##### **Pointeri în C++**

**Prezentare generală**

Când *declarăm o variabilă de un tip oarecare* în C++, acestei variabile i se atribuie o locație specifică din memoria operativă, necesară pentru păstrarea **valorii variabilei**.

Orice variabilă ocupă în memorie un anumit *număr de octeți consecutivi* (în funcție de t*ipul variabilei*); *adresa primului octet* alocat unei variabile reprezintă **adresa variabilei**.

Pointerii din C++ sunt variabile speciale care stochează adresa de memorie ale altor variabile. Pointerii adaugă mai multe caracteristici și flexibilitate limbajului de programare C++.

**Noțiuni de bază**

* Semnificația pointerilor, diferite tipuri de pointeri în C++ și modul de utilizare a acestora.
* Aritmetica pointerilor, pointerii la tablouri și constantele de tip pointer.

**Ce sunt pointerii?**

**Definiție**. ***Un pointer reprezintă un tip de dată a cărei valoare este o adresă de memorie***.

**Important**: *Nu confundați adresa unei variabile cu valoarea memorată de aceasta. Ele sunt de regulă diferite, chiar și în cazul pointerilor!*

În limbajul C++ o variabilă de tip pointer stochează **adresa** (sau locația de memorie) a unei alte variabile. Cu alte cuvinte, un pointer indică adresa altei variabile. La fel ca variabilele obișnuite, pointerii în C++ sunt de tipuri diferite de date. Un pointer ar trebui să aibă același tip de date cu cel al variabilei către care indică.

**Operator de referențiere și operator de dereferențiere**

Între pointeri și operatorii de referențiere și dereferențiere există legături strânse.

**Operator de referențiere &**

Operatorul de **referențiere****&** returnează adresa oricărei variabile statice (inclusiv a pointerilor).

**De exemplu:**

**float** a = 23.4;

*// folosim operatorul de referențiere*

cout << &a;

**Ieșire:**

0x62fe1c

**Observație:** *Adresa fiecărei variabile este diferită în sisteme diferite.*

Pointerii în C++ sunt folosiți pentru a stoca adrese ale variabilelor. Pentru a stoca adresa oricărei variabile într-un pointer, folosim operatorul de referențiere **&**. Cu alte cuvinte, putem atribui adrese pointerilor folosind operatorul de referențiere.

**Exemplu:**

**int** var= 10;

**int**\* ptr;

ptr = &var;

cout << "Valoarea lui &var este: " << &var<< endl;

cout << "Valoarea lui ptr este: " << ptr << endl;

**Ieșire:**

Valoarea lui &var este: 0x62fe10

Valoarea lui ptr este: 0x62fe10

Prin urmare, variabila de tip pointer **ptr** conține acum adresa variabilei **var**.

**Operator de dereferențiere \***

Simbolul asterisc **\*** se numește ***operator de dereferențiere*** atunci când este folosit cu date de tip pointer.

Putem accesa *valorile* stocate într-o variabilă către care indică pointerul, utilizând *numele pointerului* și *operatorul de dereferențiere*.

Relativ la pointeri, simbolul asterisc **\*** are două semnificații diferite.

* Când simbolul **\*** este folosit într-o *declarație de variabilă*, valoarea scrisă în partea dreaptă a semnului **=** ar trebui să fie adresa unei variabile (prezentă în memorie).
* Atunci când operatorul unar **\*** este utilizat cu un pointer ne permite să găsim sau să atribuim valoarea stocată în locația de memorie referită de de pointer. Operatorul unar **\*** poate fi citit ca **„valoare indicată de”**.

***Exemplu*:**

**int** var= 10;

**int**\* ptr;

ptr = &var;

cout << "Valoarea returnata de &var este: " << &var<< endl;

cout << "Valoarea returnata de ptr este: " << ptr << endl;

*// folosim dereferentierea \*ptr*

cout << "Valoarea returnata de \*ptr este: " << \*ptr << endl;

**Ieșire:**

Valoarea returnata de &var este: 0x62fe10

Valoarea returnata de ptr este: 0x62fe10

Valoarea returnata de \*ptr este: 10

După cum putem observa, pentru a obține valoarea variabilei **var**, am folosit **\*ptr**.

**Sintaxa declarației pointerului**

Pointerii în C++ sunt declarați folosind următoarea sintaxă:

**tip\_data \*nume\_pointer;**

**tip\_data\* nume\_pointer;**

**tip\_data \* nume\_pointer;**

Folosim simbolul asterisc **\*** pentru a declara o **variabilă de tip pointer**.

Dacă trebuie să declarăm *două* sau mai mulți pointeri împreună în aceeași linie, va trebui să folosim simbolul asterisc înaintea fiecărui nume de variabilă.

***Exemplu*:**

**int** \*var; *// var este un pointer la intreg*

**int**\* var1, \*var2; *// var1 si var2 sunt pointeri*

**int**\* var1, var2; *// var1 - pointer, var2 – variabila de tip intreg*

**Cum se utilizează pointerii în C++?**

Trebuie să urmăm câțiva pași pentru a folosi pointerii în C++:

1. Declarați o variabilă de tip pointer.
2. Atribuiți adresa altei variabile pointerului declarat, folosind operatorul **&**.
3. Accesați valoarea memorată la adresă folosind operatorul de dereferențiere **\***.

**Simboluri utilizate în indicatori**

Următorul tabel prezintă simbolurile care sunt utilizate cu pointerii.

|  |  |  |
| --- | --- | --- |
| **Simbol** | **Nume** | **Descriere** |
| **&** | Operator de referențiere | Folosit pentru a afla adresa unei variabile |
| **∗** | Operator de dereferențiere | Folosit pentru a accesa valoarea la o adresă |

**Exemplu:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

**int** x = 15;

**int** \*ptr;

ptr = &x;

cout << "Valoarea initiala a lui x este: " << x << endl;

cout << "Valoarea initiala a lui \*ptr este: " << \*ptr << endl;

*// modific valoarea lui x folosind pointerul ptr*

\*ptr = 250;

cout << "Valoarea noua a lui \*ptr este: " << \*ptr << endl;

cout << "Valoarea noua a lui x este: " << x << endl;

**return** 0;

}

**Ieșire:**

Valoarea initiala a lui x este: 15

Valoarea initiala a lui \*ptr este: 15

Valoarea noua a lui \*ptr este: 250

Valoarea noua a lui x este: 250

În exemplul de mai sus, am folosit operatorul de referențiere **&** pentru a memora adresa lui **x** în pointerul **ptr**. Apoi, am modificat valoarea variabilei **x** folosind operatorul de dereferențiere **\*** cu pointerul (**\*ptr**).

**Transmitere de argumente unei funcții utilizator**

În limbajul C++, *comunicarea dintre un program și subprogramele sale* se poate face în diferite moduri. În cazul unui apel de subprogram, putem transmite argumente funcțiilor în trei moduri diferite:

* Apel prin **valoare**.
* Apel prin referință cu argument de **tip pointer**.
* Apel prin referință cu argument de **tip referință**.

**Apel prin valoare**

În mod implicit, C++ utilizează metoda apel după valoare. Această metodă *copiază valoarea reală* a unui argument în parametrul funcției. Deci, dacă parametrul din interiorul funcției este modificat, acesta nu va afecta argumentul.

***Exemplu*:**

**#include** <iostream>

**using namespace** std;

*// Comunicare prin parametru formal valoare*

**int** triplu(**int** x)

{

*// adresa lui x in triplu() difera de adresa lui x in main()*

cout << "Adresa lui x in triplu(): " << &x << endl;

x = 3\*x;

**return** x;

}

**int** main()

{

**int** x = 10;

cout << "Valoarea lui x in main(): " << x << endl;

cout << "Adresa lui x in main(): " << &x << endl;

triplu(x);

cout << "Valoarea lui x dupa executarea functiei triplu(): " << x;

**return** 0;

}

**Ieșire:**

Valoarea lui x in main(): 10

Adresa lui x in main(): 0x62fe1c

Adresa lui x in triplu(): 0x62fdf0

Valoarea lui x dupa executarea functiei triplu(): 10

Din exemplul de mai sus, putem observa că adresa variabilei **x** a fost diferită în cadrul funcției **triplu()**. De asemenea, modificarea lui **x** în cadrul funcției **triplu()** nu a avut niciun impact asupra variabilei **x** din funcția de bază **main()**.

**Apel prin referință cu argument de tip pointer**

În *apelul prin referință cu argument de pointer*, folosim adresa variabilelor în calitate de parametri actuali la apelul unei funcții. De aceea valorile inițiale ale variabilelor se vor modifica la revenire din apel.

***Exemplu*:**

**include** <iostream>

**using namespace** std;

*// Comunicare prin parametru formal de tip pointer*

**int** triplu(**int** \*x)

{

*// x este un pointer.*

*// Adresa lui x in functia triplu() este aceeasi cu adresa lui x in main()*

cout << "Adresa lui x in triplu(): " << &x << endl;

\*x=3\* \*x;

}

**int** main()

{

**int** x = 10;

cout << "Valoarea lui x in main(): " << x << endl;

cout << "Adresa lui x in main(): " << &x << endl;

*// transmitem adresa lui x functiei triplu()*

triplu(&x);

cout << "Valoarea lui x dupa executarea functiei triplu(): " << x;

**return** 0;

}

**Ieșire:**

Valoarea lui x in main(): 10

Adresa lui x in main(): 0x62fe1c

Adresa lui x in triplu(): 0x62fdf0

Valoarea lui x dupa executarea functiei triplu(): 30

În exemplu anterior am folosit apelul prin referință, de aceea adresa variabilei **x** a fost aceeași în ambele funcții **triplu()** și **main()**. Aceasta înseamnă că variabilele **x** din ambele funcții au aceeași adresă sau locație de memorie. De aceea, modificarea valorii lui **x** în funcția **triplu()** a dus la modificarea lui **x** în **main()**.

**Apel prin referință cu argument de tip referință**

În *apelul prin referință cu argument de tip referință*, transmitem adresa variabilelor drept argumente. Există o singură diferență între cele două tipuri de *apeluri prin referințe* . *Apelul prin referință cu argument pointer* ia pointeri (care indică către locația de memorie a argumentelor) ca parametri ai funcției. Pe de altă parte, *apelul prin referință cu argument de referință* ia variabila originală în sine (nu o copie a variabilelor) ca parametri ai funcției.

Pentru a trece variabila originală ca parametru al unei funcții, folosim operatorul de referință (**&**) în declarația parametrilor unei funcții.

***Exemplu*:**

**#include** <iostream>

**using namespace** std;

*// Comunicare prin parametru formal de tip referinta (adresa)*

**int** triplu(**int**& x)

{

*// x este o variabila de tip intreg*

*// Adresa lui x in functia triplu() este IDENTICA cu adresa lui x din main()*

cout << "Adresa lui x in triplu(): " << &x << endl;

*// NU mai este necesara DEREFERENTIEREA*

x=3\*x;

}

**int** main()

{

**int** x = 10;

cout << "Valoarea lui x in main(): " << x << endl;

cout << "Adresa lui x in main(): " << &x << endl;

*// transmitem VALOAREA x functiei triplu()*

triplu(x);

cout << "Valoarea lui x dupa executarea functiei triplu(): " << x;

**return** 0;

}

**Ieșire:**

Valoarea lui x in main(): 10

Adresa lui x in main(): 0x62fe1c

Adresa lui x in triplu(): 0x62fe1c

Valoarea lui x dupa executarea functiei triplu(): 30

În exemplul de mai sus, așa cum am declarat **int &x** ca parametru în loc de **int x**, variabila originală **x** a fost transmisă ca argument în funcția **triplu()**.

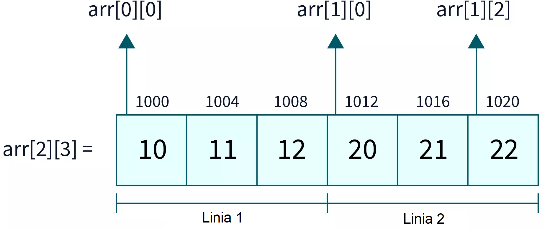
**Utilizarea avansată a variabilelor de tip pointer**

Putem accesa elementele unui tablou 2D folosind tipul de date pointer. Elementele unui tablou 2D sunt memorate în aceeași manieră ca și elementele unui tablou 1D.

Putem spune că un tablou 2D reprezintă o colecție de mai multe tablouri 1D plasate unul după altul.

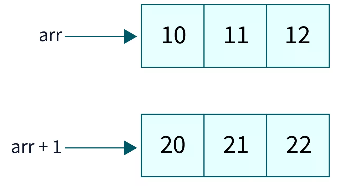
**De exemplu**, pentru următoarea declarație de tablou 2D, vom avea:

**int** arr[2][3] = {{10, 11, 12}, {20, 21, 22}}



În imaginea de mai sus, putem considera fiecare linie ca un tablou unidimensional. Deci, matricea **arr** conține **2** elemente în care fiecare element este o matrice 1D care conține câte **3** numere întregi.

Variabila **arr** conținea adresa (indică, pointează) către cele **3** elemente ale tabloului 1D. În mod similar, **arr+1** conține adresa primului element al tabloului 1D (vezi imaginile).



În general, putem spune că **arr+0** indică către elementul **0** al tabloului **arr**, unde elementul **arr+1** este un tablou 1D. Prin urmare, dereferențierea lui **arr+0** (adică **\*(arr+0)**) ne va da adresa elementului **0** (adresa de bază) prezent în tabloul 1D.

Dacă cunoaștem adresa de bază a elementului cu indice **0** din tabloul 1D, vom putea obține adresa al **m**-lea element al tabloului 1D folosind **\*(arr+0)+m**. În cele din urmă, dereferențierea **\*(arr+0)+m** , adică **\*(\*(arr+0)+m)**, ne va da valoarea unui element întreg din tabloului 2D.

Deci, în exemplul de mai sus, **\*(\*(arr+1)+2)** va returna valoarea **22**. În mod similar, **\*(\*(arr)+1)** va returna valoarea **11**.

În general, **arr[n][m]** este egal cu **\*(\*(arr+n)+m)**.

Cum putem accesa fiecare element al unui tablou bidimensional (2D) utilizând pointerii.

| **Accesare element** | **Accesare cu pointeri** | **Valoare** |
| --- | --- | --- |
| **arr[0][0]** | **\*(\*arr)** | **10** |
| **arr[0][1]** | **\*(\*arr+1)** | **11** |
| **arr[0][2]** | **\*(\*arr+2)** | **12** |
| **arr[1][0]** | **\*(\*(arr+1))** | **20** |
| **arr[1][1]** | **\*(\*(arr+1)+1)** | **21** |
| **arr[1][2]** | **\*(\*(arr+1)+2)** | **22** |

**Numele tabloului ca pointer**

Numele unui tablou acționează ca un pointer deoarece adresa primului element al unui tablou este stocată în *numele acestui tablou*. Deci, dacă un pointer conține adresa primului element al unui tablou, putem folosi acel pointer pentru a accesa restul elementelor tabloului.

***Exemplu*:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

**int** arr[3] = {1, 2, 3};

**int** \*ptr;

*// atribuirea adresei lui arr[0] unui pointer ptr*

ptr = arr;

*// ptr = &arr[0] sunt acuma identici*

cout << "Elementele tabloului sunt: "

<< ptr[0] << " " *// folosim ptr[0] in loc de arr[0]*

<< ptr[1] << " "

<< ptr[2] << endl;

**return** 0;

}

**Ieșire:**

Elementele tabloului sunt: 1 2 3

În exemplul de mai sus, am atribuit adresa **arr[0]** pointerului **ptr**. De aceea, putem accesa toate elementele tabloului folosind variabila de tip pointer.

**Aritmetica pointerilor și expresii cu pointeri**

Putem efectua doar un număr limitat de operații aritmetice pe pointeri în C++.

1. ***Incrementare*** (**++**)
2. ***Decrementare*** (**--**)
3. ***Adunare*** (**+**)
4. ***Scădere*** (**-**)

**Exemplul 1: Utilizarea operatorului de incrementare**

Când incrementăm un pointer folosind operatorul de **incrementare** (**++**), adresa pointerului se mărește. Incrementarea adresei pointerului este egală cu dimensiunea tipului său de date.

**n\*sizeof(TIP)**

Deoarece toate elementele matricei sunt stocate în memoria contiguă, putem folosi operatorul de incrementare pe pointeri pentru a accesa elementele unui tablou.

**#include** <iostream>

**using namespace** std;

**int** main ()

{

**int** arr[3] = {10, 20, 30};

*// memoreaza adresa lui arr intr-un pointer ptr*

**int** \*ptr = arr;

**for** (**int** i = 0; i < 3; i++)

{

cout << "Valoarea lui aar[" << i << "] este: " << \*ptr << endl;

cout << "Adresa lui aar[" << i << "] este: " << ptr << endl << endl;

*// pointeaza la urmatoarea locatie de memorie*

ptr++;

}

**return** 0;

}

**Ieșire:**

Valoarea lui aar[0] este: 10

Adresa lui aar[0] este: 0x62fe08

Valoarea lui aar[1] este: 20

Adresa lui aar[1] este: 0x62fe0c

Valoarea lui aar[2] este: 30

Adresa lui aar[2] este: 0x62fe10

În exemplul de mai sus, am folosit **ptr++** pentru a accesa următoarele elemente ale tabloului **arr**. Deoarece **ptr** este de tip pointer către **int**, adresa a fost mărită cu **4** (deoarece dimensiunea unui tip **int** este **4B**) când am folosit **ptr++**.

**Exemplul 2: Utilizarea operatorului de decrementare**

Operatorul de **decrementare** (**--**) este similar cu operatorul de incrementare. Operatorul de decrementare scade adresa unui pointer cu dimensiunea tipului său de date.

Operatorul de decrementare poate fi folosit și cu matrice pentru a accesa elementele acestora.

**#include** <iostream>

**using namespace** std;

**int** main ()

{

**int** arr[3] = {10, 20, 30};

*// memoreaza adresa lui arr intr-un pointer ptr*

**int** \*ptr = &arr[2];

**for** (**int** i = 2; i >= 0; i--)

{

cout << "Valoarea lui aar[" << i << "] este: " << \*ptr << endl;

cout << "Adresa lui aar[" << i << "] este: " << ptr << endl << endl;

*// pointeaza la locatia de memorie precedenta*

ptr--;

}

**return** 0;

}

**Ieșire:**

Valoarea lui aar[2] este: 30

Adresa lui aar[2] este: 0x62fe10

Valoarea lui aar[1] este: 20

Adresa lui aar[1] este: 0x62fe0c

Valoarea lui aar[0] este: 10

Adresa lui aar[0] este: 0x62fe08

În exemplul de mai sus, indicatorul **ptr** indica ultimul element al tabloului **arr**. Pentru a accesa fiecare element al vectorului folosind **ptr**, am folosit **ptr--** în bucla**for**.

**Exemplul 3: Adunarea și scăderea**

Dacă adăugăm valoarea **3** la un pointer (**ptr + 3**), pointerul va indica adresa de memorie situată cu 3 poziții mai departe decât adresa curentă. Cu alte cuvinte, pointerul va indica o adresă care este de trei ori mai mare decât tipul de date al pointerului:

**3\*dimensiunea\_tipului\_pointer**

Operația de scădere este similară cu adunarea. Dacă scădem o unitate din pointer **ptr-1**, pointerul va indica adresa de memorie precedentă celei curente.

**#include** <iostream>

**using namespace** std;

**int** main()

{

**int** arr[5] = {10, 20, 30, 40, 50};

**int** \*ptr1, \*ptr2;

// atribuim adresa primului element al tabloului arr

ptr1 = arr;

// atribuim adresa elementului al 5-lea al tabloului arr

ptr2 = &arr[4];

cout << "Valoarea lui ptr1 este: " << ptr1 << endl;

// adunarea

cout << "Valoarea lui ptr1+2 este: " << ptr1+2 << endl << endl;

cout << "Valoarea lui ptr2 este: " << ptr2 << endl;

// scaderea

cout << "Valoarea lui ptr2-1 este: " << ptr2-1 << endl << endl;

**return** 0;

}

**Ieșire:**

Valoarea lui ptr1 este: 0x28fef4

Valoarea lui ptr1+2 este: 0x28fefc

Valoarea lui ptr2 este: 0x28ff04

Valoarea lui ptr2-1 este: 0x28ff00

În exemplul de mai sus, **ptr1+2** este echivalent cu **&arr[2]**, iar **ptr2-1** este echivalent cu **&arr[3]**.

**Pointeri de constante**

C++ ne permite să folosim cuvântul cheie **const** în declarațiile unei date de tip pointer. Putem folosi cuvântul cheie **const** cu pointeri în trei moduri diferite:

* constantă de tip pointer
* pointer la constante
* constante pointeri către constante

**Constantă de tip pointer**

Dacă costantă de tip pointer, pointerul nu va putea modifica datele care sunt stocate într-o locație de memorie către care indică pointerul. Cu toate acestea, pointerul poate indica locația de memorie a unei alte variabile.

**Sintaxa** :

**const** tip \*nume\_pointer;

***Exemplu*:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

**int** var1 = 20;

**int** var2 = 100;

// declarare pointer la constanta

**const int** \*ptr;

// atribuire adresa var1 la ptr

ptr = &var1;

cout << "Valoarea lui \*ptr este: " << \*ptr << endl;

// atriibuie adresa lui var2 la ptr

ptr = &var2;

cout << "Valoarea noua a lui \*ptr este: " << \*ptr << endl;

// Linia urmatoare va provoaca eroare:

// \*ptr = 5;

**return** 0;

}

**Ieșire:**

Valoarea lui \*ptr este: 20

Valoarea noua a lui \*ptr este: 100

În exemplul de mai sus, am declarat un „*pointer către constantă*” **\*ptr**.

În acest mod, am putea stoca adrese ale diferitelor variabile în **ptr**, dar dacă încercăm să schimbăm valoarea la care indică **ptr** folosind *dereferențierea* (**\*ptr**), atunci compilatorul va genera o eroare.

**2. Pointeri la constante**

Pointerii indică către o locație fixă de memorie. Cu alte cuvinte, dacă un pointer la constantă stochează adresa unei variabile, nu putem folosi acel pointer pentru a stoca adresa altei variabile. Cu toate acestea, putem schimba valoarea variabilei către care indică pointerul.

**Sintaxă:**

tip \***const** nume\_pointer;

**De exemplu:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

**int** var1 = 10;

**int** var2 = 55;

// pointer constant

**int**\* **const** ptr = &var1;

cout << "Valoarea initiala var1: " << \*ptr << endl;

// modificam valoarea lui var1 folosind ptr

\*ptr = 11;

cout << "Valoarea noua a lui var1: " << \*ptr << endl;

// Eroare!

// ptr = &var2;

**return** 0;

}

**Ieșire:**

Valoarea initiala var1: 10

Valoarea noua a lui var1: 11

În exemplul de mai sus, am declarat un „pointer constant” **ptr**. Am putea schimba valoarea variabilei **ptr** la o altă adresă, dar dacă încercăm să stocăm o altă adresă în **ptr**, compilatorul va genera o eroare.

**3. Constantă de tip pointer la constantă**

Pointerul va indica către o constantă, iar pointerul în sine este constant (adică nu poate indica altundeva). Deci, nu putem modifica valoarea variabilei spre care indică și nici măcar nu putem folosi pointerul pentru a stoca adresa unei noi variabile.

**Sintaxă:**

**const** tip \***const** nume\_pointer;

**De exemplu:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

// declarare constante

**const int** var1 = 10;

**const int** var2 = 55;

// declarare constanta de tip pointer la constant

**const int**\* **const** ptr = &var1;

// \*ptr = 12;

// Eroare: assignment of read-only

// ptr = &var2;

// Eroare: assignment of read-only

// Adresa lui var1

cout << "Adresa lui var1: " << ptr << endl;

// Valoarea lui var1

cout << "Valoarea lui var1: "<< \*ptr;

**return** 0;

}

**Ieșire:**

Adresa lui var1: 0x28ff04

Valoarea lui var1: 10

În exemplul de mai sus, am declarat un „constantă de tip pointer la constantă” **ptr**. Din această cauză, nu am putut modifica valoarea variabilei către care indică **ptr** și nici măcar nu am putut stoca o adresă diferită în **ptr**.

**Pointeri către funcții**

La fel ca variabilele, funcțiile au și ele adrese. Deci, C++ oferă funcționalitate pentru a crea pointeri care pot stoca aceste adrese. Pointerii care pot stoca adresele funcțiilor se numesc ***pointeri la funcții***.

De obicei, pointerii către funcții sunt folosiți atunci când trebuie să transmitem o funcție ca argument unei alte funcții.

**Sintaxă:**

**tip returnat (\*nume\_pointer\_functie)(tip parametri);**

**Exemplu:**

**#include**<iostream>

**using namespace** std;

// definire functie utilizator

**int** square(**int** x)

{

**return** x\*x;

}

**int** main()

{

**int** num = 9;

**int** result;

// creare pointer la functie

**int** (\*fp)(**int**);

// memoreaza adresa functiei square() in pointerul fp

fp = square;

// foloseste pointerul fp pentru apelul functiei square()

result = (\*fp)(num);

cout << "Patratul lui 9 este: " << result;

**return** 0;

}

**Ieșire:**

Patratul lui 9 este: 81

În exemplul de mai sus, am creat un pointer către funcția **fp**. Am folosit acest pointer pentru a pentru a stoca adresa funcției **square()**. Apoi, am folosit pointerul **fp** însuși pentru a apela funcția și a calcula pătratul variabilei **num**.

**Pointeri și șiruri de caractere (literale)**

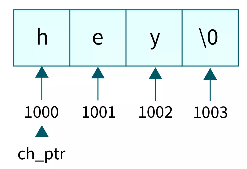
Șirurile de caractere sunt tablouri care conțin secvențe de caractere terminate cu **null** (**'\0'**). Fiecare element al unui șir este de tipul **const char**.

**De exemplu:**

**const char** \*ch\_ptr = "hey";

Șirul **"hei"** este un tablou de caractere. Pointerul **ch\_ptr** indică primul element al acestui tablou, adică **"h"**.

Dacă presupunem că **"hei"** este stocat în locațiile de memorie începând cu adresa **1000**, atunci putem reprezenta declarațiile ca:



După cum știm, *tablourile* și *pointerii* din C++ au același comportament în expresii, putem folosi **ch\_ptr** pentru a accesa caracterele șirului.

**De exemplu:**

**const char** \*ch\_ptr = "hey";

**char** ch1 = \*(ch\_ptr+1);

**char** ch2 = ch\_ptr[1];

În exemplul de mai sus, atât **ch1** cât și **ch2** indică același caracter al șirului.

**Pointer la pointer**

Un pointer către un alt pointer este un lanț de pointeri. Când definim un pointer către un pointer, primul pointer indică către al doilea pointer, iar al doilea pointer către variabila reală.

Pointer 1

Pointer 2

Variabilă

Adresă

Adresă

Pointer 1

Pentru a declara un pointer către un pointer, folosim un operator unar (**\***) pentru fiecare nivel de înlănțuire de pointeri.

**De exemplu:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

**float var** = 10;

cout << "Valoarea lui &var este: " << &**var** << endl << endl;

// declarare pointer

**float** \*ptr1;

// declarare pointer la pointer

**float** \*\*ptr2;

// atrubuie adresa lui var lui ptr1

ptr1 = &**var**;

cout << "Valoarea lui ptr1 este: " << ptr1 << endl;

cout << "Valoarea lui &ptr1 este: " << &ptr1 << endl << endl;

// atribuie adresa pointerului ptr1 lui ptr2;

ptr2 = &ptr1;

cout << "Valoarea lui ptr2 este: " << ptr2 << endl;

cout << "Valuoarea lui \*ptr2 este: " << \*ptr2 << endl << endl;

**return** 0;

}

**Ieșire:**

Valoarea lui &var este: 0x28ff08

Valoarea lui ptr1 este: 0x28ff08

Valoarea lui &ptr1 este: 0x28ff04

Valoarea lui ptr2 este: 0x28ff04

Valuoarea lui \*ptr2 este: 0x28ff08

În exemplul de mai sus, am creat o variabilă *var* și doi pointeri *ptr1* și *ptr2* . Adresa lui *var* a fost stocată în *ptr1* , în timp ce adresa lui *ptr1* a fost stocată în *ptr2* .

**Pointer de tip void**

Pointerii de tip **void** indică către o variabilă fără tip precizat de date. Acești pointeri pot fi transformați în orice tip de date și pot fi utilizați pentru a stoca adrese de orice tip.

Pointerii de tip **void** nu pot fi dereferențiați direct. Pentru a le dereferenția, trebuie să le convertim într-un alt tip de pointer care indică un anumit tip de date.

***Exemplu*:**

**#include** <iostream>

**using namespace** std;

**int** main()

{

**char** var1 = 'a';

**int** var2 = 1;

// declarare void pointer

**void** \*ptr;

ptr = &var1; // adresa lui var1 de tip char

ptr = &var2; // adresa lui var2 de tip int

// dereferenciere ptr fara conversie de tip

cout << \*ptr; // eroare!

// Asa merge!:

// cout << \*(int\*) ptr;

**return** 0;

}

**Ieșire:**

error: 'void\*' is not a pointer-to-object type

În exemplul de mai sus, am creat un pointer void **ptr**. Deoarece **ptr** era void, am putut să păstrăm adresa unei variabile **int** și **char**în **ptr**. Cu toate acestea, atunci când am încercat să dereferențiem **ptr**, compilatorul a generat o eroare, deoarece nu am convertit **ptr** la un anumit tip de date.

**Pointerul NULL**

Putem atribui constanta **NULL** unui pointer în C++. Valoarea **NULL** este zero. Un pointer căruia i se atribuie o valoare **NULL** se numește **pointer nul**.

**NULL** ne permite să creăm pointeri valizi, fără a stoca adresa vreunei variabile în pointer. Este recomandat să atribuiți **NULL** în timpul declarației pointerului. În caz contrar, compilatorul poate genera o eroare de executare.

**De exemplu:**

**#include** <iostream>

**using namespace** std;

**int** main ()

{

// defining a null pointer

**int** \*ptr = NULL;

cout << "Valoarea lui ptr: " << ptr;

**return** 0;

}

**Ieșire:**

Valoarea lui ptr: 0

**Greșeli frecvente când lucrați cu pointeri**

Câteva greșeli comune pe care utilizatorii le fac atunci când folosesc pointerii.

**int** \*ptr, var;

// Gresit

ptr = var;

// Corect

ptr = &var;

În exemplul de mai sus, **var** este o variabilă, *nu o adresă*. Deci, trebuie să scriem **&var** pentru a stoca adresa **var** în **ptr**.

**int** var= 10, \*ptr;

ptr = &var;

**var** = 20;

// Gresit

\*ptr = &var;

// Corect

\*ptr = var;

În exemplul de mai sus, **\*ptr** indică valoarea stocată în variabila **var**, în timp ce **&var** indică adresa **var**. Dacă vrem să stocăm valoarea lui **var** în **\*ptr**, trebuie să eliminăm **&** din **&var**.

**Avantajele utilizării indicatoarelor**

Următoarele sunt avantajele utilizării indicatorilor:

1. Putem aloca și de-aloca dinamic memorie folosind pointeri.
2. Pointerii sunt mai eficienți în manipularea tablourilor și a șirurilor de caractere.
3. Pointerii pot fi folosiți pentru a returna mai multe valori dintr-o funcție. Acest lucru se poate face prin trecerea argumentelor cu adresele lor și prin modificarea valorilor argumentului folosind pointeri.
4. Pointerii sunt eficienți în gestionarea structurilor dinamice de date, cum ar fi listele înlănțuite, arborii etc.