Subroutines

Una subroutine (come abbiamo visto parlando di linguaggio macchina) è un blocco di istruzioni che può essere "chiamato" da diversi punti di un programma e che, al termine, provoca il ritorno dell'esecuzione al punto di chiamata. In C++ una subroutine (in C++ si chiamano functions) si definisce così :

```
tipo nome(definizione parametri) {
    istruzioni;
    return;
}
```

E si chiama con la sintassi

```
nome(parametri);
```

main() è una particolare function (da cui comincia l'esecuzione del programma) Ogni function può essere chiamata da main() o da un'altra function (inclusa se stessa).

Variabili locali e globali

Tutte le variabili definite all'interno di una subroutine sono locali, ovvero visibili a quella sola function; sono allocate in modo automatico, a meno che l'opzione static non venga specificata prima del tipo.

Le variabili definite in un file .cpp fuori da ogni function sono invece variabili globali: sono visibili a tutte le functions contenute in quel file sorgente e sono allocate dal compilatore in modo statico.

Come anche per le functions, vale la regola che in un dato punto del sorgente possono essere utilizzati soltanto oggetti (functions, variabili globali, definizioni di strutture…) che siano stati definiti nelle righe precedenti (compilazione a passo singolo).

Esempio di variabili locali e globali

```
#include <iostream>
double in, out;
quadrato() {
        out = in*in;
}
main() {
        cin >> in;
        quadrato();
        cout << out;
}
```

in e out sono variabili globali, ovvero visibili da tutte le functions del programma. Sono allocate in modo statico.

quadrato() è una function che agisce sulle variabili locali in ed out.

main() è una function che legge da tastiera il valore di in, chiama la function quadrato() e scrive su terminale il valore di out.

I parametri delle functions

Sono variabili locali della function allocate in modo automatico il cui valore iniziale è ottenuto copiando quello della variabile specificata nella posizione omologa della riga di chiamata. Siccome la function riceve una copia della variabile specificata dal programma chiamante, non gli è possibile modificarne il valore permanentemente (passaggio "by value").

```
{
double xx = 0.123;
test(xx, 45);
}
```

```
test(double a, int b)
...
```

a vale 0.123
b vale 45

Passaggio “by reference”

Se volete che una function modifichi il valore di una variabile del programma chiamante, dovete passare un puntatore a quella variabile. Il puntatore verrà copiato, ma le due copie punteranno comunque allo stesso target; la cosa viene fatta implicitamente per i vettori (il nome del vettore corrisponde al puntatore) o esplicitamente per gli scalari (mettendo una & davanti al nome della variabile nella definizione della function).

```
int v1, v2[10];
float cx[15];
...
exa(v1,cx,v2);
```

```
exa(int &a, float b[], int *c) {
    ...
    b[3] = 11.33;
}
```

viene modificato cx[3]

Passaggio di strutture

In generale anche le strutture passate come parametri vengono copiate; questo significa che i campi vengono copiati uno a uno. Naturalmente, se tra i campi ci sono vettori o puntatori, le corrispondenti variabili sono comunque passate "by reference"; ad esempio:

```
struct lista {
    int index;
    float key;
    double dat[6];
    lista* next;
};
search(struct lista first) {
    ...
}
```

index e key verranno passati a search "by value"; il target del puntatore next e il vettore dat "by reference".

Il return value

Ogni function può restituire il valore di una variabile al programma chiamante. Il tipo di tale variabile è quello specificato prima del nome della function. Il programma chiamante può utilizzare il valore di ritorno della function inserendo la chiamata in una espressione al posto di una comune variabile:

```
float a,val;
...
val = 3*cube(x) + x;
```

```
float cube(float x) {
    float x3;
    x3 = x*x*x;
    return x3;
}
```

Viene sostituito il valore di x^3

Il main program

Ogni programma eseguibile deve contenere uno ed un solo main pogram, ovvero una function da cui inizia l'elaborazione. In C il main program è una function che si chiama main. Oltre al formato main(), esiste il formato che riceve dall'esterno (dalla riga di comandi) un intero ed un vettore di stringhe:

```
main(int argc, char** argv) {
   ...
}
```

Se si attiva il programma con la linea di comandi

```
>./test par1 par2
```

si avrà che:

argc = 3 argv[0] = "./test" argv[1] = "par1" argv[2] = "par2"

Puntatori a functions

E' possibile fabbricare puntatori a functions; la cosa è utile quando si possiede una collezione di funzioni simili e si vuole scegliere dinamicamente quale function utilizzare.

La sintassi è la seguente:

```
int func(float *x, int n) {
}
main() {
    int (*pf)(float *x, int n);
    pf = func;
    pf(a,b);
}
```

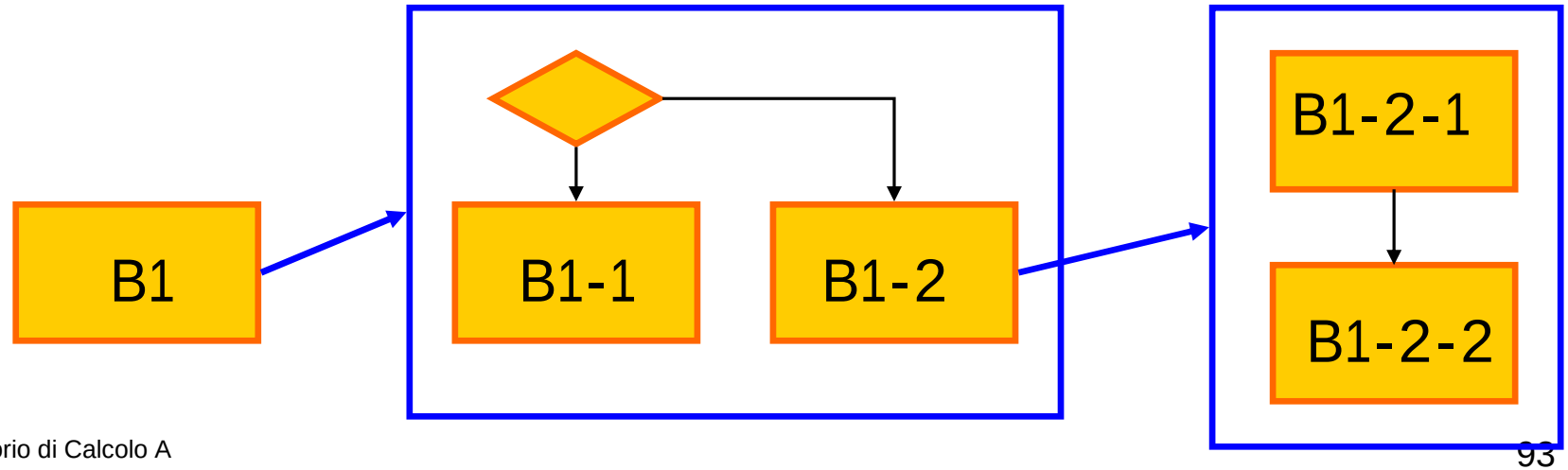
Programmazione strutturata

Con programmazione strutturata si intende la scomposizione di un programma complicato in blocchi più semplici opportunamente interconnessi tra loro. Si tratta quindi di una caratteristica del progetto più che del linguaggio, e risulta particolarmente utile in progetti complessi o quando diversi programmatori devono lavorare sullo stesso codice.

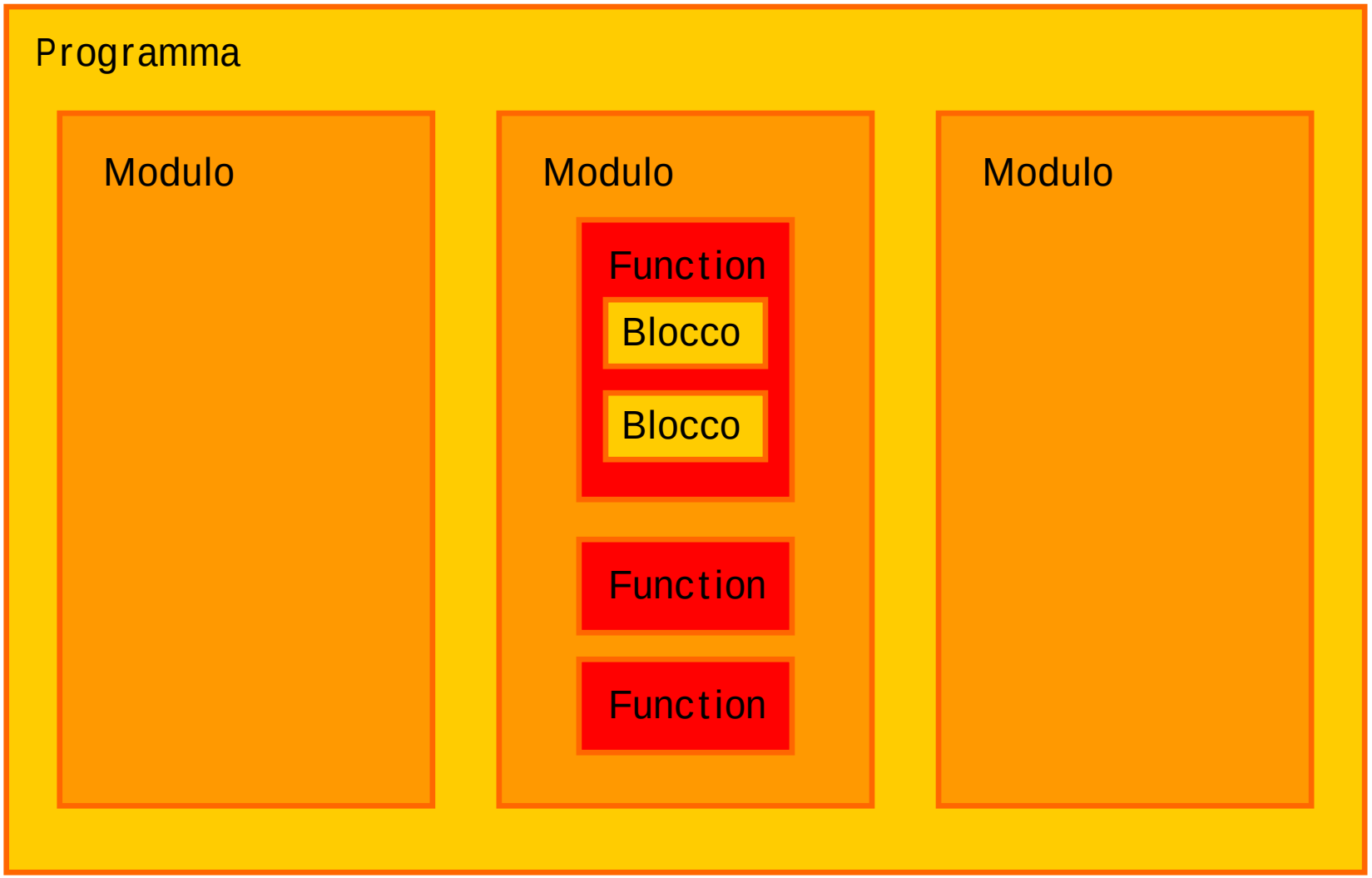
Ci sono alcuni linguaggi di programmazione che forniscono ampio supporto alla strutturazione del codice e che, in qualche modo, aiutano il programmatore a scomporre in modo corretto il programma; tra questi possiamo menzionare C++, Java, Fortan 90, C, e Pascal. Altri linguaggi,come ad esempio il Basic, non forniscono adeguato supporto alla programmazione strutturata.

Progettazione top-down

La progettazione top-down è la tecnica delle scatole cinesi: ogni blocco è scomposto in blocchi più semplici e cosi via, fino a giungere a blocchi elementari che vengono implementati nel linguaggio di programmazione. Il vantaggio di questa tecnica è che, ad ogni livello di scomposizione fornisce una descrizione completa della procedura, per cui è possibile scegliere il livello di complessità a cui si vuole analizzare il progetto (ad esempio, si può avere una idea chiara del lavoro fatto da altri senza analizzare tutti i dettagli dell'implementazione).



Struttura di un programma



I moduli

Un modulo è un insieme di functions che svolgono determinate funzioni su un insieme di variabili. In C un modulo coincide con un singolo file .c: tutte le functions all'interno di un file appartengono allo stesso modulo, e condividono tutte le variabili globali in esso definite.

In principio è possibile fare in modo che un modulo utilizzi le variabili globali di un altro modulo, ma si tratta di una pratica da censurare. Il modo corretto di utilizzare i moduli è di racchiuderci tutte le functions che servono per manipolare le variabili globali. L'accesso al modulo dall'esterno deve consistere unicamente nella chiamata di functions.

Interfacce implicite ed esplicite

Come abbiamo già detto, una function può accedere alle variabili globali o alle altre functions del modulo ad eccezione di quelle definite più avanti nel file .c. Per ovviare a questa limitazione e rendere possibile la chiamata delle functions di un modulo da parte di una function di un altro modulo si usano le interfacce esplicite. Si tratta di dichiarazioni di funzioni senza istruzioni che vengono usualmente raccolte in files di tipo .h

Tali files vengono inclusi ove sia necessario avere accesso alle routines del modulo. La ricerca delle vere functions (quelle con le istruzioni) viene effettuata durante il link.

Un pezzo di croot.h

```
// Initialization, pad handling
void R_Init(void);
void R_End(void);
void R_Divide(int nx, int ny);
void R_cd(int nn);
void R_Update(void);
void R_Delete(char *name);
void R_Interact(void);

// Basics graphical routines for 2 and 3D
void R_DrawFrame(double xmin, double xmax, double ymin, double ymax);
void R_ResizeFrame(double xmin, double xmax, double ymin, double ymax);
void R_Graph(int n, double *x, double *y, char *opt);
void R_GraphObj(char *name, int n, double *x, double *y, char *opt);
void R_GraphErrors(int n, double *x, double *y, double *ex, double *ey,
                   char *opt);
```

In principio si potrebbe omettere anche la lista degli argomenti, mettendo solo (); in questo caso però il compilatore non sarebbe in grado di verificare se i parametri specificati nelle chiamate sono coerenti con quelli della definizione della function.

Struttura all'interno delle functions

Sebbene le functions debbano eseguire solo compiti specifici e debbano essere mantenute piuttosto corte (al massimo 15-20 righe) è comunque possibile realizzare una struttura anche al loro interno: ogni gruppo di istruzioni racchiuso tra parentesi graffe rappresenta un blocco, e può definire variabili locali che non sono accessibili al resto della function.

Anche una corretta indentazione del codice è indice di buona strutturazione, in quanto aiuta ad identificare in modo rapido i blocchi e a capire il flusso del programma.

L'approccio object oriented

Nell'approccio object oriented la programmazione si basa sulla definizione di oggetti, che sono evoluzioni sia del concetto di struttura che del concetto di modulo. Si tratta di gruppi di variabili e functions che implementano funzionalità specifiche (oggetti). Nei linguaggi "OO" è possibile:

- Realizzare molte copie (funzionali) di un particolare oggetto.
- Ridefinire gli operatori standard (+, -, …) in modo che agiscano correttamente sugli oggetti definiti
- Definire oggetti che ereditano le caratteristiche da altri oggetti definiti in precedenza e ridefiniscono solo le caratteristiche nuove.

Il linguaggio C++ fornisce un ampio supporto alla programmazione object oriented. Nel seguito faremo solo alcuni accenni a queste funzionalità.

Le classi

In C++ il singolo oggetto (inteso come l'insieme dei dati e delle funzioni che su di essi operano) è rappresentato dalla classe. Si tratta di una evoluzione del concetto di struttura, alla quale è stata aggiunta la possibilità di definire funzioni che accedono in modo privilegiato ai dati contenuti nella classe (metodi) .

Per visualizzare il concetto di classe possiamo pensare ai numeri complessi. È possibile definire una struct che contenga parte reale e immaginaria in due double. Una classe consente però di aggiungere a tale definizione anche le funzioni che calcolano il modulo, la fase o una ridefinizione dell'operatore di somma.

Si deve ricordare che, come una struct, una class definisce un nuovo tipo di variabili. È poi possibile istanziare diverse variabili di quel tipo.

struct complex

Cominciamo con il creare una struttura che contiene parte reale e immaginaria in due double.

```
struct complex {
    double real;
    double imaginary;
};
```

```
int main() {
  complex c;
  double mod;
  ...
  c.real = 1.0;
  c.imaginary = -2.0;
  ...
  mod = sqrt(c.real*c.real +
             c.imaginary*c.imaginary);
}
```

I dati contenuti nella struttura sono pubblici, nel senso che tutti gli utilizzatori della struttura possono accedervi direttamente.

class complex

Il passaggio ad una classe ha come conseguenza che in assenza di indicazioni esplicite i dati sono privati, cioè vislibili solo dai metodi della classe.

In generale si raccomanda di mantenere i dati privati e di creare funzioni nella classe (metodi) che consentano di leggerli.

```
class complex {
public:
    double Re() { return real; }
    double Im() { return imaginary; }
private:
    double real;
    double imaginary;
};
```

```
int main() {
  complex c;
  double mod;
  ...
  mod = sqrt( c.Re()*c.Re() +
              c.Im()*c.Im() );
  ...
}
```

Overloading

Naturalmente i dati nella classe vanno anche scritti, e quindi si possono fabbricare delle funzioni apposite. I nomi di tali funzioni possono coincidere con quelli delle functions di lettura (overloading), visto che il compilatore può scegliere la funzione da usare in base al tipo dei parametri specificati nella chiamata.

```
class complex {
public:
    void Re(double r) { real = r; }
    double Re() { return real; }
    void Im(double i) { imaginary = i; }
    double Im() { return imaginary; }
private:
    double real;
    double imaginary;
};
```

```
int main() {
  complex c;
  double mod;
  ...
  c.Re(1.0);
  c.Im(-2.0);
  ...
  mod = sqrt( c.Re()*c.Re() +
              c.Im()*c.Im() );
  ...
}
```

Operatore di creazione

L'operatore di creazione è un metodo che viene chiamato nel momento in cui un oggetto appartenente alla classe viene creato. Il nome del metodo coincide con quello della classe. Si possono usare parametri che vengono utilizzati per inizializzare l'oggetto. È possibile l'overloading.

```
class complex {
public:
    complex() { real = 0.0; imaginary = 0.0; }
    complex(double r, double i) { real = r; imaginary = i; }
    void Re(double r) { real = r; }
    double Re() { return real; }
    void Im(double i) { imaginary = i; }
    double Im() { return imaginary; }
private:
    double real;
    double imaginary;
};
```

```
int main() {
  complex c(1.0, -2.0);
  double mod;
  ...
  mod = sqrt( c.Re()*c.Re() +
              c.Im()*c.Im() );
  ...
}
```

Metodi accessori

Si possono aggiungere metodi che semplificano la manipolazione degli oggetti.

```
class complex {
public:
    complex() { real = 0.0; imaginary = 0.0; }
    complex(double r, double i) { real = r; imaginary = i; }
    double Mod() { return sqrt(real*real + imaginary*imaginary); }
    double Phase() { return atan2(imaginary,real); }
    void Re(double r) { real = r; }
    double Re() { return real; }
    void Im(double i) { imaginary = i; }
    double Im() { return imaginary; }
private:
    double real;
    double imaginary;
};
```

```
int main() {
  complex c(1.0, -2.0);
  double mod;
  ...
  mod = c.Mod();
  ...
}
```

Ridefinizione di operatori

Il C++ consente di (ri)definire il modo in cui gli operatori (+, -, *, /, ...) agiscono sugli elementi di una classe.

Se a e b sono oggetti di una classe, allora per definire il comportamento di

a + b

si deve pensare che l'operazione sia scritta come

a.operator+(b)

e definire il metodo operator+ in modo acconcio.

Si capisce che, nel caso di operatori binari, spetta sempre alla classe che compare a sinistra dell'operatore di eseguire la definizione. Si deve anche ricordare che un operatore è una funzione che restituisce un valore (nel caso del + la somma dei due addendi).

class complex

```
class complex {
public:
    complex() { real = 0.0; imaginary = 0.0; }
    complex(double r, double i) { real = r; imaginary = i; }
    double Mod() { return sqrt(real*real + imaginary*imaginary); }
    double Phase() { return atan2(imaginary,real); }
    double Mod() { return sqrt(real*real + imaginary*imaginary); }
    double Phase() { return atan2(imaginary,real); }
    double Re() { return real; }
    double Im() { return imaginary; }
    complex operator+(const complex &b) {
        complex c;
        c.real = real + b.real;
        c.imaginary = imaginary + b.imaginary;
        return c;
    }
    bool operator==(const complex &b) {
        if (real == b.real && imaginary==b.imaginary)  {
            return true;
    }
        return false;
    }
private:
    double real;
    double imaginary;
};
```

```
int main() {
  complex a;
  complex b(1.0, -2.0);
  complex c(-1.0, 7.0);
  ...
  a = b + c;
  ...
  if (a == c) {
    ...
  }
  ...
}
```

Osservazioni sulla strategia OO

- Come abbiamo osservato nell'evoluzione da struct a class, c'è una tendenza a semplificare sempre più il main scaricando la complessità nella definizione delle classi. La programmazione a oggetti prevede che tutto il codice sia incluso in una classe.
- Noi ci siamo soffermati su uno degli aspetti importanti delle classi, ovvero la possibilità di estendere le capacità di base del linguaggio includendo nuovi tipi di variabili (o oggetti) e nuove definizioni degli operatori. Esiste un altro aspetto importante: una classe può derivare le sue funzionalità di base da una classe preesistente, aggiungendo o modificando solo alcuni metodi specifici. Questa possibilità rende molto facile adattare programmi esistenti a nuove necessità.
- Alcuni tipi di variabili che abbiamo usato (string e fstream) sono classi appartenenti alla cosiddetta "C++ Standard Library" o STL.
- Malgrado il fatto che la programmazione OO sia molto efficace e molto di moda, non è vero che tutti i problemi si prestano ad essere risolti in questo modo. Un buon progetto software non cerca di "oggettizzare" a tutti i costi, ma cerca di sfruttare al meglio le tecniche OO dove queste sono in grado di fornire i migliori risultati.