14 • Gli ultimi punti brutti

E così abbiamo concluso i nostri tre esempi, mostrandoti tutto ciò di cui avevamo bisogno del linguaggio di inform per completarli, e abbiamo fatto molte osservazioni su come e perché qualcosa andava fatto. Nonostante ciò, ci sono molte cose che non sono state dette, o che sono state solo accennate. In questo capitolo rivisiteremo gli argomenti chiave e daremo uno sguardo ad alcune delle cose più importanti tra quelle che non sono state trattate, per darti un'idea dei problemi e delle soluzioni che sono state lasciate sullo scaffale.
Parleremo anche, in "Leggere il codice di altri autori" a pagina 159, di diversi modi di fare alcune cose che prima abbiamo preferito non accennarti, ma che è probabile incontrerai guardando il codice scritto da altri programmatori.
Le spiegazioni che daremo ti sembreranno forse un poco esili, ma credici: camminando in questo campo sporco e polveroso abbiamo toccato tutto ciò che è fondamentale sapere per cominciare l'apprendimento di inform. E,come sempre, l'*Inform Designer's Manual* è li a coprire tutto ciò che non abbiamo trattato.

Espressioni

In questa guida abbiamo usato il termine *espressione* diverse volte; ecco grosso modo che cosa volevamo indicare.

• Una *espressione* è un singolo *valore*, o alcuni *valori* combinati usando gli *operatori* e qualche volta le parentesi (...).

• I possibili *valori* includono:
  • un numero (-32768 to 32767)
  • qualcosa che rappresenta un numero (un personaggio 'a', una parola del dizionario 'aardvark', una stringa "l'avventura di aardvark" o una azione ##Look)
  • l'identificativo interno di una costante, un oggetto, una classe o una routine
  • (solo in una istruzione durante l'esecuzione del gioco e non in una direttiva durante la compilazione dello stesso) i contenuti di una variabile, o il valore di ritorno di una routine.

• I possibili *operatori* includono:
  • un operatore aritmetico: + - * / % ++ --
  • un operatore logico: & | ~
  • un operatore numerico di comparazione: == ~= > < >= <=
  • un operatore condizionale di un oggetto: ofclass in notin provides has hasnt
  • un operatore booleano: && || ~~

ID interni

Molte delle cose che sono definite nel file sorgente – oggetti, variabili, routine, etc. – hanno bisogno di un proprio nome, così che altre istruzioni possano riferirsi ad esse. Chiamiamo questo nome "identificativo interno" (poiché è usato solo all'interno del codice sorgente e non è visibile al giocatore), e usiamo la dicitura *obj_id*, *var_id*, *routine_id*, etc. per rappresentare dove viene usato. Un ID interno:

• può essere lungo fino a 32 caratteri

• deve iniziare con una lettera o un underscore, e quindi continuare con lettere dalla A alla Z, underscore _ e numeri da 0 a 9 (facendo attenzione al fatto che le lettere maiuscole e quelle minuscole vengono trattate allo stesso modo, ovvero non vengono distinte)

• generalmente dovrebbe essere unico in tutti i file, compresi il sorgente, le librerie standard e ogni altra libreria supplementare usata (con l'unica eccezione delle variabili locali delle routine che non sono visibili all'esterno della stessa).

Istruzioni

Una **istruzione** è una dichiarazione all'interprete, che gli dice cosa fare durante l'esecuzione del gioco. Deve essere data usando lettere minuscole, e deve sempre terminare con un punto e virgola. Alcune istruzioni, come if, controllano una o più altre istruzioni. Usiamo la dicitura *blocco_istruzioni* per rappresentare sia una singola *istruzione*, sia un qualsiasi numero di *istruzioni* racchiuse tra parentesi graffe:

istruzione;
{ *istruzione; istruzione; .... istruzione;* **}**

Le istruzioni che abbiamo incontrato

Nei nostri esempi abbiamo usato le seguenti istruzioni, quasi la metà di quelle offerte da Inform:

give *obj_id attributo*;
give *obj_id attributo attributo ... attributo*;

if (*espressione*) *blocco_istruzioni*
if (*espressione*) *nlocco_istruzioni* **else** *blocco_istruzioni*

move *obj_id* **to** *genitore_obj_id*;

new_line;

objectloop (*var_id*) *blocco_istruzioni*

print *valore*;
print *valore*, *valore*, ... *valore*;

print_ret *valore*;
print_ret *valore*, *valore*, ... *valore*;

remove *obj_id*;

return false;
return true;

style underline; print...; **style** roman;

switch (*espressione*) **{**
valore: *istruzione; istruzione; ... istruzione;*
valore: *istruzione; istruzione; ... istruzione;*
...
default: *istruzione; istruzione; ... istruzione;*
}

"*stringa***"**;
"*stringa***"**, *valore*, ... *valore*;

<*azione***>**;
<*azione noun***>**;
<*azione noun second***>**;

<<*azione***>>**;
<<*azione noun***>>**;
<<*azione noun second***>>**;

Istruzioni che non abbiamo incontrato

Sebbene nei nostri esempi non ne abbiamo avuto bisogno, queste istruzioni cicliche sono spesso utili:

break;

continue;

do *blocco_istruzioni* **until** (*espressione*)

for (*imposta_variabile* **:** *ripeti_mentre_espressione* **:** *aggiorna_variabile*) *blocco_istruzioni*

while (*espressione*) *blocco_istruzioni*

D’altra parte, ti suggeriamo di lasciare da parte per il momento questo tipo di istruzioni:

box
font
jump
spaces
string

Le regole dell’istruzione di stampa Print

Nelle istruzioni print e print_ret , ogni *valore* può essere:

• una *espressione* numerica, visualizzata come un numero decimale,

• una "*stringa*", visualizzata alla lettera o

• una regola di stampa. Puoi crearne una tua, o usarne una standard, incluse:

[NOTA DI TRADUZIONE: Modificare il seguenti articoli con i corretti comandi delle librerie italiane]

(a) *obj_id* – the object’s name, preceded by “a”, “an” or “some”
(the) *obj_id* – the object’s name, preceded by “the”
(The) *obj_id* – the object’s name, preceded by “The”
(number) *expression* – the numeric expression’s value in words

Direttive

Una **direttiva** è intesa come una istruzione per il compilatore, che gli dice cosa fare al momento della compilazione, mentre il file sorgente viene tradotto in Z-code. Per convenzione viene data con l’iniziale maiuscola (sebbene il compilatore non lo richieda obbligatoriamente) e finisce sempre con un punto e virgola.

Le direttive che abbiamo incontrato

Abbiamo usato tutte queste direttive; nota che per Class, Extend, Object e Verb la sintassi completa supportata è più sofisticata di quella in forma base che abbiamo presentato in questa guida:

```
Class class_id
    With   proprietà valore,
           proprietà valore,
           ...
           proprietà valore,
    has    attributo attributo ... attributo;

Constant const_id;
Constant const_id = espressione;
Constant const_id espressione;

Extend 'verb'
           * token token ... token -> azione
           * token token ... token -> azione
           ...
           * token token ... token -> azione;

Include "nome_del_file";

Object obj_id "nome_esterno" genitore_obj_id
    With   proprietà valore,
           proprietà valore,
           ...
           proprietà valore,
    has    attributo attributo ... attributo;

class_id obj_id " nome_esterno " genitore_obj_id
    with   proprietà valore,
           proprietà valore,
           ...
```

```
        property value,
    has attributo attributo ... attributo;

Release espressione;
Replace routine_id;
Serial "aammgg";

Verb 'verb'
        * token token ... token -> azione
        * token token ... token -> azione
        ...
        * token token ... token -> azione;

! testo di commento che viene ignorato dal compilatore

[ routine_id; istruzione; istruzione; ... istruzione; ];

#Ifdef qualsiasi_id; ... #Endif;
```

Direttive che non abbiamo incontrato

Vi è una sola direttiva davvero utile che non abbiamo avuto necessità di usare:

```
Attribute attributo;

Global var_id;
Global var_id = espressione;

Property proprietà;

Statusline score;
Statusline time;
```

Ma ve ne sono molte che non abbiamo visto e che a questo livello possono essere per ora trascurate:

```
Abbreviate
Array
Default
End
Ifndef
Ifnot
Iftrue
Iffalse
Import
Link
Lowstring
Message
Switches
System_file
Zcharacter
```

Oggetti

Un oggetto in realtà è una semplice collezione di variabili che assieme rappresentano le capacità e il corrente status di alcune componenti specifiche del modello del mondo. Tutte le variabili vengono chiamate proprietà; le più semplici variabili che possono adottare solo due valori vengono dette attributi.

Le proprietà

La libreria definisce circa quarantotto variabili proprietà standard (come ad esempio before o name), ma se ne possono creare anche altre semplicemente usandole nella definizione di un oggetto.
In particolare puoi creare ed inizializzare una proprietà nel segmento dell'oggetto introdotto dalla parola chiave with:

```
proprietà, ! impostata al valore zero/falso
proprietà valore, ! impostata ad un singolo valore
proprietà valore valore ... valore, ! impostata con una serie di valori
```

In ogni caso, il *valore* è o una *espressione* nel momento della compilazione, o una routine incapsulata:

property espressione,
proprietà [; *istruzione*; *istruzione*; ... *istruzione*;],

Puoi riferirti al valore di una proprietà in questo modo:

self.*proprietà* ! solo all'interno dello stesso oggetto
obj_id.*proprietà* ! ovunque

e puoi testare se la definizione di un oggetto include una certa proprietà così:

(*obj_id* provides *proprietà*) ! è vera o falsa

Attributi

La libreria definisce circa trenta proprietà attributi standard (come ad esempio open o worn); la creazione di attributi supplementari è richiesta piuttosto raramente.

Puoi inizializzare degli attributi nel segmento di un oggetto introdotto dalla parola chiave has :

attributo attributo ... ! impostazione iniziale
~attributo ~attributo ... ! inizialmente non impostato (valore di default)

Puoi impostare o pulire gli attributi in questo modo:

give *obj_id attributo attributo ... attributo*;
give *obj_id ~attributo ~attributo ... ~attributo*;

e puoi testare il valore corrente di un attributo così:

(*obj_id* has *attributo*) ! è vero o falso
(*obj_id* hasnt *attribute*) ! l è falso o vero

Classi

Puoi testare se un oggetto è un membro di una determinate classe:

(*obj_id* ofclass *class_id*) ! è vero o falso

L'albero degli oggetti

Puoi specificare il genitore (parent) di un oggetto (la sua locazione all'inizio del gioco) come parte della definizione dell'oggetto:

Object *obj_id* "*nome_esterno* " *genitore_obj_id*
with ...

Esiste anche un'altra sintassi che utilizza frecce -> -> come queste, che vengono anche usate per stabilire il genitore iniziale di un oggetto. Le abbiamo spiegate in "Leggere il codice di altre persone" a pagina 159.

Puoi modificare la posizione di un oggetto all'interno dell'albero:

move *obj_id* to *genitore_obj_id*;

e puoi muovere un oggetto al di fuori dell'albero così che non abbia alcun genitore (parent):

remove *obj_id*;

Dato l' *obj_id* dell'oggetto, puoi determinare quale sia il genitore (parent) dell'oggetto, il figlio più recente (child), e quello precedente a quello più giovane – l'oggetto adiacente – e quello che ha lo stesso genitore (parent).

Puoi anche contare quanti figli (children) immediati possiede:

parent(*obj_id*)
child(*obj_id*)

sibling(*obj_id*)
children(*obj_id*)

Puoi testare se un *obj_id* è un figlio (child) immediato di un altro oggetto:

(*obj_id* in *genitore_obj_id*) ! è vero o falso
(*obj_id* notin *genitore_obj_id*) ! è falso o vero

Se hai bisogno di sapere se un oggetto è un figlio (child), o un nipote (grandchild), o un pronipote (great-grandchild), etc. di un altro oggetto, usa il comando:

IndirectlyContains(*genitore_obj_id*, *obj_id*) ! è vero o falso

Infine puoi determinare l'oggetto (se esiste) di cui due oggetti sono allo stesso momento figli (child) o nipoti (grandchildren), o pronipoti (great-grandchildren), etc. usando il comando:

CommonAncestor(*obj_id1*, *obj_id2*)

Routine

Inform prevede routine indipendenti e routine incapsulate.

Routine indipendenti

Le routine indipendenti sono definite in questo modo:

[*routine_id*; *istruzione*; *istruzione*; ... *istruzione*;];

e vengono richiamate nel codice così:

routine_id()

Le routine incapsulate

Esse sono incapsulate come il valore della proprietà di un oggetto :

proprietà [;*istruzione*; *istruzione*; ... *istruzione*;],

e sono normalmente richiamate automaticamente dalla libreria, o manualmente in questo modo:

self.*proprietà*() ! solo all'interno dello stesso oggetto
obj_id.*proprietà* () ! ovunque

Argomenti e variabili locali

Entrambi i tipi di routine supportano fino a quindici variabili locali – ossia variabili che possono essere usate solo dalle istruzioni all'interno della routine, e che sono automaticamente impostate a zero ogni volta che la routine viene richiamata:

[*routine_id var_id var_id ... var_id*; *istruzione*; *istruzione*; ... *istruzione*;];
proprietà [*var_id var_id ... var_id*; *istruzione*; *istruzione*; ... *istruzione*;],

Puoi passare ad una routine fino a 7 argomenti, elencandoli tra le parentesi quando richiami la routine. L'effetto è semplicemente quello di impostare le variabili locali corrispondenti ai valori degli argomenti invece che a zero:

routine_id(*espressione*, *espressione*, ... *espressione*)

Sebbene funzioni, questa tecnica è raramente utilizzata nelle routine incapsulate, poiché non vi è un meccanismo nella libreria che possa fornire automaticamente i valori degli argomenti quando richiama la routine.

Valori di ritorno

Ogni routine restituisce (return) un singolo valore, che può essere fornito sia esplicitamente in alcune forme dall'istruzione return:

```
[ routine_id; istruzione; istruzione; ... return expr; ]; ! restituisce expr
proprietà [; istruzione; istruzione; ... return expr; ], ! restituisce expr
```

sia implicitamente quando la routine "runs out of statements". Se nessuna di queste *istruzioni* è un – return, print_ret, "..." o <<...>> – che causa un esplicito ritorno (return), allora:

```
[ routine_id; istruzione; istruzione; ... istruzione; ];
```

restituisce (return) il valore vero e

```
proprietà [; istruzione; istruzione; ... istruzione; ]
```

restituisce (return) il valore falso.

Questa differenza è *importante*. Ricordati: lasciate a se stesse, le Routine Indipendenti riportano (return) il valore Vero, Le Routine Incapsulate restituiscono (return) il valore Falso.

Ecco un esempio di routine indipendente che restituisce (return) il valore maggiore dei due argomenti che gli sono stati passati:

```
[ Max a b; if (a > b) return a; else return b; ];
```

e qui abbiamo alcuni esempi del suo uso (nota che il primo esempio sebbene sia corretto, non fa niente di utile):

```
Max(x,y);

x = Max(2,3);

if (Max(x,7) == 7) ...

switch (Max(3,y)) { ...
```

Le routine della libreria contro le routine di entrata (entry points)

Una routine della libreria è una routine indipendente, inclusa nei file della libreria stessa, che puoi richiamare dal tuo file sorgente se hai bisogno delle funzionalità che mette a disposizione tale routine. Abbiamo menzionato le seguenti routine:

```
IndirectlyContains(parent_obj_id, obj_id)
PlaceInScope(obj_id)
PlayerTo(obj_id, flag)
StartDaemon(obj_id)
StopDaemon(obj_id)
```

Diversamente, una routine che funge da punto di entrata è una routine che si può inserire nel file sorgente, in questo caso la libreria chiama la routine in uno specifico momento. Abbiamo menzionato nei nostri esempi le seguenti routine punti di entrata:

```
DeathMessage()
InScope(attore_obj_id)
```

E questa come unica obbligatoria:

```
Initialise()
```

Vi è una lista completa in "Le Routine della Libreria" a pagina 236 e in "Punti di entrata opzionali" a pagina 242.

Leggere il codice di altre persone

Giusto all'inizio di questa guida ti abbiamo avvertito che questo lavoro per forza di cose non può essere esauriente; ci siamo concentrati sulla presentazione degli aspetti più importanti di inform, cercando di essere il più chiari possibile. Naturalmente, leggendo l'*Inform Designer's Manual*, e più specificatamente andando a vedere i sorgenti di giochi completi o delle librerie opzionali prodotti da altri autori, ti imbatterai in altri modi di scrivere il codice in inform – ed è possibile che tu, come altri autori, preferisca un altro approccio diverso dai nostri metodi. La cosa importante è trovare uno stile che ti sia adatto e che sia efficace a raggiungere gli obiettivi che ti proponi. In questa sezione metteremo in risalto alcune delle differenze più evidenti che potresti trovare in altri lavori rispetto al nostro approccio.

Il layout del codice

Ogni autore ha il suo stile nello scrivere il codice sorgente, e solitamente sono tutti peggiori di quello che hai adottato tu. La flessibilità di inform rende semplice per un programmatore scegliere lo stile che lo soddisfa maggiormente; sfortunatamente, per alcuni autori questa scelta sembra essere influenzata dalla scuola d'arte di Jackson Pollock. Noi ti consigliamo di essere coerente, di usare sempre al massimo gli spazi bianchi e l'identazione, di scegliere nomi chiari per oggetti, variabili e routine, di aggiungere commenti alle sezioni difficili da comprendere, di *pensare* attivamente, mentre scrivi il tuo codice, su come renderlo il più comprensibile possibile.

Tale esigenza ti risulterà ancora più chiara andando a dare un sguardo al codice di alcune librerie opzionali prodotte dalla comunità. Questo esempio, con i nomi alterati, proviene dritto dritto dall'Archivio:

Archivio:

```
[xxxx i j;
if (j==0) rtrue;
if (i in player) rtrue;
if (i has static || (i has scenery)) rtrue;
action=##linktake;
if (runroutines(j,before) ~= 0 || (j has static || (j has scenery))) {
print "You'll have to disconnect ",(the) i," from ",(the) j," first.^";
rtrue;
}
else {
if (runroutines(i,before)~=0 || (i has static || (i has scenery))) {
print "You'll have to disconnect ",(the) i," from ",(the) j," first.^";
rtrue;
}
else
if (j hasnt concealed && j hasnt static) move j to player;
if (i hasnt static && i hasnt concealed) move i to player;
action=##linktake;
if (runroutines(j,after) ~= 0) rtrue;
print "You take ",(the) i," and ",(the) j," connected to it.^";
rtrue;
}
];
```

Ecco la stessa routine dopo alcuni minuti spesi unicamente a renderla più comprensibile; non abbiamo provveduto a testare se essa funziona, anche se il secondo else sembra sospetto:

```
[ xxxx i j;
        if (i in player || i has static or scenery || j == nothing) return true;
        action = ##LinkTake;
        if (RunRoutines(j,before) || j has static or scenery)
                "You'll have to disconnect ", (the) i, " from ", (the) j, " first.";
        else {
                if (RunRoutines(i,before) || i has static or scenery)
                        "You'll have to disconnect ", (the) i, " from ", (the) j, " first.";
                else
                  if (j hasnt static or concealed) move j to player;
                if (i hasnt static or concealed) move i to player;
                if (RunRoutines(j,after)) return true;
                "You take ", (the) i, " and ", (the) j, " connected to it.";
        }
];
```

Speriamo che tu sia d'accordo con noi che il risultato vale la pena di un piccolo lavoro extra.

Il codice viene scritto una volta, ma viene letto dozzine e dozzine di volte.

Scorciatoie

Ci sono alcune istruzioni brevi, alcune più utili di altre, nelle quale ti potresti imbattere.

• Queste cinque linee fanno la stessa cosa:

```
return true;
return 1;
return;
rtrue;
]; ! alla fine di una routine indipendente
```

• Queste quattro linee fanno la stessa cosa:

```
return false;
return 0;
rfalse;
]; ! alla fine di una routine incapsulata
```

• Queste quattro linee fanno la stessa cosa:

```
print "stringa"; new_line; return true;
print "stringa^"; return true;
print_ret "stringa";
"stringa";
```

• Queste linee sono identiche:

```
print valore1; print valore2; print valore3;
print valore1, valore2, 3;
```

• Queste linee sono identiche:

```
<azione noun second>; return true;
<<azione noun second>>;
```

• Anche queste linee sono identiche:

```
print "^";
new_line;
```

• Queste istruzioni if sono equivalenti:

```
if (MiaVar == 1 || MiaVar == 3 || MiaVar == 7) ...
if (MiaVar == 1 or 3 or 7) ...
```

• Queste istruzioni if sono equivalenti:

```
if (MiaVar ~= 1 && MiaVar ~= 3 && MiaVar ~= 7) ...
if (MiaVar ~= 1 or 3 or 7) ...
```

• In una istruzione if , l'argomento tra parentesi può essere qualsiasi tipo di espressione; tutto ciò che conta è il suo valore: zero (falso) o qualsiasi altra cosa (vero). Per esempio queste istruzioni sono equivalenti:

```
if (MiaVar ~= falsa) ...
if (~~(MiaVar == falsa)) ...
if (MiaVar ~= 0) ...
if (~~(MiaVar == 0)) ...
if (MiaVar) ...
```

Nota che la seguente istruzione testa specificatamente se MiaVar contiene il valore vero (1), e *non* se il suo valore è qualcosa di diverso da zero.

```
if (MiaVar == true) ...
```

• Se MiaVar è una variabile, l'istruzione MiaVar ++; e ++ MiaVar; funziona esattamente come MyVar = MiaVar + 1;

Per esempio, queste linee sono equivalenti:

```
MiaVar = MiaVar + 1; if (MiaVar == 3) ...
if (++MiaVar == 3) ...
if (MiaVar++ == 2) ...
```

Ciò che accomuna MiaVar++ e ++MiaVar è che entrambe aggiungo uno a MiaVar.
Ciò che le differenzia è il valore che viene ritornato (return): MiaVar++ restituisce (return) il valore corrente di MiaVar e quindi effettua l'incremento, mentre ++MiaVar esegue il "+1" prima e quindi riporta (return) il valore incrementato.
Nell'esempio, se MiaVar correntemente contiene 2 allora ++MiaVar riporta (return) 3 e MiaVar++ restituisce (return) 2, sebbene in entrambi i casi il valore di MiaVar alla fine sia 3. Come altro esempio, prendiamo questo codice (da Helga in "Guglielmo Tell"):

```
Talk: self.frasi_dette = self.frasi_dette + 1;
            switch (self.frasi_dette) {
               1:  score = score + 1;
                   print_ret "Ringrazi calorosamente Helga per la mela.";
               2:  score = score + 1;
                   print_ret "~Ci vediamo presto.~";
              default: return false;
            }
],
```

esso potrebbe essere riscritto più succintamente in questo modo:

```
Talk: switch (++self.frasi_dette) {
              1: score++;
                  print_ret "Ringrazi calorosamente Helga per la mela.";
              2: score++;
                  print_ret "~Ci vediamo presto.~";
              default: return false;
              }
],
```

• Allo stesso modo, le istruzioni MiaVar--; e --MiaVar; funzionano come MiaVar = MiaVar - 1;
Ancora una volta queste linee sono equivalenti:

```
MiaVar = MiaVar - 1; if (MiaVar == 7) ...
if (--MiaVar == 7) ...
if (MiaVar-- == 8) ...
```

proprietà "number" e attributo "General"

La libreria definisce una proprietà standard number e un attributo standard general per ogni oggetto, i cui ruoli sono indefiniti: sono variabili dallo scopo generico a disposizione del programmatore per i suoi più reconditi desideri.

Ti raccomandiamo di evitare l'uso di questi due tipi di variabili, soprattutto perché i loro nomi sono, per loro precisa natura, così insulsi da essere per lo più senza senso. Il tuo gioco risulterà molto più chiaro e semplice in fase di debug se ti avvarrai di nuove variabili proprietà – con i nomi appropriati – come parte delle definizioni dei tuoi Oggetti e Classi.

Proprietà e attributi comuni

Come alternative alla creazione di nuove proprietà individuali, che si applicano solo ad un singolo oggetto (o classe di oggetti), è possibile escogitare proprietà e nuovi attributi che, come quelli definiti dalla libreria, sono disponibili su tutti gli oggetti. La necessità di eseguire simili operazioni è piuttosto rara, ed è perlopiù usata nelle librerie supplementari (come ad esempio l'estensione alla libreria pname.h che abbiamo incontrato in "Capitan Destino: parte terza" a pagina 127 che da ad ogni oggetto la proprietà pname e un attributo phrase_matched).
Per crearli, dovresti usare queste direttive vicino l'inizio del tuo codice sorgente:

```
Attribute attributo;
Property proprietà;
```

Ti raccomandiamo di evitare di sfruttare queste due direttive senza una reale necessità del loro uso all'interno del tuo gioco. Esiste un limite di quarantotto attributi (dei quali correntemente la libreria ne definisce trenta) e sessantadue proprietà comuni (delle quali la libreria correntemente ne definisce quarantotto).
D'altra parte, il numero di proprietà particolari che puoi aggiungere è virtualmente illimitato.

Costruire l'albero degli oggetti

Per tutta questa guida, abbiamo definito la posizione iniziale di ogni oggetto nell'insieme di tutti gli oggetti (albero degli oggetti) menzionando esplicitamente il genitore (parent), nel caso vi fosse, nella prima linea della definizione di ogni oggetto – ciò che abbiamo chiamato intestazione – o, per pochi oggetti che spuntavano fuori in più di un posto usando la loro proprietà found_in . Per esempio in "Guglielmo Tell" abbiamo definito ventisette oggetti; omettendo quelli che usano la proprietà found_in per definire la loro posizione all'inizio del gioco, abbiamo definito la posizione iniziale degli oggetti in questo modo:

```
Room    strada "Una strada di Altdorf"

Room    vicino_piazza "Lungo la strada"
Furniture   banco "banco di frutta e verdura" vicino_piazza
Prop    "patate" vicino_piazza
Prop    "frutta e verdura" vicino_piazza
NPC     proprietaria "Helga" vicino_piazza

Room    piazza_sud "Lato sud della piazza"

Room    centro_piazza "In mezzo alla piazza"
Furniture   palo "palo di legno" centro_piazza
Room    piazza_nord "Lato nord della piazza"

Room   mercato "Piazza del mercato"
Object  albero "tiglio" mercato
NPC     balivo "balivo" mercato

Object  arco "arco"

Object  faretra "faretra"
Arrow   "freccia" faretra
Arrow   "freccia" faretra
Arrow   "freccia" faretra

Object  mela "mela"
```

Alcuni di questi oggetti cominciano il gioco come genitori (parent): vicino_piazza, centro_piazza, mercato e faretra tutti hanno oggetti figli (child) sotto di essi: questi figli (children) menzionano il loro genitore (parent) come ultima voce nella prima riga di definizione dell'oggetto (intestazione).
Esiste una sintassi alternativa capace di ottenere lo stesso albero di oggetti, usando delle "frecce".
Ecco come potremmo alternativamente definire le relazioni di parentela (child-parente) tra gli oggetti:

```
Room    strada "Una strada di Altdorf"

Room    vicino_piazza "Lungo la strada"
Furniture    -> banco "banco di frutta e verdura"
Prop         -> "patate"
Prop         -> "frutta e verdura"
NPC          -> proprietaria "Helga"

Room    piazza_sud "Lato sud della piazza"

Room    centro_piazza "In mezzo alla piazza"
Furniture    -> palo "palo di legno"

Room    piazza_nord "Lato nord della piazza"

Room    mercato "Piazza del mercato"
Object       -> albero "tiglio"
NPC          -> balivo "balivo"

Object  arco "arco"

Object  faretra "faretra"
Arrow        -> "freccia"
Arrow        -> "freccia"
Arrow        -> "freccia"

Object  mela "mela"
```

L'idea è che le informazione dell'intestazione di un oggetto o iniziano con una freccia, *o* finiscono con un *obj_id*, *o* nessuna delle due (entrambe le forme allo stesso tempo non sono permesse). Un oggetto con nessuna delle due diciture non ha genitore (parent): in questo esempio, abbiamo tutte le locazioni e anche l'arco, la faretra (che viene spostata nell'oggetto player nella routine Initialise) e la mela (che rimane senza genitore (parent) fino a quando Helga non la da a Guglielmo).
Un oggetto che comincia con una freccia singola -> è definito come figlio (child) dell'oggetto che lo precede più vicino senza genitore (parent). Così nell'esempio, l'oggetto albero e balivo sono entrambi figli (children) del mercato. Per definire un figlio (child) di un figlio (child), devono essere utilizzate due frecce -> ->, e così via.
In "Guglielmo Tell", questa situazione non si incontra; per illustrare come funziona, immaginiamo che all'inizio del gioco le patate e gli altri ortaggi siano *sul* banco. Avremmo quindi dovuto usare:

```
Room    vicino_piazza "Lungo la strada"
Furniture    -> banco "banco di frutta e verdura"
Prop         -> -> "patate"
Prop         -> -> "frutta e verdura"
NPC          -> proprietaria "Helga"
...
```

In questo modo, gli oggetti con una freccia (il banco e la proprietaria) sono figli (children) dell'oggetto precedente più vicino senza genitore (parent) (ovvero la locazione), e gli oggetti con due frecce (i vegetali) sono figli (children) dell'oggetto precedente più vicino definito con una singola freccia (il banco).

I vantaggi di usare le frecce includono che:

• Si è costretti a definire gli oggetti in un ordine "sensibile".

• Si devono usare meno *obj_ids* (sebbene in questo gioco la cosa non faccia molta differenza).

Gli svantaggi invece sono:

• Il fatto che gli oggetti sono vincolati alla posizione della loro definizione non è necessariamente un sistema più intuitivo per tutti i programmatori.

• Specialmente in una locazione affollata, è più difficile essere certi di come le varie relazioni di parentela (child-parent) siano impostate, oltre il dover contare con attenzione un gran numero di frecce.

• se si sposta un genitore (parent) nella gerarchia iniziale ad un livello più alto o più basso, bisogna necessariamente andare a cambiare tutti i suoi figli (children) aggiungendo o rimuovendo frecce; ciò non si rende necessario quando il genitore (parent) è invece indicato nell'intestazione dell'oggetto figlio (child).

Noi preferiamo esplicitare il nome del genitore (parent), ma ricorda che potresti incontrare entrambe le forme facilmente.

Le virgolette nel "nome" delle proprietà

Abbiamo già spiegato a pagina 47 in "Le cose tra virgolette" la differenza tra virgolette "..." (stringe che devono essere stampate) ed apici '...' (input tokens – parole del dizionario). Sfortunatamente qualche volta inform potrebbe trarti in inganno in questa distinzione: *puoi* infatti trovare in alcuni codici le virgolette nella propretà name e nelle direttive Verb:

```
NPC     proprietaria "Helga" vicino_piazza
  with  name "proprietaria" "verduraia" "venditrice" "negoziante" "mercante"
         "Helga" "vestito" "sciarpa",
        ...

Verb "chiacchera" "conversa" "c//"    * "con" creature
                                -> Talk;

Extend "parla" first            * "a"/"ad"/"all^"/"allo"/"alla"/"a"/
                              "agli"/"ai"/"alle"/"con" creature
                                -> Talk;
```

PER FAVORE non utilizzare le virgolette in questi casi. Non farai che confondere te stesso: questo sono parole

di dizionario, non stringhe; rendi le cose semplici – e più chiare possibile – in modo da non confonderti con la punteggiatura.

Usi obsolete

Infine, ricorda che Inform è stato sviluppato a partire dal 1993. In tutto questo tempo, Graham si è preoccupato di mantenere il più possibile la compatibilità con le vecchie versioni così che i giochi scritti anni fa, per versioni precedenti del compilatore e delle librerie, possano essere ancora compilati. Generalmente questa è considerata una buona cosa, ma ha i suoi svantaggi portando con se una quantità di vecchi comandi abbandonati che ancora ci rimangono tra i piedi. Potresti ad esempio vedere giochi che usano la direttiva `Nearby` (che denota la parentela, allo stesso modo di `->`) e la condizione `near` (ossia lo stesso genitore (parent)) o ancora con "\" che controlla la fine della riga in lunghe istruzioni di stampa `print` . Prova ad interpretarle, cerca di *non* usarle.