Si **assumano** le seguenti specifiche riguardanti la libreria Qt (**attenzione: non si tratta di codice da definire!**).

- `QWidget` è la classe base di tutte le classi Gui della libreria Qt.
  - La classe `QWidget` ha il distruttore virtuale.
  - La classe `QWidget` rende disponibile un metodo virtuale `QSize sizeHint() const` con il seguente comportamento: `w.sizeHint()` ritorna un oggetto di tipo `QSize` che rappresenta la dimensione raccomandata per il widget `w`. È disponibile l'operatore esterno di uguaglianza `bool operator==(const QSize&, const QSize&)` che testa l'uguaglianza tra oggetti di `QSize`.
  - La classe `QWidget` rende disponibile un metodo virtuale di clonazione `QWidget* clone()` con l'usuale contratto di "costruttore di copia polimorfo": `pw->clone()` ritorna un puntatore polimorfo ad nuovo oggetto `QWidget` che è una copia polimorfa di `*pw`. Ogni sottoclasse di `QWidget` definisce quindi il proprio overriding di `clone()`.
- La classe `QAbstractButton` deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti i button widgets.
  - Le classi `QCheckBox` e `QPushButton` derivano direttamente e pubblicamente da `QAbstractButton`. Le classi `QCheckBox` e `QPushButton` definiscono il proprio overriding di `QWidget::sizeHint()`.
- La classe `QAbstractSlider` deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti gli slider widgets.
  - Le classi `QScrollBar` e `QSlider` derivano direttamente e pubblicamente da `QAbstractSlider`. Entrambe le classi definiscono il proprio overriding di `QWidget::sizeHint()`.

Definire una funzione `vector<QAbstractButton*> fun(list<QWidget*>&, const QSize&, vector<const QWidget*>&)` con il seguente comportamento: in ogni invocazione `fun(lst,sz,w)`, per ogni puntatore `p` elemento (di tipo `QWidget*`) della lista `lst`:

(a) se `p` non è nullo e `*p` ha una dimensione raccomandata uguale a `sz` allora inserisce nel vector `w` un puntatore ad una copia di `*p`;

(b) se `p` non è nullo, `*p` non è uno slider widget e ha una dimensione raccomandata uguale a `sz` allora rimuove dalla lista `lst` il puntatore `p` e dealloca l'oggetto `*p`;

(c) se `p` non è nullo, `p` non è già stato rimosso al precedente punto (b) e `*p` è un `QCheckBox` oppure un `QPushButton` allora rimuove dalla lista `lst` il puntatore `p` e lo inserisce nel vector di `QAbstractButton*` che la funzione deve ritornare;

La funzione infine ritorna il vector di `QAbstractButton*` che è stato popolato come specificato al punto (c).

Esercizio 2

Scrivere un programma che consista esattamente di tre classi `A`, `B` e `C`, dove `B` è un sottotipo di `A`, mentre `C` non è in relazione di subtyping nè con `A` nè con `B`, che dimostri in un metodo di `C` un tipico esempio di un **uso giustificato e necessario** della conversione di tipo `dynamic_cast` per effettuare type downcasting. A questo fine, si usino il **minor numero possibile di metodi**.

Esercizio 3

Siano `A`, `B`, `C` e `D` quattro **diverse** classi polimorfe. Si considerino le seguenti definizioni.

```
template <class X, class Y>
X* fun(X* p) { return dynamic_cast<Y*>(p); }

main() {
  C c; fun<A,B>(&c);
  if( fun<A,B>(new C()) == 0 ) cout << "Bjarne ";
  if( dynamic_cast<C*>(new B()) == 0 ) cout << "Stroustrup";
  A* p = fun<D,B>(new D());
}
```

Si supponga che:

1. il `main()` compili correttamente ed esegua senza provocare errori a run-time;
2. l'esecuzione del `main()` provochi in output su `cout` la stampa `Bjarne Stroustrup`.

In tali ipotesi, per ognuna delle relazioni di sottotipo `T1`≤`T2` nelle seguenti tabelle segnare con una croce l'entrata

(a) "Vero" per indicare che `T1` **sicuramente è** sottotipo di `T2`;

(b) "Falso" per indicare che `T1` **sicuramente non è** sottotipo di `T2`;

(c) "Possibile" **altrimenti**, ovvero se non valgono nè (a) nè (b).

	Vero	Falso	Possibile
A≤B			
A≤C			
A≤D			
B≤A			
B≤C			
B≤D			

	Vero	Falso	Possibile
C≤A			
C≤B			
C≤D			
D≤A			
D≤B			
D≤C			

Esercizio 4

```
class Z {
public:
  Z(int x=0) {}
};

class B {
private:
  Z bz;
};

class C: virtual public B {
private:
  Z* cz;
};

class D: public C {
};

class E: virtual public B {
public:
  Z ez;
};

class F: public D, public E {
private:
  Z* pz;
public:
  // ridefinizione del costruttore di copia di F
};
```

Si considerino le definizioni nel riquadro sopra. Ridefinire (ovviamente senza usare la keyword `default`) nel riquadro sottostante il costruttore di copia della classe `F` in modo tale che il suo comportamento coincida esattamente con quello del costruttore di copia **standard** di `F`.

SOLUZIONE