Seminar 1

Introducere

În acest seminar, vom discuta despre concepte introductive, mai specific diferențe dintre C și C++, care ne vor ajuta mai târziu.

Referințe și pointeri în C++

Referințele sunt o caracteristică specifică C++ care nu există în limbajul C. O referință reprezintă un *alias* sau un *nume alternativ* pentru o variabilă existentă. Toate operațiile aplicate asupra unei referințe acționează asupra obiectului la care aceasta face referire.

```
type &newName = existingName;
// or
type& newName = existingName;
// or
type & newName = existingName;
```

Adresa unei referințe este aceeași cu **adresa obiectului** pe care îl reprezintă. Toate referințele (cu excepția parametrilor de funcție) trebuie **inițializate** în momentul definirii. După ce este definită, o referință **nu poate fi reasignată** către un alt obiect, deoarece este un **alias**.

```
#include <iostream>

int main() {
   int number = 88;
   int & refNumber = number;
   std::cout << number << std::endl;
   std::cout << refNumber << std::endl;
   refNumber = 99;
   std::cout << refNumber << std::endl;
   std::cout << number << std::endl;
   number = 55;
   std::cout << number << std::endl;
   std::cout << refNumber << std::endl;
}
```

Nu putem avea referințe pentru:

- Alte referințe
- Câmpuri de biți (**bit fields**)

- Tablouri de referințe
- Pointeri către referințe
- NULL
- void
- obiecte/funcții invalide

O variabilă de tip **pointer** reține adresa din memoria calculatorului a unui obiect sau a unei funcții. Pointerii trebuie declarați înainte de a-i folosi,

```
type *ptr;   // Declare a pointer variable called ptr as a pointer
of type
// or
type* ptr;
// or
type * ptr;
```

```
int *p1, *p2, i;
int* p1, p2, i;
int * p1, * p2, i;
```

Când declarăm o variabilă de tip pointer, conținutul acesteia **nu este inițializat**. Cu alte cuvinte, ea conține o adresă de „undeva”, care, desigur, nu reprezintă o locație validă. Acest lucru este periculos! Trebuie să inițializezi un pointer atribuindu-i o adresă validă. Acest lucru se face, de obicei, folosind operatorul de adresa (&).

Operatorul de adresa (&) operează asupra unei variabile și returnează adresa acelei variabile. De exemplu, dacă `number` este o variabilă de tip `int`, `&number` returnează adresa variabilei `number`.

Se poate utiliza operatorul de adresa pentru a obține adresa unei variabile și a o atribui unei variabile pointer. De exemplu,

```
int number = 88;
int * pNumber;
pNumber = &number;
int * pAnother = &number;
```

Operatorul de **indirecționare** (sau operatorul de **dereferențiere**) (`*`) operează asupra unui pointer și returnează valoarea stocată la adresa păstrată în variabila pointer. De exemplu, dacă `pNumber` este un pointer la un `int`, `*pNumber` returnează valoarea `int` „la care pointează" `pNumber`.

```
int number_2 = 88;
int* pNumber_2 = &number;
std::cout << pNumber_2<< std::endl;
std::cout << *pNumber_2 << std::endl;
*pNumber_2 = 99;
std::cout << *pNumber_2 << std::endl;
std::cout << number_2 << std::endl;
```

De reținut că `pNumber` stochează o locație de adresă de memorie, în timp ce `*pNumber` se referă la valoarea stocată la adresa păstrată în variabila pointer, sau valoarea la care pointează pointerul. De asemenea, simbolul `*` are un sens diferit în declarațiile de variabile și în expresii. Când este folosit într-o declarație (de exemplu, `int *pNumber`), acesta indică faptul că numele care urmează este o variabilă de tip pointer. Pe de altă parte, când este folosit într-o expresie (de exemplu, `*pNumber = 99;` sau `temp << *pNumber;`), se referă la valoarea la care pointează variabila pointer.

Și pointerii au un tip !!!

```
int ii = 88;
double d = 55.66;
int* iPtr = &ii;
double* dPtr = &d;

iPtr = &d;
dPtr = &ii;
iPtr = ii;

int jj = 99;
iPtr = &jj;
```

Un pointer poate fi inițializat cu 0/NULL, dar dereferențierea lui produce excepția de `STATUS_ACCESS_VIOLATION:`

Referințe	Pointeri
Trebuie inițializate la declarare	Pot fi inițializați ulterior
Nu pot fi NULL	Pot avea valoarea NULL
Nu pot fi reasignate	Pot adresa diferite zone de memorie
Sintaxă mai clară și intuitivă	Necesită operatori de dereferențiere (*)

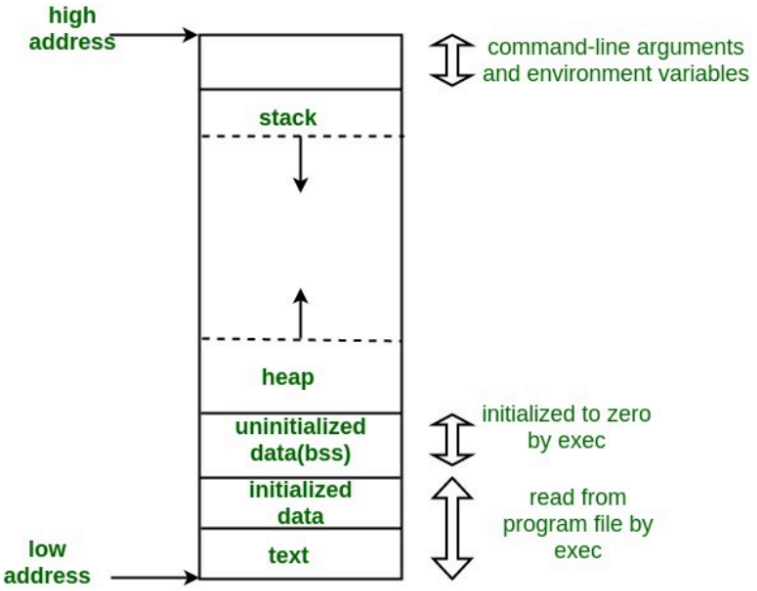
Transmiterea Parametrilor în Funcții

https://dev.to/erraghavkhanna/pass-by-value-reference-explained-with-single-gif-believe-me-it-s-true-23ki

```
void functie(int valoare, int *pointer, int &referinta) {
	valoare++;
	(*pointer)++;
	referinta++;
}

int main()
{
	int x = 0, y = 0, z = 0;
	functie(x, &y, z);
}
```

Alocarea Dinamică de Memorie



Reamintim diagrama cu așezarea unui program în memorie. Observăm că în această diagramă avem la dispoziție o zonă numită *heap* unde putem stoca datele. În această zonă de memorie putem aloca obiecte de dimensiune decisă la runtime, nu o dimensiune fixă (statică).

În C++, folosim ***new*** pentru a aloca o zonă de memorie. ***New*** va returna un pointer către adresa de memorie nou alocată. Este de datoria noastră să eliberăm acea zonă de memorie după ce nu o mai folosim, deoarece calculatorul nu face asta automat. Dacă nu eliberăm zonele de memorie pe care le alocăm dinamic este posibil să avem *memory leak-uri*, adică memorie care este alocată, dar nu mai este folosită niciodată. Chiar mai grav când pierdem pointerii către acele zone de memorie, întrucât nici dacă am vrea, nu am mai putea să o eliberăm din moment ce nu mai avem pointerii. Pentru a elibera zonele de memorie alocate dinamic folosim ***delete***, respectiv ***delete[]*** în cazul array-urilor alocate dinamic.

C	C++
malloc(), calloc()	new, new[]
free()	delete, delete[]

Returneaza void*	Returnează pointer de tipul corect
Nu apelează constructori	Apelează constructorii
Necesită cast explicit	Nu necesită cast

Exerciții

Este codul din următoarele exemple corect? Dacă da, ce afișează, dacă nu, de ce nu și modificați o linie astfel încât să compileze.

1.

```
#include <iostream>

class cls
{
 public:
   cls() {x = 3;}
   void f(cls& c) {std::cout << c.x;}
   int x;
};

int main()
{
 cls d;
 f(d);
 return 0;
}
```

Nu va functiona, deoarece f apartine clasei, deci ar fi trebuit fie apelarea *d.f(d)*, fie sa definim f global.

2.

```
class cls
{
 public:
   cls(int i = 0, int j = 0) {x = i; y = j;}
   int x, y;
};

int main()
{
 cls a, b, c[3]={cls(1,1), cls(2,2), a};
 std::cout << c[2].x;
 return 0;
}
```

Compileaza si afiseaza 0.

3.

```
class cls
{
 public:
   int x, y;
   cls(int i=2, int j=3)
   {
     x=i+j/2;
     y=i+j/2;
   }
};

int main()
{
 cls a, b, c=a;
 std::cout <<a.x;
 return 0;
}
```

Compileaza si afiseaza 3.

4.

```
class Test
{
 public:
   Test();
};
Test::Test()
{
   std::cout<<"Constructor Called \n";
}

int main()
{
   std::cout<<"Start \n";
   Test t1();
   std::cout<<"End \n";
   return 0;
}
```

Compileaza si afiseaza Start \n End \n. Nu se apeleaza costructorul de initializare pt ca t1() e interpretat ca o functie. S-ar fi apelat constructorul daca aveam *Test t1;* sau *Test t1{};*

5.

```
class Point
{
   int x, y;
   public:
       Point(int i, int j);
};
Point::Point(int i=0, int j=0)
{
   x = i;
   y = j;
   std::cout << "Constructor called";
}

int main()
```

```
{
   Point t1, *t2;
   return 0;
}
```

Compileaza si afiseaza “Constructor called” o singura data (pointerul nu e initializat).

6.

```
class Test
{
   int value;
 public:
   Test(int v);
};
Test::Test(int v)
{
   value = v;
}

int main()
{
   Test t[100];
   return 0;
}
```

Nu compileaza, pentru ca nu mai avem constructor fara parametrii. Putem repara in 3 moduri: fie adaugam constructor fara parametrii (Test()), fie dam o valoare default parametrului v din constructor, fie adaugam Test t[100] = {Test(1), Test(2), ..., Test(100)}.

7.

```
class Test
{
   int &t;
 public:
   Test(int &v){t = v;}
   int getT() {return t;}
};
```

```
int main()
{
   int x = 20;
   Test t1(x);
   std::cout << t1.getT() << " ";
   x = 30;
   std::cout << t1.getT() << std::endl;
   return 0;
}
```

Nu compileaza, avem nevoie de o lista de initializare Test (int &x) : t(x) {}. Acum programul ruleaza si afiseaza 20 30.

Referințele trebuie inițializate la momentul declarației sau în constructor, iar nu **atribuite ulterior**. **Nu se poate atribui o referință după ce a fost definită**, deoarece referințele nu pot fi schimbate pentru a se lega de alt obiect după ce au fost inițializate. *(Before the compound statement that forms the function body of the constructor begins executing, initialization of all direct bases, virtual bases, and non-static data members is finished. The member initializer list is the place where non-default initialization of these subobjects can be specified, preluat de aici).*

8.

```
class Fraction
{
   int den;
   int num;
 public:
   void print() {std::cout<<num << '/' << den;}
   Fraction() {num = 1; den = 1;}
   int &Den() {return den;}
   int &Num() {return num;}
};
int main()
{
   Fraction f1;
   f1.Num() = 7;
   f1.Den() = 9;
   f1.print();
   return 0;
}
```

Compileaza, afiseaza 7/9. Ambele metode returnează o referință la variabilele membre den și num.

9.

```
class Test
{
   int x;
 public:
   void setX(int x) {Test::x = x;}
   void print() { std::cout << "x = " << x << std::endl; }
};


int main()
{
   Test obj;
   int x = 40;
   obj.setX(x);
   obj.print();
   return 0;
}
```

Compileaza, afiseaza x = 40.

10.

```
class A
{
   int id;
 public:
   A (int i) {id = i;}
   void print() {std::cout << id << std::endl;}
};


int main()
{
   A a[2];
   a[0].print();
   a[1].print();
```

```
    return 0;
}
```

Nu compileaza pt ca nu are constructor fara parametri. Putem fie sa adaugam un constructor fara parametri, fie sa adaugam o valoare implicita pentru parametrul i din constructorul parametrizat. Daca alegem prima varianta, adica sa adaug linia A () {}, programul va afisa 2 valori garbage. Daca in schimb adaug parametru default, adica sa modific linia 7 astfel:
A (int i=0) id = i;
Programul va afisa 2 de 0.