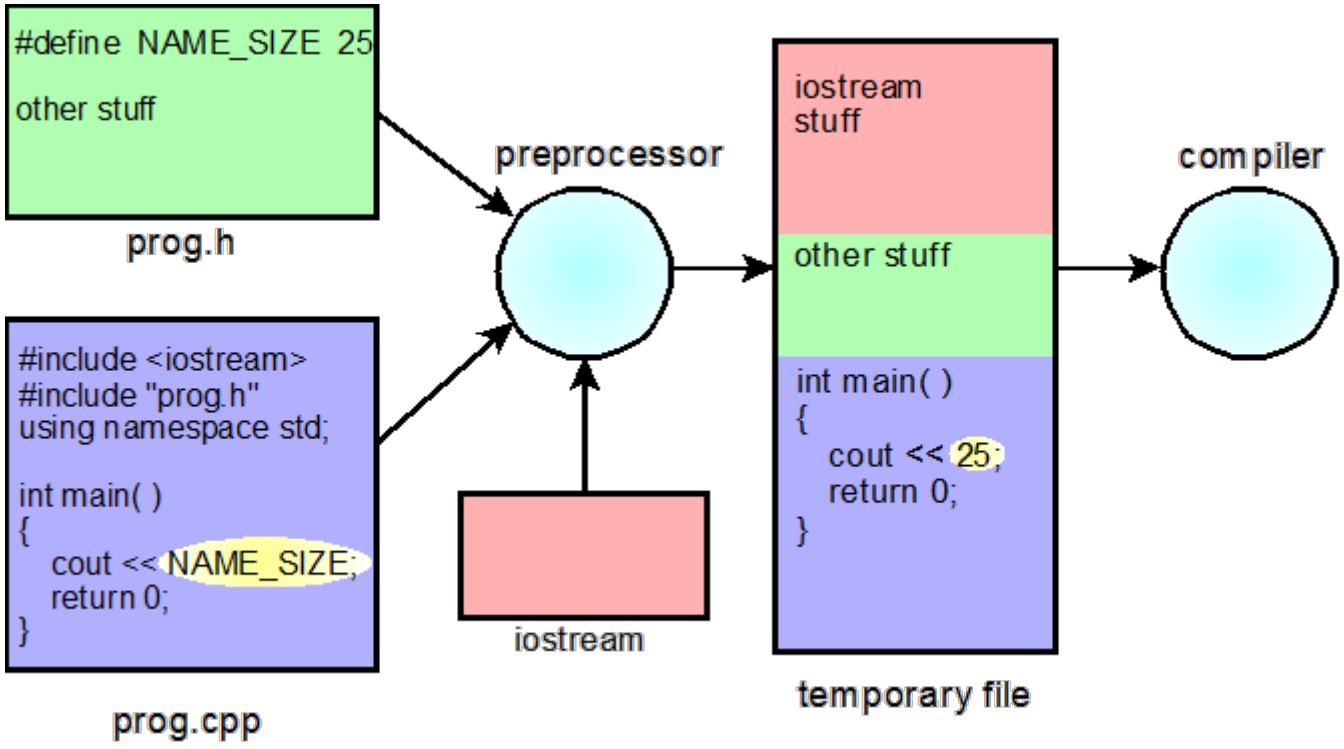
Programmazione a oggetti

Direttive al preprocessore
spazio dei nomi
Parametri di default
Overlaoding delle funzioni

A.A. 2020/2021
Francesco Fontanella

Preprocessore

- Il preprocessore C effettua elaborazioni testuali dei file sorgente e genera il “codice sorgente” effettivamente compilato



Direttive al preprocessore

- È possibile dare delle direttive al preprocessore.
- Le direttive del preprocessore iniziano con il carattere '#'
- **Esempio (noto)**
  - `#include <nomefile>`
    Inclusione header file delle librerie del linguaggio (compilatore)
  - `#include "filepath"`
  - Inclusione di altri file header, tipicamente definiti dal programmatore

Funzioni macro

- Con la direttiva `#define` E' possibile definire delle "macro":

Esempio

```
#define max(a,b)  ((a) > (b) ? (a) : (b))

cout<<endl<<2 * max(3+3, 7);
```



```
cout<<endl<<2 * ((3+3) > (7) ? (3+3) : (7));
```

L'operatore ?

- L'operatore ? implementa una forma (compatta) di **if then else**:

```
if (a > b)
   c = a
else  c = b
```

equivalente a

```
c = a > b ? a : b
```

Compilazione condizionale

- Le direttive

  `#if, #elif, #else, #endif`

  informano il preprocessore di compilare solo parte del codice

- Possono essere utilizzate per creare codice più efficiente e soprattutto più portabile

Compilazione condizionale

```
#if condition_1
source_block_1
...
#elif condition_2
source_block_2
...
#elif condition_n
source_block_n
...
#else
default_source_block
...
#endif
```

Esempio

```
#define ENGLAND   0
#define FRANCE 1
#define ITALY  0
#if ENGLAND
#include "england.h"
#elif FRANCE
#include "france.h"
#elif ITALY
#include "italy.h"
#else
#include "canada.h"
#endif
```

E' possibile testare il valore di macro definite con #define

Compilazione condizionale: Debug

- E' possibile, ad esempio, aggiungere/rimuovere codice di debug dalla compilazione:
  - in fase di debug vengono compilate le istruzioni per il debug
  - in fase di rilascio si escludono dalla compilazione le istruzioni utilizzate per il debug
- Esempio

```
#define DEBUG 1

   ...
   #if DEBUG
   cout<<endl<<"Debug: function funz...";
   #endif
```

La direttiva #ifdef

- verifica se una certo nome di macro è definito
- **Esempio**

```
#ifdef _WIN32
  void pausa() {system("PAUSE");}
  #else
  void pausa() {
    char a;
    cout<<endl<<"premere un tasto per continuare";
    cin >> a;
  }
  #endif
```

Definita dal compilatore, specifica il tipo di sistema operativo

NOTA
La direttiva `#ifdef` può essere scritta anche così:

```
#if defined _WIN32
```

La direttiva #ifndef

- La direttiva `#ifndef` ha effetto se un certo nome di macro NON è definito
- Utile per impedire che uno stesso header file venga incluso più di una volta nello stesso file sorgente
- **Esempio**

Nomeheader.h

```
#ifndef _NOMEHEADER_H
#define _NOMEHEADER_H
//Corpo dell'header
...
#endif
```

La direttiva #undef

- La direttiva **`#undef`** rimuove la definizione di un certo parametro
- Serve a rendere locali i nomi delle macro

- **Esempio**
  - si vuole che DEBUG sia definita solo nella primaparte di un file, mentre nella seconda parte no:
    - Si inserirà **`#undef DEBUG`** dove necessario

Lo spazio dei nomi

- regione di codice che viene nominata dal programmatore, sintassi:

```
namespace identificatore
{corpo_dello_spazio_dei_nomi}
```

- Si può accedere ai suoi elementi per mezzo dell'operatore di risoluzione di scopo (::).

  Esempi

```
identificatore::nome_variabile

identificatore::nome_funzione
```

- I namespace consentono di evitare "collisioni" tra i nomi delle tantissime variabili, funzioni, classi definite sia nelle librerie standard che in quelle definite dagli stessi utenti

> **NOTA**
> Il **namespace** `std` è quello della libreria standard del C++

Esempio

```
namespace geo {
 const double PIGRECO = 3.141592;
 double lungcircon (double raggio)
   { return 2*PIGRECO*raggio; }
}
...
double c = lungcircon(16); // ERRORE!
...
double c = geo::lungcircon(16);   // OK!
```

La direttiva using

- Utilizzata per non ripetere molte volte l'operatore ::
- Lo spazio di nome diventa visibile da quel momento in poi, fino alla fine del blocco di codice in cui essa è inserita
- **Esempio**

```
 void calcoloSup()
{
   using namespace geo;
   double c = lungcircon(16);   // OK!
}

...

double c = lungcircon(16); // ERRORE!,
```

Spazi dei nomi multipli

- la dichiarazione può spezzarsi e distribuirsi in vari punti del sorgente
- **Esempio**

```
namespace geo
    { double PIGRECO = 3.141592;
    } //fine dello spazio di nome Geo
    ...
    namespace geo
    { double lungcircon (double raggio)
          { return 2 * PIGRECO * raggio;
            }
    }  //fine dello spazio di nome Geo
```

Parametri di default

- In C++ è possibile specificare dei valori default per i parametri.
- Se l'invocazione della funzione non specifica gli ultimi argomenti, questi vengono assunti uguali agli argomenti di default.
- Solo gli ultimi (o tutti) i parametri possono avere valori di default.

Esempi

```
void myfunc(int a, int b, int c =0);

int main() {
int x,y;

myfunc(x,y);

}
```

La funzione prende 3 parametri, il terzo è omesso in quanto è quello di default (che vale 0)

IMPORTANTE!

E' il prototipo che deve specificare gli argomenti di default.

```
void PrintValues(int val1, int val2=10)
{
   cout << "1st value: " << nValue1 << endl;
   cout << "2nd value: " << nValue2 << endl;
}

int main()
{
  PrintValues(1); // secondo parametro: default(è 10)
  PrintValues(3, 4); // secondo parametro: è 4

  return;
}
```

OUTPUT

```
1st value: 1
2nd value: 10
1st value: 3
2nd value: 4
```

- Una funzione può avere tutti i parametri di default:

```
void PrintValues(int val1=10, int val2=10, int val3=30)
{
  cout<<"Values: "<<val1<<" "<<val3<<" "<< val3<<endl;
}

int main()
{
   PrintValues(1, 2, 3);
   PrintValues(1, 2);
   PrintValues(1);
   PrintValues();

  return;
}
```

OUTPUT

```
Values: 1 2 3
Values: 1 2 30
Values: 1 20 30
Values: 10 20 30
```

- Per fornire un valore diverso da quello di default per `val3` è necessario fornire anche i valori di default per `val1` e `val2`

```
PrintValues(10, 20, 3);
```

- Infatti la chiamata:

```
PrintValues(3);
```

è equivalente a

```
PrintValues(3, 20, 30);
```

- I parametri di default devono stare per ultimi:

```
void PrintValue(int val1=10, int val2);
```

vietato!

- I parametri di default più a sinistra dovrebbero essere quelli che più di frequente sono diversi da quelli di default.

Overloading delle funzioni

- In C++, è possibile dare a funzioni diverse lo stesso nome, a condizione che le funzioni abbiano liste di parametri diverse (in numero e/o tipo).
- Il compilatore è in grado di associare in modo univoco ciascuna chiamata a una delle funzioni, distinte in base alla lista degli argomenti
- **Esempio**

```
void incr(int &x) {
  x++;
}

void incr(float &x) {
  x = x + 1.0;
}

void incr(int &x, int dx) {
  x += dx;
}
```

```
main () {

float a=0.0;
int i = 1;
int d = 10;

incr(a);

incr(i);

incr(i, d);

}
```

Firma delle funzioni

- Quando uno stesso nome corrisponde a più funzioni, si dice che tale nome è “sovraccaricato” (overloaded) di significato.

- La cosiddetta “firma” (signature) di una funzione è costituita da:

  nome + tipo dei parametri

- In presenza della chiamata ad una funzione overloaded, il compilatore è in grado di “capire” quale funzione chiamare

NOTA

Avere tipi di ritorno diversi non è sufficiente a distinguere le funzioni,

Overloading di funzioni e ambiguità

- Esistono situazioni in cui il compilatore non è in grado di distinguere tra due o più funzioni in overloading. In questo caso si parla di ambiguità. Le ambiguità generano errori di compilazione.
- Le ambiguità sono di solito causate dalle conversioni automatiche di tipo.
- **Esempio**

```
double myfunc(double i); // versione A
float myfunc(float i);   // versione B

int main()
{
  myfunc(10.1);
  myfunc(10);

  return 0;

}
```

NON AMBIGUA: chiama la A. le costanti con virgola mobile sono automaticamente di tipo double

AMBIGUA!: come deve essere convertita la costante? In float o double?