modificare, a volte anche per chi lo ha scritto.
Quindi, il codice dovrebbe essere scritto in modo da poter essere eseguito molto velocemente e, allo stesso tempo dovrebbe essere facilmente manutenibile. Per questo motivo, l'ottimizzazione deve essere fatta necessariamente alla fine, possibilmente commentando il codice dando una spiegazione dell'ottimizzazione. Inoltre è fortemente consigliato ottimizzare solo quando la porzione di codice interessata è stata giá testata.

8.1 Assembler

Codice assembler dovrebbe essere evitato, sia per quanto riguarda la portabilità, sia perché non è sicuro che riscrivendo una sequenza di istruzioni da C/C++ in Assembler si ottenga un miglioramento di prestazioni. In ogni caso, bisogna tenere sempre presente che difficilmente riusciremo ad ottimizzare il codice meglio del compilatore che, nella maggior parte dei casi ha delle opzioni specifiche riguardanti l'ottimizzazione.

Quindi, il consiglio è quello di fare ottimizzazioni di "alto livello", lasciando fare le altre al compilatore. Ad esempio, dovendo velocizzare le prestazioni di gioco con visualizzazione in 3d è preferibile "lavorare" sugli algoritmi per determinare quali sono gli oggetti visibili, piuttosto che scrivere in assembler altre porzioni di codice.

In generale, questi consigli vanno molto bene per chi sviluppa per PC e devono invece essere rivisti per quanto riguarda le console e sistemi embedded.

8.2 Inlining

Il c++ ha delle caratteristiche che possono essere utili per rendere il codice più performante. La più utilizzata è sicuramente l'*inlining*, la seconda invece sono i *template.*
L'inlining può essere considerato come un'evoluzione delle macro ma con dei vantaggi molto importanti. Il primo è quello di non avere *function-call overhead*, quindi di garantire le stesse prestazioni delle macro e il secondo è il fatto di essere *type-safe*.

Anche per quanto riguarda queste particolarità del linguaggio bisogna fare attenzione a non usarlo dove non è necessario, infatti, è possibile che del codice *inlined* o *unrolled* risulti meno performante. Ciò è dovuto al meccanismo di caching delle porzioni di codice eseguite più frequentemente dal processore. Il codice in versione *unrolled* occupa più spazio nella cache del processore, mentre quello in versione compatta potrebbe stare tutto nella cache senza dover richiedere continuamente che venga sostituito. A questo proposito è consigliabile l'utillizzo di un profiler per verificare se, nel caso specifico, è conveniente usare l'inlining.

8.3 Template

Quando è possibile, conviene sempre usare classi e funzioni template.
I *template* permettono di scrivere codice molto generico in modo da garantire un alto grado di riutilizzabilità. Infatti, l'utilizzo di librerie come l'stl (standard template library), oltre che ad essere un buon metodo per ingegnerizzare il codice è utile per ottimizzarlo, infatti, il codice di un qualsiasi contenitore *stl* viene automaticamente inserito inline.
Un lato negativo dell'uso dei template potrebbe essere la fare di debug. Nel caso in cui il codice risulti molto complesso il consiglio è quello di scrivere le proprie classi senza utilizzarli e di "templetizzarle" solo successivamente quando il codice è più definitivo.
I template sono una ultime caratteristica del linguaggio che sono entrate a far parte dello standard. Per questo motivo, non tutti i compilatori garantiscono lo stesso supporto ai template.

9 Varie

9.1 Costanti

Le costanti devono essere utilizzate in sostituzione dei valori numerici.

Ad esempio, uno switch di questo tipo:

```
switch (type) {
      case 0 :
            ...
            break;
      case 1 :
            ...
            break;
      case 2 :
            ...
            break;
      default :
            cout << "Unknown value" << endl;
            break;
}
```

è di difficile interpretazione ma potrebbe diventare piú chiaro riscritto in questo modo:

```
switch (value) {
      case kButton :
            ...
            break;
```

```
      case kPicture :
            ...
            break;

      case kEditField :
            ...
            break;

      default :
            cout << “Unknown value” << endl;
            break;
}
```

9.2 Direttiva di preprocessore #define

Devono esistere solo per quanto riguarda la compilazione condizionale.
L’uso più comune della direttiva #define riguarda la definizione di costanti:

```
#define NETWORK_NODE          0
#define ELECTRICAL_NODE       1
#define ROAD_NODE             2
#define OBJECT_NODE           3
#define BUILDING_NODE         4
```

questo tipo di dichiarazioni devono essere evitate e si potrebbero scrivere come:

```
const int kNetworkNode        =     0;
const int kElectricalNode     =     1;
const int kRoadNode           =     2;
const int kObjectNode         =     3;
const int kBuildingNode       =     4;
```

anche se la soluzione migliore è sicuramente quella che fa uso degli *enum*:

```
enum NodeType{
    kNetworkNode        = 0,
    kElectricalNode,    // = 1
    kRoadNode,          // = 2
    kObjectNode,        // = 3
    kBuildingNode       // = 4
};
```

In questo modo il programmatore non si deve più preoccupare dell’univocità delle costanti, infatti, questo compito è demandato al compilatore. Inoltre, è possibile utilizzare il tipo *NodeType* per il passaggio di parametri ai metodi o come valore di ritorno, evitando errori a tempo di compilazione.

La definizione di costanti è uno dei due principali utilizzi del #define. Il secondo riguarda le macro.
Ad esempio, dichiarazioni del tipo:

```
#define FOR_EACH_ITEM( l)     for( list<int>::iterator iter(l.begin()); \
                                    iter != l.end(); iter++)

FOR_EACH_ITEM(myList){
      // do something
      ...
```

```
}
```

Devono essere assolutamente evitate, infatti, il codice potrebbe risultare poco chiaro e soprattutto lo rende difficilmente debuggabile.
Anche dichiarazioni come:

```
#define abs(x)    ( ((x) < 0) ? -(x) : (x) )
```

non devono comparire ma devono essere sostituite da funzioni *inline*:

```
inline int abs( int x) {
      return x < 0 ? -x : x;
}
```

oppure, nel caso in cui non si voglia limitare la funzione agli *int*, si possono usare i template:

```
template<class T>
T abs(const T& x) {
      return x < 0 ? -x : x;
}
```

In C++ non ha più senso usare le #define per motivi prestazionali, pertanto devono essere evitate.

9.3 new e delete

Nel caso in cui la dimensione di un array sia conosciuto a compile-time e sia la new che delete vengono chiamate nello stesso scope, allora è preferibile allocare l'array nello stack.

situazioni come:

```
{
    char* temp = new char[10];
    … // do something
    delete [] temp;
}
```

possono essere sostituite con:

```
{
    char temp [10];     //
    … // do something

}
```

in questo caso, temp viene deallocato automaticamente al termine dello *scope*.
Vi sono due motivi che rendono preferibile questa soluzione e riguardano entrambe la deallocazione: il primo è che questa operazione è demandata al compilatore e quindi non si rischiano *memory-leak*. La seconda è che richiede solo una modifica allo *stack-pointer*, senza richiedere che venga eseguita alcune routine di allocazione.

9.4 Passaggio di Parametri

Quando è possibile è consigliabile utilizzare i *reference*, in questo modo l'utilizzo si evita l'utilizzo, da parte del compilatore, di variabili temporanee che vengono allocate in occasione dell'esecuzione del metodo.

9.5 Valori di ritorno

Stesso discorso per i valori di ritorno di una funzione. Anche in questo caso, utilizzando i reference, si evita la costruzione di un oggetto temporaneo che viene poi immediatamente distrutto.

9.6 Namespace

La dichiarazione di namespace non deve avere sintassi particolari, l'unico accorgimento è quello di aggiungere un commento per rendere più esplicita la loro conclusione:

```
namespace gfx {

      // my declarations/implementations
      // ...
      // ...

} // end of gfx namespace
```

9.7 Cast

Il cast dovrebbe essere limitato il più possibile e nei casi in cui non se ne può fare a meno bisogna usare gli operatori di cast forniti dal C++:

- *static_cast*
- *dynamic_cast*
- *reinterpret_cast*
- *const_cast*

Lo static_cast equivale al classico cast del linguaggio c è deve essere usato ogni qual volta il tipo tra parentesi angolari si conosce a compile-time. Attenzione però, codice in cui sono presenti molti cast, potrebbe essere sintomo di un errore in fase di design. Il cast non è affatto indispensabile è nella maggior parte dei casi può essere evitato. Personalmente lo considero come un metodo provvisorio per accedere ad una serie di dati in modo veloce.

9.8 RTTI

L'RTTI (Run-Time-Type-Identification) è una della caratteristiche più potenti del C++, ma attenzione a non abusarne, infatti, un *dynamic_cast* viene risolto a tempo di esecuzione e, per questo motivo, va a scapito della prestazioni.
Un metodo alternativo è quello di assegnare ad ogni classe un identificativo univoco di 4 caratteri:

```
class Window{
public:
      enum { kClassId = ‘wndw’ };

                        Window(...);
      virtual           ~Window();

      virtual UInt32    GetClassId() const {
                              return kClassId;
                        }
      ...
protected:
      ...
};

class Dialog : public Window {
public:
      enum { kClassId = ‘dlg ’ };

                        Dialog(...);
      virtual           ~ Dialog();

      virtual UInt32    GetClassId() const {
                              return kClassId;
                        }
      ...
protected:
      ...
};
```

In questo modo è possibile identificare a *run-time* un oggetto di una classe derivata da Window:

```
Window* pWnd = new Dialog(…);

if (pWnd->GetClassId() == Dialog::kClassId)
      cout << “This object is a Dialog” << endl;
else
      cout << “This object is a Window” << endl;
```

In questo caso il risultato sarà: “This object is a Dialog”

9.9 Puntatori a funzioni

I puntatori a funzioni sono molto utili per poter eseguire un particolare procedura in determinate situazioni, di solito in seguito al verificarsi di una particolare condizione. Si puó considerare come uno degli strumenti piú potenti che metta a disposizione il linguaggio C. Il C++, oltre a mantenere questa possibilità, permette anche di definire delle classi-funzioni.

Ecco come potrebbe essere sostituito il seguente puntatore a funzione:

```
void (*proc)( unsigned int);
```

Prima soluzione:

```
class Func{
```

```
public:
                        Func(){
                        }

      virtual           ~ Func(){
                        }

      virtual     void  Do( unsigned int param) = 0;
}
```

seconda soluzione: definizione di un *operator ()*:

```
class Func{
public:
                        Func(){
                        }

      virtual           ~ Func(){
                        }

      virtual     void  operator()( unsigned int param) = 0;
}
```

Nel secondo caso, la classe Func non può essere istanziata, infatti, possiede l' operatore () che è un metodo virtuale puro. In questo modo è obbligatorio ridefinire l'operatore () nella classe derivata:

```
class MyFunc : public Func{
public:
      virtual     void  operator()( Type param){
                        ...   // do something
                        }
}
```

9.11 continue, goto, break, return

Le istruzioni *goto, break e return* possono essere considerate come dei jump in assembly ma a prescindere dalla loro implementazione che potrebbe variare da un compilatore all'altro, una cosa certa è che contribuiscono notevolmente a rendere il codice poco leggibile.

L'istruzione *goto* non deve essere mai utitlizzata, in nessun caso**.** Perché ridursi a programmare in QBasic usando i costrutti del C++?
Anche il break è una sorta di goto e deve essere usato solo all'interno degli switch.
Al contrario del goto, il ritorno da procedura, *return*, è inevitabile e non se ne può fare a meno ma bisogna fare molta attenzione a non abusarne. Infatti, codice con molti *return* diventa difficile da seguire e da modificare.
Ad esempio:

```
void MyProc( MyClass* pObject)
{
    if (nil == pObject)
        return;

    pObject->DoSomething();
}
```

potrebbe diventare:

```
void MyProc( MyClass* pObject)
{
    if (nil != pObject) {
        pObject->DoSomething();
    }
}
```

in sostanza non è cambiato molto ma contribuisce a migliorare la leggibilità e soprattutto non abbiamo usato il *return* in sostituzione del *goto*!

9.12 Altri Consigli

- Attenzione a distinguere sempre tra *delete* e *delete []*. In generale, il compilatore non segnala alcun errore nel caso in cui si tenti di deallocare un array mediante la *delete* semplice.
- Limitare le dichiarazioni *extern*.
- Evitare l'uso di variabili globali.
- Usare lo *static_cast*<> invece del classico cast.
- Non usare la *memset()* per inizializzare i membri di una classe
- Non usare la *memcpy()* all'interno degli operatori di assegnamento
- Usare i reference quando è possibile. I puntatori quando non se ne può fare a meno.
- Invece di usare il postincremento è preferibile usare il preincremento. Infatti, il primo è fatto in funzione del secondo.
- Evitare l'uso dell'istruzione *continue*
- Mai fare assegnamenti all'interno di una condizione. Es.: `if (result = CreateBitmap()) { }`

10 Conclusioni

Credo che sia praticamente impossibile programmare in C++ alla perfezione, ma una cosa è usare il C++ come il C, ognuno è libero di farlo e di programmare con le tecniche che conosce meglio. Un altra cosa, invece è programmare utilizzando alcune particolarità del linguaggio solo perché sono disponibili. Infatti, è impensabile che un programmatore C appena passato al C++ si metta subito ad usare i template. Insomma, come per tutte le cose ci vuole un po' di tempo. Quindi il consiglio di scrivere usando i costrutti e le tecniche che si conoscono meglio cercando di ottenere codice "pulito" e documentato. In questo modo il codice risulta manutenibile e può essere modificato successivamente senza dover riscrivere tutto.

Spero che questa guida possa essere utile quantomeno per programmare meglio in modo da ottenere un codice pulito e, soprattutto, manutenibile. Infine, vorrei dare un ultimo consiglio: esistono infiniti stili di programmazione, ognuno con le proprie regole e fissazioni ma la cosa più importante è che quando si scrive del codice vi sia quantomeno un criterio.

11 Links Utili

1. C++ Faq Lite
2. C++ Coding Standard
3. Stanley Lippman Homepage
4. Doxygen Homepage
5. How to force MS Visual C++ to use .cc as extension for C++ files

12 Bibliografia

1. John Stenersen, “A Case for Code Review”, Gamasutra Features Articles (2000)
2. Naughty Dog, “Writing Portable Code”, Game Developers Conference Proceedings (2000)
3. Herb Sutter, “Exceptional C++”, Addison-Wesley (1999)
4. Bjarne Stroustup, “The C++ Programming Guide” 3rd edition, Addison-Wesley (1999)
5. Stanley B.Lippman, “C++ Primer 3rd edition”, Addison-Wesley (1998)
6. Stanley B.Lippman, “C++ Gems”, SIGS Books & Multimedia (1998)
7. Scott Meyers, “Effective C++” 2nd edition, Addison-Wesley (1997)
8. Steve McConnel, “Code Complete”, Microsoft Press (1993)
9. Cline, Lomow, and Girou, “*C++ FAQs*”, Addison-Wesley (1999)