Curs 3
Memoria unui program este impartita pe segmente :
- Stack : Variabile locale , parametrii transmisi [gestionat automat si eficient]
- Heap : Variabile alocate dinamic [gestionat manual folosind malloc, calloc, realloc, free]
- Data segment : Variabile globale si statice initializate din program
- Bss segment : Variabile globale si statice neinitializate (memorie initializata cu 0)
- Code segment : Instructiuni cod masina (programul compilat)

Stack avantaje :
- Gestionat automat
- Structura de date LIFO
- Eficient deoarece se gestioneaza doar un pointer la capul stivei

Stack dezavantaje :
- Memoria pentru variabilele de pe stiva este eliberata la terminarea blocului {}
- Dimensiune limitata (1Mb by default)
- Memoria valorilor din stack trebuie cunoscuta la compilare

Heap avantaje :
- Permite alocarea memoriei in timpul rularii programului
- Memoria alocata nu va fii dealocata automat la terminarea blocului {}
- Dimensiune limitata doar de sistemul de operare

Heap dezavantaje :
- Alocarea/de-alocarea de memorie se face de programator

Alocarea dinamica se face folosind functiile malloc(size) - aloca memorie si free(pointer) - elibereaza memorie de pe Heap

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	//allocate memory on the heap for an int
	int *p = malloc(sizeof(int));

	*p = 7;
	printf("%d \n", *p);
	//Deallocate
	free(p);
   //allocate space for 10 ints (array)
	int *t = malloc(10 * sizeof(int));
	t[0] = 0;
	t[1] = 1;
	printf("%d \n", t[1]);
	//dealocate
	free(t);
	return 0;
}
/**
 * Make a copy of str
 * str - string to copy
 * return a new string
 */
char* stringCopy(char* str) {
	int len = strlen(str) + 1; // +1 for the '\0'
	char* newStr = malloc(sizeof(char) * len); // allocate memory
	strcpy(newStr, str); // copy string
	return newStr;
}
```

Memory management
void* malloc(int n); - aloca n bytes memorie neinitializata
void* calloc(int n, int size); - aloca n*size bytes de memorie initializata cu 0
void* realloc(void* address, int newsize); - redimensioneaza memoria deja alocata
void free(void *address); - elibereaza memoria

Nu putem avea variabile de tip **void** dar putem folosi pointer de tip **void*** care functioneaza cu orice tip de elemente, problema este ca nu putem face verificare de egalitate intre elemente de tip **void***.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	void* p;
	int *i=malloc(sizeof(int));
	*i = 1;
	p = i;
	printf("%d /n", *((int*)p));
	long j = 100;
	p = &j;
	printf("%ld /n", *((long*)p));
	  free(i);
	return 0;
}
```

TAD - Tip abstract de date

- Defineste domeniul de valori
- Defineste operatiile asociate (interfata)
- Definitia operatiilor este independenta de implementarea acestora (abstractizare)

Tad implementat in C

<tip.h> + <tip.c>

interfata implementare

Exemplu vector dinamic :

interfata

```c
/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el);

/**
 *Returneaza elementul de pe pozitia data
 * v - vector
 * poz - pozitie, poz>=0
 * returneaza elementul de pe pozitia poz
 */
Element get(VectorDinamic *v, int poz);

/**
 * Aloca memorie aditionala pentru vector
 */
void resize(VectorDinamic *v) {
  int nCap = 2*v->capacitate;
  Element* nElems=
malloc(nCap*sizeof(Element));
	//copiez din vectorul existent
  int i;
  for (i = 0; i < v->lg; i++) {
		nElems[i] = v->elems[i];
  }
  //dealocam memoria ocupata de vector
  free(v->elems);
  v->elems = nElems;
  v->capacitate = nCap;
}
/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el) {
	if (v->lg == v->capacitate) {
		resize(v);
	}
	v->elems[v->lg] = el;
	v->lg++;
}
```

implementare

```c
typedef void* Element;

typedef struct {
	Element* elems;
	int lg;
	int capacitate;
} VectorDinamic;

/**
 *Creaza un vector dinamic
 * v vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic();


/**
 *Initializeaza vectorul
 * v vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic() {
  VectorDinamic *v =
malloc(sizeof(VectorDinamic));
  v->elems = malloc(INIT_CAPACITY *
sizeof(Element));
  v->capacitate = INIT_CAPACITY;
  v->lg = 0;
  return v;
}
/**
 * Elibereaza memoria ocupata de vector
 */
void distruge(VectorDinamic *v) {
	int i;
	for (i = 0; i < v->lg; i++) {
//!!!!functioneaza corect doar daca
//elementele din lista NU refera
// memorie alocata dinamic
		free(v->elems[i]);
	}
	free(v->elems);
	free(v);
}
```

Pointer la functii

```
void(*funcPtr)(int);  // function returns void has an int parameter
int(*funcPtr2)(int,int); // function returns int has two int
parameters
```

```
void f(int a) {
     printf("%d\n", a);
}
```

```
int main() {
  void(*funcPtr)(int);
  funcPtr = f;
  funcPtr(6);
  return 0;
}
```

```
int sum(int a, int b) {
     return a + b;
}
```

```
int main() {
   int(*funcPtr2)(int,int);
   funcPtr2 = sum;
   int c = funcPtr2(6,3);
   printf("%d\n", c);
   return 0;
}
```

```
typedef void* ElemType; //Se creeaza un alias (ElemType) pentru tipul void*
//function type for dealocating an element
typedef void(*DestroyF)(ElemType); //Se creeaza un alias (DestroyF) pentru un
                                   pointer la functie care are un parametru de
                                   tip ElemType
typedef struct {
     ElemType* elems;
     int lg;
     int capacitate;
} MyList;
```

```
/*
Dealocate list
*/
void destroy(MyList* l, DestroyF dealocate) { //Daca nu aveam typedef
     //free elements                          void(*DestroyF)(ElemType)
     for (int i = 0; i < l->lg; i++) {        atunci aici ar fi trebui sa
          dealocate(l->elems[i]);             avem ca si parametrul doi :
     }                                        void (*dealocate)(void *)
     //free list
     free(l->elems);
     l->lg = 0;
}
```

```
void testCopyList() {
     MyList l = createEmpty();
     add(&l, createPet("a", "b", 10));
     add(&l, createPet("a2", "b2", 20));
     MyList l2 = copyList(&l);
     assert(size(&l2) == 2);
     Pet* p = get(&l2, 0);
     assert(strcmp(p->type, "a") == 0);
     destroy(&l, destroyPet);
     destroy(&l2, destroyPet);
}
```

Limbajul C++
Tipuri de date predefinite:
int, long, double, char, bool, void, etc.

Conversii intre tipuri :
Este nevoie de o conversie explicita type-casting cand dorim sa interpretam o valoare
`char c = (char)2000;` C-style cast,elimina warningurile cauzate de conversii periculoase
`char c = char(2000);` De evitat pentru ca poate cauza probleme daca tipurile nu sunt compatibile
`int a = static_cast<int>(7.5);` Este verificat la compilare si afiseaza eroare de compilare daca conversia este imposibila (tipuri incompatibile)

Tipul referinta :
`int y = 7;`
`int &x = y;` //make x a reference to, or an alias of y
Daca schimbam x se schimba si y si invers, sunt doua nume pentru aceasi locatie de memorie

```
/**
 *  C++ version
 *  Sum of 2 rational number
 */
void sum(Rational nr1, Rational nr2, Rational &rez) {
     rez.a = nr1.a * nr2.b + nr1.b * nr2.a;
     rez.b = nr1.b * nr2.b;
     int d = gcd(rez.a, rez.b);
     rez.a = rez.a / d;
     rez.b = rez.b / d;
}
```

```
/**
 *  C version
 * Sum of 2 rational number
 */
void sum(Rational nr1, Rational nr2, Rational *rez) {
     rez->a = nr1.a * nr2.b + nr1.b * nr2.a;
     rez->b = nr1.b * nr2.b;
     int d = gcd(rez->a, rez->b);
     rez->a = rez->a / d;
     rez->b = rez->b / d;
}
```

Declarare / Initializare de variabile :

`int b { 7 };`
 	
`int c = { 7 };`
	Universal form
	varianta de preferat in modern in C++
	Evită problemele legate de conversii prin care se pierde precizie (narrowing)

`int a = 7;`	Varianta “clasică” moștenită din C
`int d;//gresit`	Varianta greșita, compilează dar variabila este neinițializată (are o valoare aleatoare)

auto :
Cand definim o variabila putem sa nu specificam tipul exact si compilatorul deduce tipul din expresia de initializare

```
auto a = 7;//a e int
double b{7.4};
double c{1.4};
auto d = b+c;//d e double
```

auto este util pentru :
- a evita scrierea de nume lungi de tipuri
- scriere de cod generic

Const :
const semnaleaza compilatorului ca nu dorim sa schimbam valoarea care urmeaza dupa const, daca se incearca schimbarea valorii atunci rezulta o eroare de compilare

```
const int nr = 100;
```

```
typedef struct{
    int a;
    int b;
} Rationa;

void add(const Rationa& r1, const Rationa& r2, Rationa&
rez){
    ...
}
```

Folosim const pentru a exprima ideea de imutabil peste tot unde are sens:
- compilatorul ajuta la evitarea unor greseli (compilatorul verifica si da eroare daca se incearca comunicarea)
- codul este mai usor de inteles de altii (se exprima mai bine intentia programatorului)

Const Pointer
type* const ptr - semnaleaza compilatorului ca nu dorim sa modificam adresa referita de ptr, in schimb putem modifica valoarea de la adresa retinuta de ptr

```
int * const p3 = &j;
cout << *p2 << "\n";
//change the memory address (compiler error)
p3 = &i;
cout << *p3 << "\n";
//change the value (valid)
*p3 = 7;
cout << *p3 << "\n";
```

const type* ptr - semnaleaza compilatorului ca nu dorim sa modificam valoarea de la adresa retinuta de ptr, in schimb se poate modifica adresa referita de ptr

```
const int* p2 = &j;
cout << *p2 << "\n";
p2 = &i;//change the memory address (valid)
cout << *p2 << "\n";
*p2 = 7;//change the value (compiler error)
cout << *p2 << "\n";
```

const type* const ptr - semnaleaza compilatorului ca nu dorim sa modificam adresa referita de ptr cat si valoarea de la adresa retinuta de ptr

```
int * const p3 = &j;
```

Range for

```
int a[] = {0, 1, 2, 3, 4, 5};
for (auto v:a) {
    cout << v << "\n";
}
```

```
for (auto v:{0,1,2,3,4,5} ) {
    cout << v << "\n";
}
```

Pentru fiecare element din a, se face o copie in variabila v.
Range se poate folosi cu orice secventa de elemente.

Daca dorim sa facem copie prin referinta (atat si elementul din a cat si v sa aiba aceeasi adresa) atunci putem folosi auto&.
Daca dorim sa facem copie prin referinta dar nu dorim sa modificam elementele putem folosi const auto&. Este util pentru copierea de stringuri pentru ca evita copierea intregului sir de caractere si face referinta la adresa stringului.

```
for (auto& v : a) {
    ++v;
}
```

```
for (const auto& v : a) {
    cout << v << "\n";
}
```

Input/Output library in C++
Libraria este <iostream>
cin - corespunde intrari standard (stdin), tip **istream**
cout - corespunde iesiri standard (stdout), tip **ostream**
cerr - corespunde erori standard (stderr), tip **ostream**

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

void testStandardIOStreams() {
    //prints Hello World!!! to the console
    cout << "Hello World!!!\n";
    int i = 0;
    cin >> i; //read an int from the console
    cout << "i=" << i << "\n"; // printsto the console
    //write a message to the standard error stream
    cerr << "Error message";
}
```

- Operatia de output se realizeaza folosind operatorul "**<<**", insertion operator
- Operatia de input se realizeaza folosind operatorul "**>>**", extraction operator