Costanti

- Le costanti definite dal preprocessore hanno scope globale e non hanno tipo

```
#define NUMELEM 100
```

- Le costanti definite con la keyword const hanno scope locale e hanno tipo

```
const int NumElem = 100
```

Costanti runtime

- Vantaggio = sostituzione nome/valore da parte del compilatore
- Anche nei casi in cui il compilatore non può sostituire nome/valore può essere utile usare costanti
- In pratica si indica che non può essere modificato il valore di una "variabile"

Safecons.cpp

Costanti e puntatori

- Puntatore a costante

```
const int* p;
```

p è un puntatore ad un "const int", quindi è un puntatore ad un valore che non può esser modificato

- Puntatore costante

```
int x = 1;
int * const p = &x;
```

p è un puntatore costante ad un intero, quindi è un puntatore che non può esser modificato (ma può essere modificato il valore puntato)

ConstPointers.cpp

Costanti e funzioni

- Parametri Const
  - il default è il passaggio per valore: non ha molto senso dichiarare const dei parametri passati per valore : int f(const int x)
  - utile per passaggio per reference : int f(&x)
- Reference (&)
  - è come un puntatore costante che viene automaticamente dereferenziato.
  - A differenza dei puntatori standard, dopo l'inizializzzione una reference non può puntare ad un oggetto diverso

Reference e funzioni

- Parametri Reference
  - equivale ad usare puntatori, con una sintassi più semplice
  - modifiche fatte nella funzione cambiano l'originale
- Reference come valori ritornati
  - è necessario che la memoria indirizzata dalla reference non scompaia all'uscita dalla funzione

Reference (esempio)

```
int* f(int* x) {
  (*x)++;
  return x; // Safe, x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe, outside this scope
}
```

q è in uno scope locale ad h e scompare al termine della funzione - x, anche se locale non scompare

```
int& h() {
  int q;
//!  return q;  // Error
  static int x;
  return x; // Safe, x lives outside this scope
}

int main() {
  int a = 0;
  f(&a); // Ugly (but explicit)
  g(a);  // Clean (but hidden)
} ///:~
```

Costanti e Reference

- Parametri Const Reference
  - si passa l'indirizzo
  - non si modifica il valore
  - una function con argomenti che non sono const non può ricevere parametri da variabili const
  - una function con argomenti const può ricevere parametri sia da variabili const che non const
  - attenzione alle variabili temporanee create automaticamente dal compilatore (es.)

```
void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Error
  g(1);
}
```

Valore ritornato Const

- Restituire un indirizzo
  - se una function ritorna un indirizzo, questo può essere per ricevere un valore in una assegnazione
  - se il valore ritornato è definito const allora non può essere usato a sinistra in una assegnazione

```
class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};
X::X(int ii) { i = ii; }
void X::modify() { i++; }
```

```
X f5() {
  return X();
}
const X f6() {
  return X();
}
```

```
int main() {
  f5() = X(1); // OK -- non-const return value
  f5().modify(); // OK
// Causes compile-time errors:
//!  f6() = X(1);
//!  f6().modify();
}
```

Const e Classi
1 - Const e Costruttori

- variabile const in una classe
  - ha storage distinto in ogni oggetto istanziato, e non può essere modificata

```
class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};
```

- in ogni istanza può aver un valore diverso
  - come assegnare un valore iniziale ?
  - Constructor initializer list

```
Fred::Fred(int sz) : size(sz) {}
```

ConstInitialization.cpp
BuiltInTypeConstructors.cpp

Const e Classi
2 - Const del compilatore

- Per definire una const "del compilatore" (sostituzione nome/valore a compile time) si usa la keyword **static const**
  - deve essere inizializzata nella definizione

```
class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  ...
```

StringStack.cpp

Const e Classi
3 - oggetti const

- un oggetto const ha lo stesso significato di un tipo di base const:

```
const Utente X(9,8,7);
```

- l'oggetto X, di classe Utente, viene inizializzato con il costruttore della classe
- dopo l'inizializzazione non è più possibile modificare il contenuto dell'oggetto
- come fa il compilatore a garantirlo ?

Const e Classi
4 - metodi const

- se un oggetto è dichiarato const, non è possibile richiamarne i metodi standard
- possono essere richiamati solo metodi dichiarati (e definiti) come "metodi const"

```
class X {
  int i;
public:
  X(int ii);
  int f();
};
int X::f() {
  return i;
}
```

NO !

```
const X x2(20);
x2.f();
```

SI !

```
class X {
  int i;
public:
  X(int ii);
  int f() const;
};
int X::f() const {
  return i;
}
```

Funzioni inline

- Le macro del preprocessore sono da usare con cautela (side effects, scope, etc)

```
#define Doppio(x) (2*x)
```

MacroSideEffects.cpp

- Le funzioni inline sono function che sono (possono essere) espanse dal compilatore nel punto in cui vengono richiamate

```
inline int Doppio(int x) { return 2*x; }
```

- Limitazioni: complessità, indirizzo

inline e classi

- Metodi definiti nella definizione della classe sono inline per default

Inline.cpp

- inline per l'accesso allo stato di oggetti
  - metodi pubblici per lettura/scrittura variabili private
  - l'overloading dei nomi consente di usare lo stesso nome

Access.cpp

Rectangle.cpp

- inline e costruttori
  - Attenzione all'attività nascosta

Hidden.cpp