```
class A {
public:
  virtual ~A() = 0;
};
A::~A() = default;

class B: public A {
public:
  ~B() = default;
};

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

```
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};
```

```
int main() {
B b; C c; D d; E e;

cout << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...)
     << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...);
}
```

Si considerino le precedenti definizioni ed il `main()` incompleto. Definire opportunamente negli appositi spazi `...,...` le chiamate alla funzione `F` di questo `main()` usando gli oggetti locali `b`, `c`, `d`, `e`, `f` in modo tale che: (1) non vi siano errori in compilazione o a run-time; (2) le chiamate di `F` siano **tutte diverse** tra loro; (3) l'esecuzione produca in output **esattamente** la stampa `6544233241`.

Esercizio 1

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

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

int main() {
  B b; C c; D d; E e;
  if( fun<A,B>(c) == nullptr ) cout << "Programmazione ";
  if( dynamic_cast<C*>(&b) == nullptr ) cout << "ad Oggetti";
  const A* p = fun<D,B>(d);
  const D*q= fun<E,B>(d);
  c=e;
}
```

Si supponga che:

1. il `main()` compili correttamente ed esegua senza provocare errori a run-time o undefined behaviour;
2. l'esecuzione del `main()` provochi in output su `cout` la stampa `Programmazione ad Oggetti`.

In tali ipotesi, per ognuna delle relazioni di sottotipo `T1`≤`T2` nella seguente **tabella da ricopiare nel foglio** scrivere nella corrispondente cella:

(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).

A≤C	A≤D	B≤A	B≤C	B≤D	C≤D	E≤A	E≤B	E≤C	E≤D
F	F	V	F	F	F	V	V	V	F

Quesito 3:

quesito3.cpp 1

```
template <class A,class B>
A*Fun(X*p){return dynamic_cast<B*>(p);}

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

	Vero	Falso	Possibile
A<=B		X	
A<=C		X	
A<=D		X	
B<=A	X		
B<=C			X
B<=D		X	

	Vero	Falso	Possibile
C<=A	X		
C<=B			X
C<=D		X	
D<=A	X		
D<=B	X		
D<=C		X	

```cpp
class A {
  bool x;
public:
  virtual ~A() = default;
};

class B {
  bool y;
public:
  virtual void f() const { cout << "B::f "; }
};

class C: public A {};

class D: public B {
public:
  void f() const { cout << "D::f "; }
};

class E: public D {
public:
  void f() const { cout << "E::f "; }
};

template<class T>
void Fun(const T& ref) {
  try{ throw ref; }
  catch(const C& c) {cout << "C ";}
  catch(const E& e) {cout << "E "; e.f();}
  catch(const B& b) {cout << "B "; b.f();}
  catch(const A& a) {cout << "A ";}
  catch(const D& d) {cout << "D ";}
  catch(...)        {cout << "GEN ";}
}

C c; D d; E e; A& a1 = c; B& b1 = d; B& b2 = e; D& d1 = e; D* pd = dynamic_cast<E*>(&b2);
```

```
/*
RISPOSTE ACCETTATE:
C
B D::f
E E::f
A
B D::f oppure B B::f
B E::f oppure B D::f
B E::f oppure B D::f
B E::f oppure B D::f
B E::f oppure B D::f
NON COMPILA
E E::f
NON COMPILA
A

La possibilità di due risposte è dovuta al fatto che throw non lancia
direttamente l'oggetto usato, ma una sua copia. Viene quindi lanciato un T(ref)
e non ref! Tale oggetto chiaramente ha tipo statico e dinamico uguale a T.
Questa nozione non e' stata accennata in aula (anche perché era ignota al
professore stesso). È stato deciso perciò di accettare anche delle risposte
alternative, seppur coerenti col comportamento del tipo dinamico degli oggetti
*/
```

```cpp
#include <iostream>
using namespace std;

class A{
public:
   virtual ~A() {};
};

class B: public A {};
class C: public A {};

class D: public C {};

template <class T>
A* Fun(T* pt) {
    bool b = false;
    try {throw pt;}
    catch(B*) {cout << "B"; b=true;}
    catch(C*) {cout << "C"; b=true;}
    catch(D*) {cout << "D"; b=true;} // Exception of type 'D *' will be caught by earlier handle
    catch(A*) {cout << "A"; b=true;}
    if(!b) cout << "NO";
    return dynamic_cast<C*>(pt) != nullptr ? static_cast<A*>(pt):new D;
}

int main(){
    B b; C c; D d; A* pa1 = &b; A* pa2 = &d; B* pb1 = dynamic_cast<B*>(pa1); B* pb2 = dynamic_cast<B*>(pa2);
    Fun(&c); cout << endl; // "C"
    Fun(&d); cout << endl; // "C"
    Fun(pa1); cout << endl; // "A"
    Fun(pa2); cout << endl; // "A"
    Fun(pb1); cout << endl; // "B"
    Fun(pb2); cout << endl; // "B"
    Fun<A>(pb1); cout << endl; // "A"
    Fun<A>(pa2); cout << endl; // "A"
    Fun<B>(pb1); cout << endl; // "B"
  //Fun<C>(pa2); no matching function
    Fun<C>(&d); cout << endl; // "C"
  //Fun<D>(pa2); no matching function
    Fun(Fun(pa2)); cout << endl; // "AA"
    Fun(Fun(pb2)); cout << endl; // "BA"
    Fun(Fun(pb1)); cout << endl; // "BA"
}
```

POLIMORFA => DISTR. VIRTUALE

FUN(PB1) => B*

FUN(RISULTATO) A

A ← (

DOWNCAST (B

T* fun(T* pt) ⟶ A B (B* pb1 = dyn_cast<B*>(pa1) → dereferenzia e becca pa1(A)

T* fun(T& ref) ⟶ B B (è sempre riferimento ⟶ mette sempre il suo TS (B)

STL = Standard Template Library

STL = Classe container → Vector<T*>

C++ 17 in su:
for_each
auto
default
decltype

```cpp
class Z {
public:
  ...
};

template <class T1, class T2=Z>
class C {
public:
  T1 x;
  T2* p;
};

template<class T1, class T2>
void fun(C<T1,T2>* q) {
  ++(q->p);
  if(true == false) cout << ++(q->x);
  else cout << q->p;
  (q->x)++;
  if(*(q->p) == q->x) *(q->p) = q->x;
  T1* ptr = &(q->x);
  T2 t2 = q->x;
}

main(){
  C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
}
```

Q = Z ? / T ?

) PREFIX ++

POSTFIX ++

BOOL OPERATOR == Z

(Z) OPERATOR ++

(Z) OPERATOR ++ (INT)

→ Z (CONST Z OR Z)

Si considerino le precedenti definizioni. Fornire una dichiarazione (non è richiesta la definizione) dei membri pubblici della classe `Z` nel **minor numero possibile** in modo tale che la compilazione del precedente `main()` non produca errori. **Attenzione:** ogni dichiarazione in `Z` non necessaria per la corretta compilazione del `main()` sarà penalizzata.

```
template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); //nessun requirement
  if(true == false) cout << ++(q->x); else cout << q->p; //q->x di tipo T1, operator++() T1
  (q->x)++; //operator++(int) su T1
  if(*(q->p) == q->x) *(q->p) = q->x; //(1) bool operator==(T2, T1), (2) operator=(T2, const T1&)
  T1* ptr = &(q->x); //nessun requirement
  T2 t2 = q->x; //T2(const T1&)
}

Soluzione:
class Z{
        Z operator++(int); // 1
        Z operator++(); // 2
        Z(const Z& z); // 3
        bool operator==(T1* t1, T2& t2) const;
};
```

Scrivere un template di classe `SmartP<T>` di **puntatori smart** a `T` che definisca assegnazione profonda, costruzione di copia profonda e distruzione profonda di puntatori smart. Il template `SmartP<T>` dovrà essere dotato di una interfaccia pubblica che permetta di **compilare correttamente** il seguente codice, la cui esecuzione dovrà **provocare esattamente** le stampe riportate nei commenti.

```
class C {
public:
  int* p;
  C(): p(new int(5)) {}
};

int main() {
  const int a=1; const int* p=&a;
  SmartP<int> r; SmartP<int> s(&a); SmartP<int> t(s);
  cout << *s << " " << *t << " " << *p << endl; // 1 1 1
  *s=2; *t=3;
  cout << *s << " " << *t << " " << *p << endl; // 2 3 1
  r=t; *r=4;
  cout << *r << " " << *s << " " << *t << " " << *p << endl; // 4 2 3 1
  cout << (s == t) << " " << (s != p) << endl; // 0 1
  C c; SmartP<C> x(&c);
  cout << *(c.p) << " " << *(x->p) << endl; // 5 5
  *(c.p)=6;
  cout << *(c.p) << " " << *(x->p) << endl; // 6 6
  SmartP<C>* q = new SmartP<C>(&c);
  delete q;
  cout << *(x->p) << endl; // 6
}
```