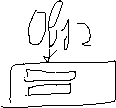
In C++, variabilele nu stocheaza, in mod implicit, referinte catre obiecte. Variabilele sunt direct legate, mereu, de zona de memorie a obiectului respectiv, asemanator ca in Python. Ele anume asta si reprezinta in C++, o zona de memorie de care sunt legate.De aceea, cand se egaleaaza 2 variabile ce pastreaza obiecte sau date primitive, pur si simplu variabila din stanga va face o copie a obiectului de care este legata variabila din dreapta. In Java, variabilele pastreaza referinte la obiectele de care sunt legate.

In C++ egalarea a doua variabile ce sunt legate de obiecte va face ca variabila din stanga de egal pur si simplu sa copie obiectul variabilei din dreapta de egal.

Type obj1;

Type obj2;



Deci, fie ca variabilele stocheaza date primitive sau obiecte, orice operatie de egalare duce doar la crearea unei copii, nu la legarea variabile de obiectul vaiabilei din dreapat de egal.

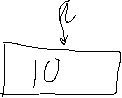
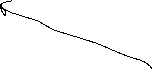
Totusi, in C++ exista ceva nu au nici Java, nici Python. C++ ne permite sa cream variabile ce stocheaza referinta, ca si Java, doar ca exista anumite deosebiri. Operatorul & se comporta diferit in diferite situatii:

**1.**

int a = 10;

int &b = a;

Aceste instructiuni declara o variabila a, ce stoheaza in adresa ei de memorie valoarea 10, si mai apoi creaza o variabila b, dar care nu pur si simplu va copia valoarea lui a. Atunci cand se adauga & in momentul declararii variabilei, variabila noua nu va stoca in ea o valoare, ci o referinta la alt variabila! Referinta nu este locatia! Referinta este doar o legatura catre o alta variabila, asa incat daca folosim variabila ce stocheaza o referinta, vom afecta direct variabila catre care ea are referinta.



Insa, acest mod de a crea variabile ce pastreaza referinte are anumite limitari:

* Nu pot avea referinta null
* Referinta, odata setata la momentul crearii, nu mai poate fi modificata
* Trebuie sa oferim neaparat o variabila catre care sa se creeze referinta la momentul crearii unei astfel de variabile.

**int** arata pur si simplu ca variabila va stoca o valoare

**int &** arata ca ea va stoca o referinta la o variabila int

**2.**

int a = 20;

int \*b = &a;

Daca & este utilizat de o variabila care deja este declarata si are un spatiu de memorie deja creat, acest operator va trimite deja locatia ei de memorie. Pointerul poate sa stocheze locatii de memorie, dar nu si variabilele! De aceea, o declaratie ca:

int a = 20;

~~int &b = &a;~~

va genera eroare! Variabila b are nevoie de variabila concreta catre care sa faca referinta, nu de adresa ei de memorie!

3.

Daca o functie returneaza o referinta, lucrurile stau diferit.

De ex:

int& func() {

static int b = 1000;

cout << &b << endl;

return b;

}

int main()

{

int a = func();

cout << &a << endl;

cout << a << endl;

return 0;

}

a nu va avea aceeasi locatie ca b! a aici e declarant ca variabila ce poate stoca valori, nu referinte, de aceea ea va copia nu referinta, ci valoarea referintei, si nu va fi nici-o eroare, caci functia ce returneaza & returneaza referinta, nu adresa de memorie, si variabila a poate alege in acest caz daca sa copie referinta, sau valoarea. Dar in acest caz ea va copia valoarea. Daca o variabila deja definite are in fata &, se returneaza adresa de memorie, dar functia nu asta face! Functia returneaza anume referinta variabilei din return, nu adresa ei!!! Adica,

**int a = func()** e echivalent cu **int a = b;** (b din func!)

Daca functia ar returna int, nu int&, atunci ar fi echivalent cu:

int a = 1000; adica se va trimtie valoarea lui b, nu b

int &a = func();

In acest caz deja, var a va stoca referinta returnata de functie.