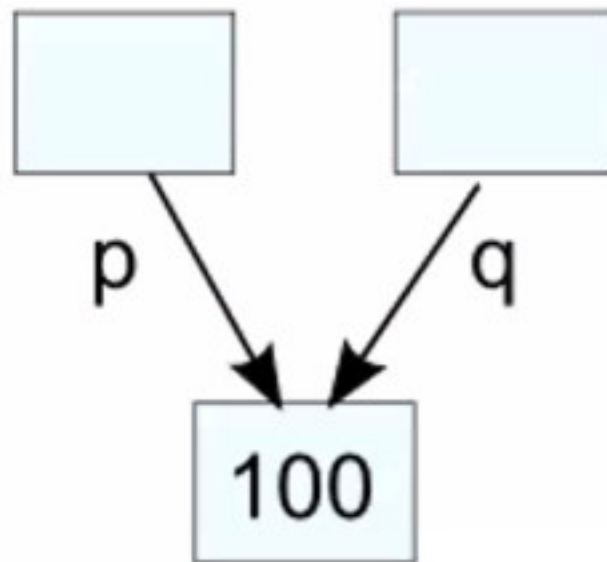


Uhh, Houston we
have a problem...
again.

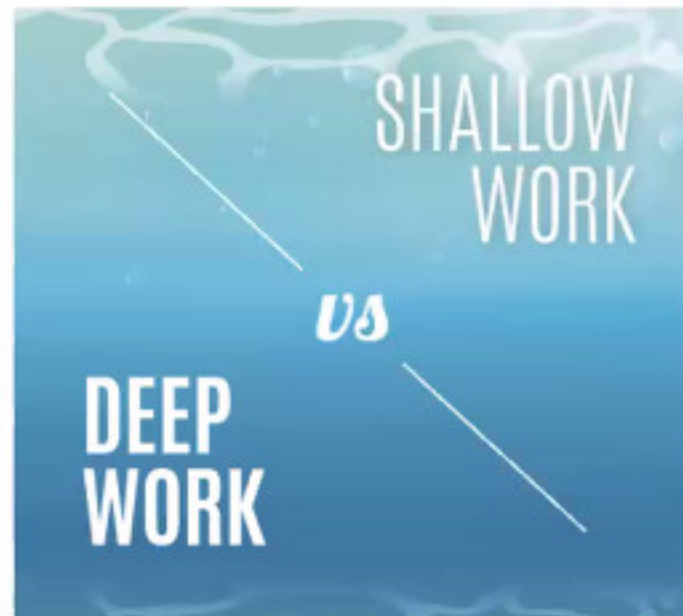
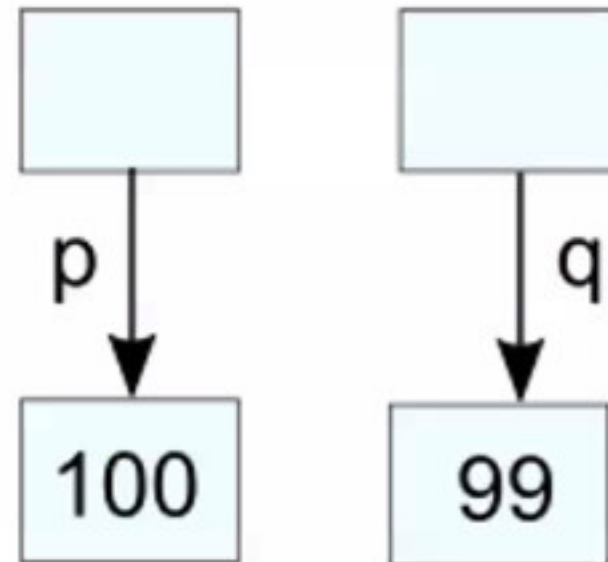
Go ahead
student.



Shallow Copy



Deep Copy



Fenomeno dell'**interferenza** o **aliasing**.

L'assegnazione standard esegue una *shallow copy*. Idem il costruttore di copia standard.

Shallow copy [\[edit\]](#)

One method of copying an object is the shallow copy. In the process of shallow copying A, B will copy all of A's field values.^{[\[1\]](#)[\[2\]](#)[\[3\]](#)[\[4\]](#)} If the field value is a memory address it copies the memory address, and if the field value is a primitive type it copies the value of the primitive type.

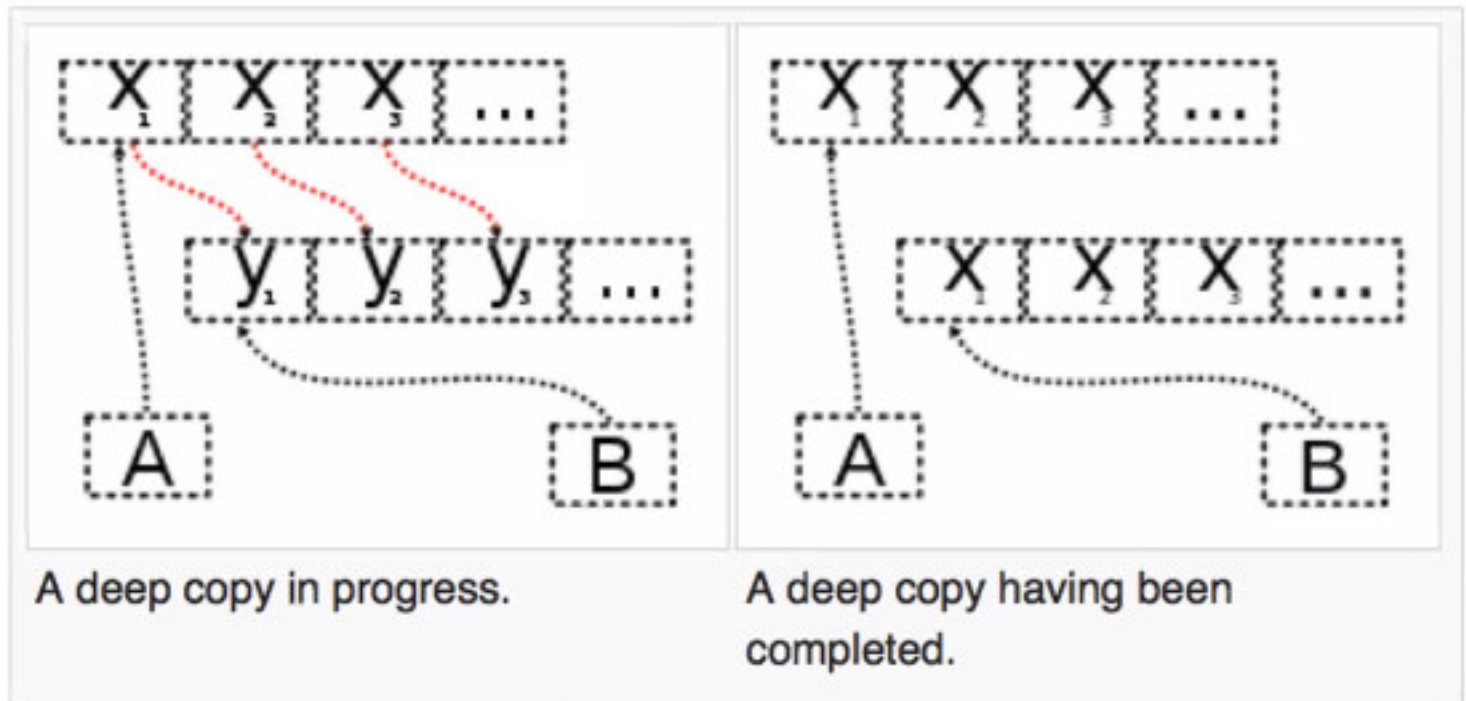
The disadvantage is if you modify the memory address that one of B's fields point to, you are also modifying what A's fields point to.

Copie profonde per il costruttore di copia e l'assegnazione

Deep copy [\[edit\]](#)

An alternative is a deep copy. Here the data is actually copied over. The result is different from the result a shallow copy gives. The

advantage is that A and B do not depend on each other but at the cost of a slower and more expensive copy.



Ridefinizione dell'assegnazione in `bolletta`

```
class bolletta {  
public:  
    bolletta& operator=(const bolletta& y);  
    ...  
};
```

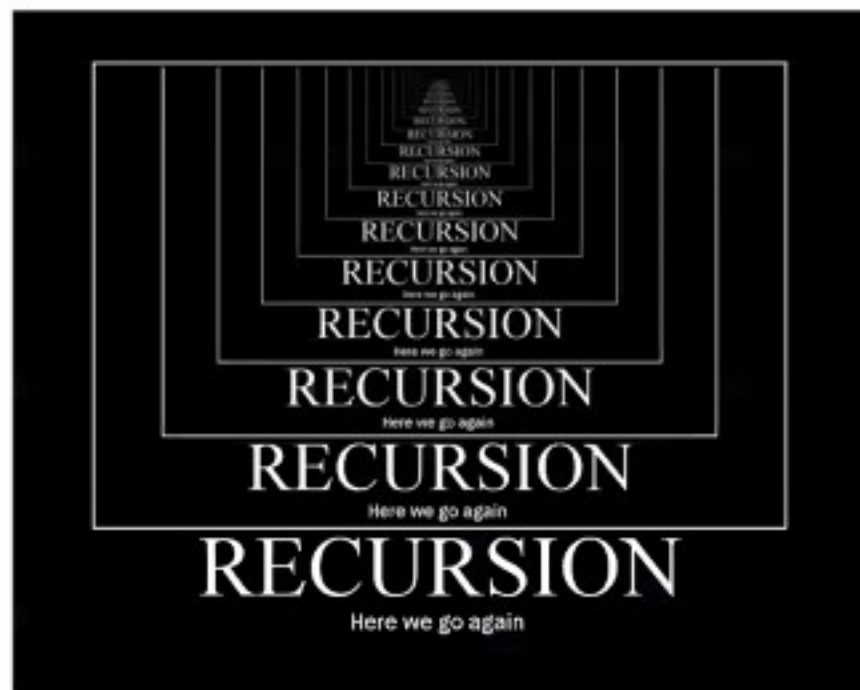
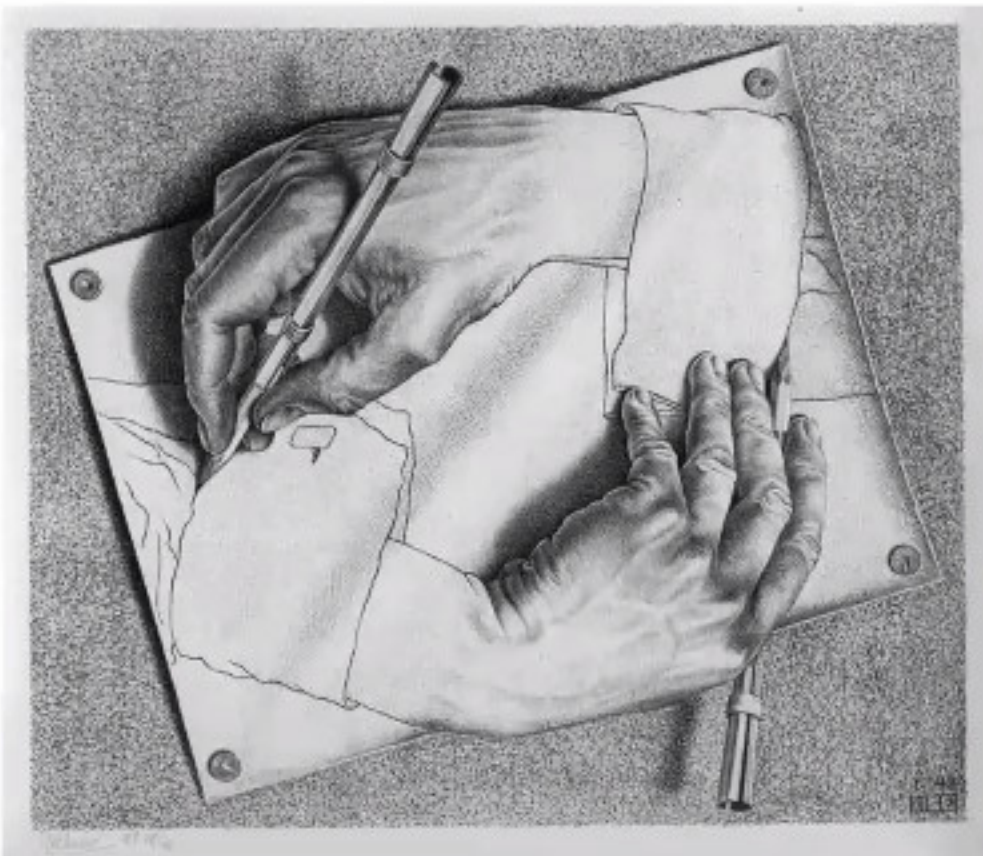

Aggiungiamo a `bolletta` due *metodi statici* di utilità che dichiariamo privati

```
class bolletta {  
    ...  
private:  
    static nodo* copia(nodo*) ;  
    static void distruggi(nodo*) ;  
    ...  
};
```

La definizione di questi due metodi è
un **esercizio facile** di Programmazione.

```
// implementazione iterativa
bolletta::nodo* bolletta::copia(nodo* p) {
    if (!p) return 0;
    nodo* primo = new nodo(p->info,0);
    // primo punta al primo nodo della copia della lista
    nodo* q = primo;
    // q punta all'ultimo nodo della lista finora copiata
    while (p->next) {
        p = p->next;
        q->next = new nodo(p->info,0);
        q = q->next;
    }
    return primo;
}
```

```
// implementazione iterativa
void bolletta::distruggi(nodo* p) {
    if(p!=nullptr) {
        // scorro tutta la lista deallocando ogni nodo
        nodo* q = p;
        while (p!=nullptr) {
            p = p->next;
            delete q;    // dealloco il nodo puntato da q
            q = p;
        }
    }
}
```

Implementazioni *ricorsive*.

```
bolletta::nodo* bolletta::copia(nodo* p) {  
    if (!p) return 0; // caso base: lista vuota  
    else  
        // passo induttivo:  
        // per induzione copia(p->next) è la copia della coda  
        // di p, e quindi inserisco una copia del primo nodo  
        // di p in testa alla lista copia(p->next)  
        return new nodo(p->info, copia(p->next));  
}
```

```
void bolletta::distruggi(nodo* p) {  
    // caso base: lista vuota, nulla da fare  
    if (p) {  
        // passo induttivo:  
        // per induzione distruggi(p->next) dealloca  
        // la coda di p, e quindi rimane da deallocare  
        // solamente il primo nodo di p  
        distruggi(p->next);  
        delete p; // dealloco il nodo puntato da p  
    }  
}
```

Primo tentativo di definizione di operator=

```
bolletta& bolletta::operator=(const bolletta& b) {  
    first = copia(b.first); // operator= tra puntatori  
    return *this; // ritorna l'oggetto di invocazione  
};
```

BUT...



(1) memoria puntata da first prima della
sua assegnazione



```
bolletta& bolletta::operator=(const bolletta& b) {  
    distruggi(first); // pulizia dello heap  
    first = copia(b.first);  
    return *this;  
}
```





**che
succede?**

(1) memoria puntata da first prima della sua assegnazione

(2) possibile assegnazione $b = b$; con:

```
bolletta& bolletta::operator=(const bolletta& b) {  
    distruggi(first); // pulizia dello heap  
    first = copia(b.first);  
    return *this;  
}
```



```
bolletta&bolletta::operator=(const bolletta& b) {  
    if (this != &b) {    // != tra puntatori  
        distruggi(first); // pulizia dello heap  
        first = copia(b.first);  
    }  
    return *this;  
}
```

Ridefinizione del costruttore di copia

```
class bolletta {  
public:  
    bolletta(const bolletta&) ;  
    ...  
};
```

```
bolletta::bolletta(const bolletta& b) :  
    first(copia(b.first)) {}
```

Definiamo una funzione esterna

`orario Somma_Durate(bolletta b)`

che restituisca la somma delle durate delle telefonate in `b`.

```
orario Somma_Durate(bolletta b) { // NOTA: b passato per valore
    orario durata; // costruttore di default di orario
    while (!b.Vuota()) {
        // estrae dal primo nodo della lista
        telefonata t = b.Estrai_Una();
        durata = durata + t.Fine() - t.Inizio();
    } // vincolo: durata < 24 ore !
    return durata;
}
```



```
int main() {  
    bolletta b1;  
  
    telefonata t1(orario(9,23,12),orario(10,4,53),2121212);  
    telefonata t2(orario(11,15,4),orario(11,22,1),3131313);  
  
    b1.Aggiungi_Telefonata(t2);  
    b1.Aggiungi_Telefonata(t1);  
  
    cout << b1;  
  
    cout << "LA SOMMA DELLE DURATE è "  
        << Somma_Durate(b1) << endl;  
  
    cout << b1;  
}
```


OUTPUT

TELEFONATE IN BOLLETTA:

- 1) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
- 2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

LA SOMMA DELLE DURATE è 0:38:38

TELEFONATE IN BOLLETTA:

- 1) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
- 2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

WHY?!!





- (1) **b1** è passato per valore in **Somma_Durate(b1)**
- (2) **b** oggetto locale a **Somma_Durate()** è una copia profonda di **b1**
- (3) **Estrai_Una()** provoca side effects su **b**

```
orario Somma_Durate(bolletta b) {  
    // NOTA: b passato per valore  
    orario durata; // costruttore di default di orario  
    while (!b.Vuota()) {  
        // estrae dal primo nodo della lista  
        telefonata t = b.Estrai_Una();  
        durata = durata + t.Fine() - t.Inizio();  
    } // vincolo: durata < 24 ore !  
    return durata;  
}
```

La funzione esterna `Chiamate_A` rimuove dalla bolletta `b` passata per riferimento tutte le telefonate a `num` e restituisce una nuova bolletta contenente le telefonate tolte.

```
bolletta Chiamate_A(int num, bolletta& b) {  
    bolletta selezionate, resto; // oggetti locali  
    while (!b.Vuota()) {  
        telefonata t = b.Estrai_Una();  
        if (t.Numero() == num)  
            selezionate.Aggiungi_Telefonata(t);  
        else  
            resto.Aggiungi_Telefonata(t);  
    }  
    b = resto; // overloading di operator= in bolletta  
    return selezionate;  
}
```



```
int main() {  
    bolletta b1;  
    telefonata t1(orario(9,23,12),orario(10,4,53),2121212);  
    telefonata t2(orario(11,15,4),orario(11,22,1),3131313);  
    telefonata t3(orario(12,17,5),orario(12,22,8),2121212);  
    telefonata t4(orario(13,46,5),orario(14,0,33),3131313);  
    b1.Aggiungi_Telefonata(t4);  
    b1.Aggiungi_Telefonata(t3);  
    b1.Aggiungi_Telefonata(t2);  
    b1.Aggiungi_Telefonata(t1);  
    cout << b1;  
    bolletta b2 = Chiamate_A(2121212, b1);  
    cout << b1;  
    cout << b2;  
}
```

OUTPUT

```
bolletta b2 = Chiamate_A(2121212, b1);
```

TELEFONATE IN BOLLETTA: (NB: b1)

- 1) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
- 2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313
- 3) INIZIO 12:17:5 FINE 12:22:8 NUMERO 2121212
- 4) INIZIO 13:46:5 FINE 14:0:33 NUMERO 3131313

TELEFONATE IN BOLLETTA: (NB: b1)

- 1) INIZIO 13:46:5 FINE 14:0:33 NUMERO 3131313
- 2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

TELEFONATE IN BOLLETTA: (NB: b2)

- 1) INIZIO 12:17:5 FINE 12:22:8 NUMERO 2121212
- 2) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212

BUT...


```
bolletta Chiamate_A(int num, bolletta& b) {  
    bolletta selezionate, resto;  
    while (!b.Vuota()) {  
        telefonata t = b.Estrai_Una();  
        if (t.Numero == num)  
            selezionate.Aggiungi_Telefonata(t);  
        else  
            resto.Aggiungi_Telefonata(t);  
    }  
    b = resto;  
    return selezionate;  
}
```

I due oggetti locali **selezionate** e **resto** esistono solo durante l'esecuzione della funzione **Chiamate_A()**.





(1) memoria di **resto** e **selezionate** non è deallocata

(2) **return selezionate**: oggetto ritornato costruito di copia, la memoria non viene deallocata



*Lifetime*TM

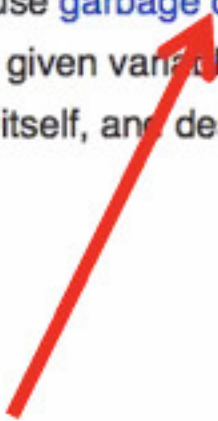


Object lifetime

From Wikipedia, the free encyclopedia

In [object-oriented programming](#) (OOP), the **object lifetime** (or **life cycle**) of an [object](#) is the time between an object's creation and its destruction. Rules for object lifetime vary significantly between languages, in some cases between implementations of a given language, and lifetime of a particular object may vary from one run of the program to another.

In some cases object lifetime coincides with [variable lifetime](#) of a variable with that object as value (both for [static variables](#) and [automatic variables](#)), but in general object lifetime is not tied to the lifetime of any one variable. In many cases – and by default in many [object-oriented languages](#) (OOLs), particularly those that use [garbage collection](#) (GC) – objects are allocated on the [heap](#), and object lifetime is not determined by the lifetime of a given variable: the value of a variable holding an object actually corresponds to a [reference](#) to the object, not the object itself, and destruction of the variable just destroys the reference, not the underlying object.



Static variable

From Wikipedia, the free encyclopedia



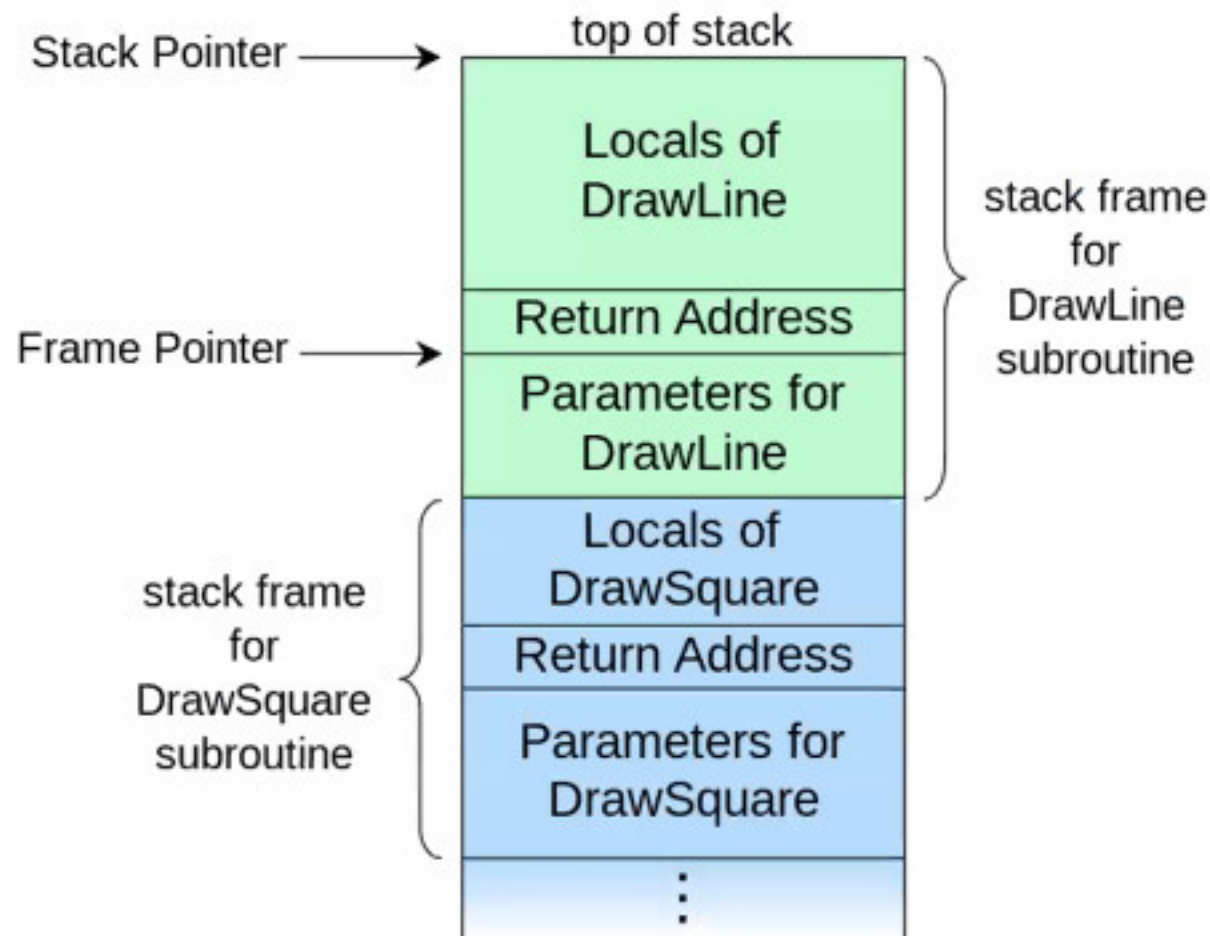
It has been suggested that *Static memory allocation* be merged into this article.
([Discuss](#)) *Proposed since January 2013.*

See also: *Static (keyword)*

In [computer programming](#), a **static variable** is a [variable](#) that has been [allocated statically](#)—whose [lifetime](#) or "extent" extends across the entire run of the program. This is in contrast to the more ephemeral [automatic variables](#) ([local variables](#) are generally automatic), whose storage is allocated and deallocated on the [call stack](#); and in contrast to objects whose storage is [dynamically allocated](#) in [heap memory](#).

Tempo di vita di una variabile

(1) Variabili di classe automatica (**Call stack**)



Tempo di vita di una variabile

- (1) Variabili di classe automatica (**Call stack**)
- (2) Variabili di classe statica (**Memoria statica**)
 - campi dati statici (P2)
 - variabili globali (P1, almost deprecated)
 - variabili statiche in corpo di funzione (bad practice)
- (3) Variabili dinamiche (**Heap**)

