

The background features abstract, overlapping purple geometric shapes, primarily triangles and polygons, in various shades of purple, creating a modern, layered effect.

P00

Curs-5

Gavrilut Dragos

Modalitati de cast

- ▶ Castarea de la un tip la altul poate implica mai multe operatii:
 - ▶ Apelul unei functii de cast explicite
 - ▶ Modificarea unui pointer (daca discutam de clase derivate)
 - ▶ Etc
- ▶ Ce putem face daca dorim ca un anumit cast sa se realizeze intr-un fel specific ?

Modalitati de cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : public A, public B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    B* b = &c;  
}
```

- Cum se realizeeaza castarea de la C* la B* ?

Modalitati de cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : public A, public B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    B* b = &c;  
}
```

- Cum se realizeaza

```
lea     eax,[c]  
test    eax,eax  
je      NULL_CAST  
lea     ecx,[c]  
add     ecx,0Ch  
mov     dword ptr [ebp-0F8h],ecx  
jmp     DONE  
  
NULL_CAST:  
mov     dword ptr [ebp-0F8h],0  
  
DONE:  
mov     edx,dword ptr [ebp-0F8h]  
mov     dword ptr [b],edx
```

Modalitati de cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : public A, public B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    B* b = &c;  
}
```

- Dar daca dorim ca “b” sa puncteze chiar la inceputul lui “c” si sa nu se mai ia in calcul deplasamentul de 12 octeti, ce putem face ?

Modalitati de cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : public A, public B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    B* b = (B*)((void *)&c);  
}
```

- O solutie ar fi sa castam la un void* si apoi de acolo catre un B*

Modalitati de cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : public A, public B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    B* b = (B*)((void *)&c);  
}
```

lea
mov

eax,[c]
dword ptr [b],eax

- O solutie ar fi sa castam la un void* si apoi de acolo catre un B*

Modalitati de cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : public A, public B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    B* b = reinterpret_cast<B*> (&c);  
}
```

- O alta solutie mai eleganta e sa folosim modalitati specifice de castare pe care le pune la dispozitie C++-ul → in cazul de fata utilizarea cuvintului cheie **reinterpret_cast**.

Modalitati de cast

- ▶ C++ pune la dispozitie mai multe modalitati prin care se poate specifica cum se doreste a se realiza o castare:
 - ▶ `static_cast`
 - ▶ `reinterpret_cast`
 - ▶ `dynamic_cast`
 - ▶ `const_cast`
- ▶ Toate au urmatoarea sintaxa de utilizare:

App.cpp

```
static_cast <tipul in care vrem sa castam> (expresia de castat)  
const_cast <tipul in care vrem sa castam> (expresia de castat)  
dynamic_cast <tipul in care vrem sa castam> (expresia de castat)  
reinterpret_cast <tipul in care vrem sa castam> (expresia de castat)
```

reinterpret_cast

- ▶ La modul general reinterpret_cast presupune transformarea cea mai simpla posibila (pentru pointeri tot timpul se pastreaza exact valoarea primita).

App.cpp

```
reinterpret_cast <tip*> (ptr) ⇔ ((tip*)((void*) ptr))
```

- ▶ Este conversia care garanteaza si utilizarea unui numar redus de instructiuni in limbaj de asamblare (deci cod mai efficient si mai rapid)
- ▶ Trebuie insa utilizata cu atentie pentru ca poate produce cast-uri intre tipuri incompatibile, sau cu alinieri diferite in memorie.

reinterpret_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : private A, private B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    A* a = &c;  
}
```

- Codul nu compileaza pentru ca, desi exista o conversie valida de la C* la A* (care nu implica modificarea pointerului), derivarea lui C din A se face cu private si compilatorul nu poate da access la date membru private.

reinterpret_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
class C : private A, private B { public: int c1, c2; };  
  
void main(void)  
{  
    C c;  
    A* a = reinterpret_cast<A*> (&c);  
}
```

- Codul compileaza → reinterpret_cast nu tine cont de tip, de derivare, etc (ci doar va lua pointerul care puncteaza la obiectul “c” si il va asigna lui a.

reinterpret_cast

- ▶ reinterpret_cast nu se poate folosi pentru valori constante, pentru variabile. Urmatoarele utilizari ale lui reinterpret_cast NU vor compila:

Cod C++

```
int x = reinterpret_cast <int>(1000);  
  
int y = 100;  
int x = reinterpret_cast <int>(y);
```

- ▶ reinterpret_cast poate sa traduca dintr-un pointer intr-o valoare (chiar daca e o valoare care nu poate tine toata adresa pointerului).

Cod C++

```
int x = reinterpret_cast<int>("test");  
  
char x = reinterpret_cast<char>("test");
```

ASM

```
mov dword ptr [x],205858h  
  
mov     eax, 205858h  
mov     byte ptr [x],al
```

- ▶ In acest caz **0x205858** este adresa unde se gaseste stringul constant "test" in memorie.

reinterpret_cast

- ▶ reinterpret_cast insa NU va functiona in acelasi fel cu tipuri de date flotante. Urmatoarele sintaxe nu vor compila

Cod C++

```
float x = reinterpret_cast<float>(1.2f);
```

```
Double x = reinterpret_cast<double>(1.2);
```

- ▶ reinterpret_cast poate fi in schimb utilizat cu referinte numerice pentru a le schimba valoarea (inclusive cu referinte de tip diferit)

Cod C++

```
int number = 10;  
reinterpret_cast<int &>(number) = 20;
```

```
int number = 10;  
reinterpret_cast<char &>(number) = 20;
```

ASM

```
mov dword ptr [number],20
```

```
mov byte ptr [number],20
```

reinterpret_cast

- Deasemenea, reinterpret_cast poate fi folosit pentru a lua adresa unei functii.

App.cpp

```
int addition(int x, int y)
{
    return x + y;
}

void main(void)
{
    char* a = reinterpret_cast<char*> (addition);
}
```

- In acest caz, pointerul “a” va puncta catre adresa de inceput din memorie a functiei addition.

static_cast

- ▶ static_cast implica o conversie in care:
 - ▶ Valorile pot fi trunchiate sau rotunjite inainte de asignare
 - ▶ In cazul claselor derivate, valoarea pointerului de pornire poate sa fie modificata
- ▶ Poate fi folosit pentru a obtine o lista de initializare
- ▶ NU face nici o verificare pe tipul obiectului (campul RTTI)
- ▶ Mai poate fi folosit pentru a identifica exact o functie (atunci cand avem supraincarcare pentru acea functie).

static_cast

- ▶ `static_cast` se poate folosi si pentru valori constante. In acest caz conversia este uneori facuta automat in faza de precompilare.

Cod C++	ASM
<code>int x = static_cast<int>(1000);</code>	<code>mov dword ptr [x],1000</code>
<code>char x = static_cast<char>(1000);</code>	<code>mov byte ptr [x],232 (1000 % 256)</code>
<code>char x = static_cast<char>(3.75);</code>	<code>mov byte ptr [x],3 (3 = int(3.75))</code>
<code>char x = static_cast<char>("test");</code>	Nu exista conversie posibila intre un <code>const char*</code> si un <code>char</code>
<code>const char * x = static_cast<const char *>(100);</code>	Similar, nu exista conversie posibila

static_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B { public: int b1, b2; };  
  
void main(void)  
{  
    B b;  
    A* a = static_cast<A *>(&b);  
}
```

- Codul nu v-a compila pentru ca nu este posibil un cast de la un obiect de tipul B* la unul de tipul A*

static_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B  
{  
    public:  
        int b1, b2;  
        operator A* () { return new A(); }  
};  
void main(void)  
{  
    B b;  
    A* a = static_cast<A *>(&b);  
}
```

- Codul nu v-a compila din acelasi motiv. Prezenta operatorului de cast de la B la A* implica un obiect de tipul B pentru cast (si nu un pointer de tipul B).

static_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B  
{  
    public:  
        int b1, b2;  
        operator A* () { return new A(); }  
};  
void main(void)  
{  
    B b;  
    A* a = static_cast<A *>(b);  
}
```

- Codul compileaza si apeleaza operatorul de conversie de la B (prin instanta b) la A*

static_cast

- Fie urmatorul cod:

App.cpp

```
int Add(int x, char y)
{
    return x + y;
}
int Add(char x, int y)
{
    return x + y;
}
void main(void)
{
    int suma = Add(100, 200);
}
```

- Codul NU compileaza (cazul e ambiguu - compilatorul nu poate decide care din cele doua versiuni de functii Add ar trebui utilizata).

static_cast

- Fie urmatorul cod:

App.cpp

```
int Add(int x, char y)
{
    return x + y;
}
int Add(char x, int y)
{
    return x + y;
}
void main(void)
{
    int suma = static_cast<int(*) (int, char)> (Add)(100, 200);
}
```

- Codul compileaza si apeleaza functia care trebuie. Prin utilizarea lui static_cast putem sa accesam un pointer catre o functie si sa specificam exact care dintre cele doua suprainncarcari a functiei Add vrem sa o folosim.

dynamic_cast

- ▶ `dynamic_cast` functioneaza avand ca si parametru doar un pointer si/sau o referinta si converteste pe aceasta intr-un mod sigur intr-un obiect derivat/derivabil din tipul pointerului.
- ▶ Evaluarea se face folosind campul `RTTI` din `vfptr` si se face la runtime. Din acest motiv, pe cat posibil e recomandat de utilizat `static_cast` (insa acesta nu poate garanta mereu ca un cast este sigur)
- ▶ Acest lucru implica ca pentru a putea functiona, campul `RTTI` trebuie sa fie present (clasele pe care se aplica trebuie sa contina macar o metoda virtuala).

dynamic_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B  
{  
    public:  
        int b1, b2;  
};  
class C : public A, public B { public: int c1, c2; };  
void main(void)  
{  
    C c;  
    B *b = dynamic_cast<B*>(&c);  
}
```

- Codul compileaza insa conversia este una sigura. Cum C este derivat din B este sigur sa convertim un pointer al unui obiect de tipul C la unul de tipul B.

dynamic_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B  
{  
    public:  
        int b1, b2;  
};  
class C : public A, public B { public: int c1, c2; };  
void main(void)  
{  
    B b;  
    C *c = dynamic_cast<C*>(&b);  
}
```

- Codul NU compileaza (“error C2683: 'dynamic_cast' : 'B' is not a polymorphic type”). Cum C nu este derivat din B, conversia nu este sigura asa ca este nevoie de existenta RTTI-ului pentru a evalua daca conversia este una posibila. B nu are nici o functie virtuala, deci codul nu va compila.

dynamic_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
        virtual void f() {};
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
    B b;
    C *c = dynamic_cast<C*>(&b);
}
```

- Codul compileaza, insa “c” va primi valoarea NULL pentru ca nu exista un obiect C care sa se poata obtine din “b”

dynamic_cast

- Fie urmatorul cod:

App.cpp

```
class A { public: int a1, a2, a3; };  
class B  
{  
    public:  
        int b1, b2;  
        virtual void f() {};  
};  
class C : public A, public B { public: int c1, c2; };  
void main(void)  
{  
    C c;  
    B *b = (B*)&c;  
    C *c2 = dynamic_cast<C*>(b);  
}
```

- Codul compileaza. De aceasta data, cu in realitate “b” puncta catre un obiect de tipul “c” se poate obtine un cast catre acel obiect, si “c2” va avea aceeaasi adresa ca si obiectul “c”

dynamic_cast

- Fie urmatorul cod:

App.cpp

```
class A {  
    public: int a1, a2, a3;  
    virtual void f2() {};  
};  
class B {  
    public: int b1, b2;  
    virtual void f() {};  
};  
class C : public A, public B { public: int c1, c2; };  
void main(void) {  
    C c;  
    A *a = (A*)&c;  
    B *b = dynamic_cast<B*>(a);  
}
```

- Codul compileaza. In realitate nu exista o legatura intre A si B (ca se se poata face o conversie implicita). Insa, in cazul de fata, atat “a” cat si “b” sunt pointeri in interiorul obiectului “c”. Din acest punct de vedere, exista o legatura si se poate ajunge la un B dintr-un A (sau invers) cu conditia ca pointerii sa apartina unui obiect de tipul C.

const_cast

- ▶ `const_cast` e utilizat pentru a putea schimba caracteristica “const” a unor obiecte.
- ▶ Chiar daca acest operator exista, utilizarea lui poate duce in anumite circumstante la un comportament nedefinit.
- ▶ operatorul se poate aplica pe date de acelasi tip (si va schimba doar statusul de const sau nu).
- ▶ Este posibil de folosit intr-o clasa care contine un obiect de tipul `const` / metode care sunt constante.

const_cast

- Fie urmatorul cod:

App.cpp

```
void main(void)
{
    const int x = 100;
    x = 200;
}
```

- Codul nu compileaza. X a fost definit ca un const, si nu mai poate fi modificat dupa asignare.

const_cast

- Fie urmatorul cod:

App.cpp

```
void main(void)
{
    const int x = 100;
    *(const_cast<int*>(&x)) = 200;
}
```

- Codul compileaza. Rezultatul insa este nedefinit.

const_cast

- Fie urmatorul cod:

App.cpp

```
void main(void)
{
    const int x = 100;
    *(const_cast<int*>(&x)) = 200;
    printf("%d", x);
}
```

```
const int x = 100;
mov     dword ptr [x],64h
*(const_cast<int*>(&x)) = 200;
mov     dword ptr [x],0C8h
printf("%d", x);
mov     esi,esp
push    64h
push    2058A8h ; "%d"
call    printf
add     esp,8
```

- Codul compileaza. Rezultatul insa e
- Caz 1: executie pe mod DEBUG
 - X este definit pe stiva
 - Valoarea lui practic poate fi schimbata
 - Compilatorul insa stie ca e o valoare constanta asa ca, chiar daca a fost modificata tot va afisa 100 (in codul din ASM se trimite 0x64 (push 64h) ls apelul functiei printf)

const_cast

- Fie urmatorul cod:

App.cpp

```
void main(void)
{
    const int x = 100;
    *(const_cast<int*>(&x)) = 200;

    printf("%d", *(const_cast<int*>(&x))) );
}
```

```
const int x = 100;
mov     dword ptr [x],64h
*(const_cast<int*>(&x)) = 200;
mov     dword ptr [x],0C8h
printf("%d", x);
mov     esi,esp
push    x
push    2058A8h ; "%d"
call    printf
add     esp,8
```

- Codul compileaza. Rezultatul insa este nedefinit.
- Caz 2: executie pe mod DEBUG
 - X este definit pe stiva
 - Valoarea lui eeste schimbata
 - Pentru ca fortam un const_cast compilatorul citeste chiar valoarea de pe stiva. In acest caz , va afisa 200 !

const_cast

- Fie urmatorul cod:

App.cpp

```
class Test
{
public:
    const int x;
    Test(int value) : x(value) { }
    void Set(int value) { *(const_cast<int*>(&x)) = value; }
};

void main(void)
{
    Test t(100);
    printf("%d ", t.x);
    t.Set(200);
    printf("%d ", t.x);
}
```

- Codul compileaza. In acest caz, variabila “x” chiar daca este constanta apartine cu siguranta instantei ceea ce inseamna ca in realitate poate fi modificata. Codul ruleaza si afiseaza “100 200”



► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ Template-uri specializate
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ Template-uri specializate
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

Macro-uri

- ▶ Macro-urile sunt modalități prin care se poate controla preprocesarea codului scris în C/C++
- ▶ Se definesc folosind sintaxa:
`#define <macro> <valoare>`
- ▶ Un macro o dată definit poate fi scos din lista de definiții folosind sintaxa:
`#undef macro`
- ▶ Macro-urile sunt o componentă importantă a pre-procesării codului → ele funcționează prin înlocuirea secvențelor de cod pe care se potrivesc cu valoarea lui.

Macro-uri

- ▶ Exemple (macro-uri simple)

App.cpp

```
#define DIMENSIUNE_BUFFER    1024  
  
char Buffer[DIMENSIUNE_BUFFER];
```

- ▶ După faza de precompilare, codul obținut va arata in felul următor:

App.cpp

```
char Buffer[1024];
```

Macro-uri

- ▶ Macro-urile funcționează secvențial (se aplica imediat după momentul definiției lor). De asemenea, fiind în faza de precompilare, nu există noțiunea de identificator încă, deci o variabilă și un macro cu același nume pot exista

App.cpp

```
void main(void)
{
    int value = 100;
    int temp;
    temp = value;
    #define value    200
    temp = value;
}
```

- ▶ După faza de precompilare, codul obținut va arăta în felul următor:

App.cpp

```
void main(void)
{
    int value = 100;
    int temp;
    temp = value;
    temp = 200;
}
```

Macro-uri

- ▶ Macro-urile pot fi scrise si pe mai multe linii. Pentru acest lucru se folosește caracterul special ‘\’ la sfârșitul fiecărei linii. ATENTIE - după acel caracter nu trebuie sa mai urmeze un altul (decât EOL).

App.cpp

```
#define AFISEAZA \  
    if (value > 100) printf("Mai mare !"); \  
    else printf("Mai mic !");  
  
void main(void)  
{  
    int value = 100;  
    AFISEAZA;  
}
```

- ▶ După faza de precompilare, codul obținut va arata in felul următor:

App.cpp

```
void main(void)  
{  
    int value = 100;  
    if (value > 100) printf("Mai mare !");  
    else printf("Mai mic !");;  
}
```


Macro-uri

- ▶ Macro-urile se pot defini folosind un alt macro. In acest caz, utilizarea `#undef` si `#define` poate schimba valoarea unui macro in timpul preprocesării

App.cpp

```
#define BUFFER_SIZE      VALOARE
#define VALOARE          1024
char Temp[BUFFER_SIZE];

#undef VALOARE

#define VALOARE          2048
char Temp2[BUFFER_SIZE];
```

- ▶ După faza de precompilare, codul obținut va arata in felul următor:

App.cpp

```
char Temp[1024];
char Temp2[2048];
```

Macro-uri

- ▶ Macro-urile se pot defini ca o funcție cu mai mulți parametri.

App.cpp

```
#define MAX(x,y) ((x)>(y)?(x) : (y))

void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    int v3 = MAX(v1, v2);
}
```

- ▶ După faza de precompilare, codul obținut va arata în felul următor:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    int v3 = ((v1)>(v2) ? (v1) : (v2));
}
```

Macro-uri

- ▶ Macro-urile definite ca o funcție pot avea și număr variabil de parametri dacă folosim operatorul '...'

App.cpp

```
#define AFISEAZA(format,...) \
{ \
    printf("\nAfisam valori:"); \
    printf(format, __VA_ARGS__); \
}
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    AFISEAZA("%d,%d", v1, v2);
}
```

- ▶ După faza de precompilare, codul obținut va arăta în felul următor:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    printf("\nAfisam valori:");
    printf("%d,%d", v1, v2);
}
```

Macro-uri

- ▶ Macro-urile pot folosi caracterul special ‘#’ pentru a transforma un parametru într-un string corespunzător:

App.cpp

```
#define VERIFICA(conditie) \
    if (!(conditie)) { printf("Conditia '%s' nu s-a evaluat corect", #conditie); };

void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    VERIFICA(v1 > v2);
}
```

- ▶ După faza de precompilare, codul obținut va arăta în felul următor:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    if (!(v1>v2)) { printf("Conditia '%s' nu s-a evaluat corect", "v1 > v2"); };
}
```

Macro-uri

- ▶ Macro-urile pot folosi caracterul special '#' pentru a transforma un parametru intr-un string corespunzator:

App.cpp

```
#define VERIFICA(conditie) \
    if (!(conditie)) { printf("Conditia '%s' nu s-a evaluat corect", #conditie); };\n\nvoid main(void)\n{\n    int v1, v2;\n    v1 = 100;\n    v2 = 200;\n    VERIFICA(v1 > v2);\n}
```

- ▶ Dupa faza de precompilare, codul obtinut va arata in felul urmator:

App.cpp

```
void main(void)\n{\n    int v1, v2;\n    v1 = 100;\n    v2 = 200;\n    if (!(v1>v2)) { printf("Conditia '%s' nu s-a evaluat corect", "v1 > v2"); };\n}
```

Macro-uri

- ▶ Macro-urile pot folosi secventa de caractere “##” pentru a concatena variabile:

App.cpp

```
#define ADUNA(tip) \  
    tip add_##tip(tip v1, tip v2) { return v1 + v2; }  
  
ADUNA(int);  
ADUNA(double);  
ADUNA(char);  
void main(void)  
{  
    int x = add_int(10, 20);  
}
```

- ▶ Dupa faza de precompilare, codul obtinut va arata in felul urmator:

App.cpp

```
int add_int(int v1, int v2) { return v1 + v2; }  
double add_double(double v1, double v2) { return v1 + v2; }  
char add_char(char v1, char v2) { return v1 + v2; }  
  
void main(void)  
{  
    int x = add_int(10, 20);  
}
```

Macro-uri

- ▶ Macro-urile nu suporta supraincarcare (mai exact - daca exista doua macrouri cu acelasi nume, ultimul il inlocuieste pe cel precedent, se emite un warning in compilator si se compileaza).
- ▶ Codul de mai jos nu compileaza pentru ca SUM are nevoie de 3 parametri (doar al doilea macro e utilizat)

App.cpp

```
#define SUM(a,b) a+b  
#define SUM(a,b,c) a+b+c  
  
void main(void)  
{  
    int x = SUM(1, 2);  
}
```

- ▶ Codul de mai jos compileaza si functioneaza corect

App.cpp

```
void main(void)  
{  
    #define SUM(a,b) a+b  
    int x = SUM(1, 2);  
    #define SUM(a,b,c) a+b+c  
    x = SUM(1, 2, 3);  
}
```

Macro-uri

- Utilizati parantezele ! Macro-urile nu analizeaza expresia ci doar fac un replace in text → rezultatele pot fi altele decat cele asteptate.

Incorect	Corect
<pre>#define DIV(x,y) x/y void main(void) { int x = DIV(10 + 10, 5 + 5); }</pre>	<pre>#define DIV(x,y) ((x)/(y)) void main(void) { int x = DIV(10 + 10, 5 + 5); }</pre>

- Dupa faza de precompilare, codul obtinut va arata in felul urmator:

Incorect	Corect
<pre>void main(void) { int x = 10 + 10 / 5 + 5; }</pre>	<pre>void main(void) { int x = ((10 + 10) / (5 + 5)); }</pre>

Macro-uri

- Atentia la utilizarea intr-o bucla. Pe cat posibil folositi '{' si '}' ca sa evidentiati o instructiune complexa !

Inc corect	Corect
<pre>#define AFISEAZA(x,y) \ printf("X=%d",x);printf("Y=%d",y); void main(void) { int x = 10, y = 20; if (x > y) AFISEAZA(x, y); }</pre>	<pre>#define AFISEAZA(x,y) \ { printf("X=%d",x);printf("Y=%d",y); } void main(void) { int x = 10, y = 20; if (x > y) AFISEAZA(x, y); }</pre>

- Dupa faza de precompilare, codul obtinut va arata in felul urmator:

Inc corect	Corect
<pre>void main(void) { int x = 10, y = 20; if (x > y) printf("X=%d",x);printf("Y=%d",y); }</pre>	<pre>void main(void) { int x = 10, y = 20; if (x > y) {printf("X=%d",x);printf("Y=%d",y);} }</pre>

Macro-uri

- Atentia la operatori sau functii care modifica parametrul primit de macro. Din cauza substitutiei, unele apeluri se vor face de mai multe ori.

Incorect (res va fi 3)	Corect
<pre>#define set_min(result,x,y) \ result = ((x)>(y)?(x):(y)); void main(void) { int x = 2, y = 1; int res; set_min(res, x++, y); }</pre>	<pre>#define set_min(result,x,y) {\ int t_1 = (x); \ int t_2 = (y); \ result = ((t_1)>(t_2)?(t_1):(t_2)); \ } void main(void) { int x = 2, y = 1; int res; set_min(res, x++, y); }</pre>

- Dupa faza de precompilare, codul obtinut va arata in felul urmator:

Incorect	Corect
<pre>void main(void) { int x = 2, y = 1; int res; result = ((x++>(y)?(x++):(y)); }</pre>	<pre>void main(void) { int x = 2, y = 1; int res; { int t_1 = (x); int t_2 = (y); result = ((t_1)>(t_2)?(t_1):(t_2)); } }</pre>

Macro-uri

- Exista o serie de macro-uri care sunt predefinite pentru orice compilator de C/C++:

Macro	Valoare
__FILE__	Fisierul current in care se substitue macro-ul
__LINE__	Linia din fisierul current in care se substitue macro-ul
__DATE__	Data curenta la momentul compilarii
__TIME__	Timpul la momentul compilarii
__STDC_VERSION__	Versiunea compilatorului (Cx98, Cx03,Cx13,...)
__cplusplus	Este definit daca se foloseste un compilator de C++
__COUNTER__	Un indicativ unic (0..n) folosit pentru indexare

- Pe langa acestia fiecare compilator in parte mai defineste multe alte macro-uri specific (pentru versiunea de compilator, optiuni de linkare, etc)

Macro-uri

- Utilizarea lui `__COUNTER__` este foarte benefica daca dorim sa creem indexi unici in program:

App.cpp

```
void main(void)
{
    int x = __COUNTER__;
    int y = __COUNTER__;
    int z = __COUNTER__;
    int t = __COUNTER__;
}
```

- Dupa faza de precompilare, codul obtinut va arata in felul urmator:

App.cpp

```
void main(void)
{
    int x = 0;
    int y = 1;
    int z = 2;
    int t = 3;
}
```

► Macro-uri

► **Template-uri**

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ Template-uri specializate
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

Template-uri

- ▶ Macro-urile aduc o putere mare unui cod scris in C/C++
- ▶ Template-urile pot fi considerate o derivare da la notiunea de macro → adaptata pentru functii si clase
- ▶ Scopul este sa definim un model al unei functii sau a unei clase in care tipurile de date cu care lucram sa poata sa fie modificate din faza de precompilare (similar ca si la macro-uri)
- ▶ Pentru acest lucru, exista un cuvand cheie “template”
- ▶ Ca si in cazul macro-urilor, utilizarea template-urilor genereaza cod in plus la compilare (specific pentru tipurile de date pentru care se fac template-urile). Codul insa este mult mai rapid si mai efficient.

Template-uri

- ▶ Template-urile definite in C++ sunt de doua tipuri:
 - ❖ Pentru clase
 - ❖ Pentru functii
- ▶ Template-urile functioneaza exact ca si un macro - prin substitutie
- ▶ Diferenta e ca substituirea template-urilor pentru clase nu se face in locul in care se foloseste prima data o instant a acelei clase ci separate, astfel incat alte instate care folosesc aceleasi tipuri de date sa poata folosi acelasi template substituit
- ▶ **ATENTIE**: Pentru ca substituirea se face in faza de precompilare, template-urile trebuiesc tinute in fisierele care se exporta dintr-o librarie (fisierele .h) altfel compilatorul (in cazul in care se incearca crearea unei clase pe baza unui template exportat dintr-o librarie) nu va putea face acest lucru.

Template-uri

- ▶ Template-urile pentru o functie se definesc in felul urmator:

App.cpp

```
template <class T>
Tip_return nume_functie(parametri)

sau
template <typename T>
Tip_return nume_functie(parametri)
```

- ▶ Macar unul dintre “Tip_return” sau “parametri” trebuie sa contina un membru de tipul T.

App.cpp

```
template <class T>
T Suma(T valoare_1, T valoare_2)
{
    return valoare_1 + valoare_2;
}
```




► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ Template-uri specializate
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

Template-uri

- Un exemplu simplu:

App.cpp

```
template <class T>
T Suma(T valoare_1, T valoare_2)
{
    return valoare_1 + valoare_2;
}

void main(void)
{
    double x = Suma(1.25, 2.5);
}
```

- Codul compileaza → x primește valoarea 3.75

Template-uri

- Un exemplu simplu:

App.cpp

```
template <class T>
T Suma(T valoare_1, T valoare_2)
{
    return valoare_1 + valoare_2;
}

void main(void)
{
    double x = Suma(1, 2.5);
}
```

- Codul NU compileaza - 1 este evaluat la tipul int, 2.5 la tipul double, iar functia Suma primeste doi parametri de acelasi tip.
- Compilatorul nu poate decide pe care sa il foloseasca → cod ambiguu

Template-uri

- Un exemplu simplu:

App.cpp

```
template <class T>
T Suma(T valoare_1, T valoare_2)
{
    return valoare_1 + valoare_2;
}

void main(void)
{
    int x = Suma(1, 2);
    double d = Suma(1.5, 2.4);
}
```

- In cazul de fata → x va primi valoarea 3 iar functia apelata va substitui class T cu int, iar d va primi valoarea 3.9 (in acest caz substituirea se va face cu double).
- In codul compilat vor exista 2 functii Suma (una cu parametri de tipul int, alta cu parametri de tipul double)

Template-uri

- Atentie la cum puneti parametri !!!

App.cpp

```
template <class T>
T Suma(int x, int y)
{
    return (T)(x + y);
}

void main(void)
{
    int x = Suma(1, 2);
    double d = Suma(1.5, 2.4);
}
```

- Codul de mai sus nu compileaza → nu pentru ca valoarea de return este diferita ci pentru ca compilatorul nu poate deduce ce sa foloseasca pentru clasa T (mai exact ce valoare de return sa obtine)

Template-uri

- Atentie la cum puneti parametri !!!

App.cpp

```
template <class T>
T Suma(int x, int y)
{
    return (T)(x + y);
}

void main(void)
{
    int x = Suma<int>(1, 2);
    double d = Suma<double>(1.5, 2.4);
}
```

- In acest moment codul compileaza (am specifica prin operatorul <> tipul pe care vrem sa il folosim in template.
- X va avea valoarea 3, iar d va avea valoare tot 3 (1.5 si 2.4 sunt aproximati la int)

Template-uri

- ▶ Template-urile se pot face pe mai multe tipuri

App.cpp

```
template <class T1, class T2, class T3>
T1 Suma(T2 x, T3 y)
{
    return (T1)(x + y);
}

void main(void)
{
    int x = Suma<int>(1, 2);
    double d = Suma<int,double,double>(1.5, 2.4);
}
```

- ▶ Pentru primul caz (Suma<int>) compilatorul traduce in Suma<int,int,int> care face match cu parametri)

Template-uri

- ▶ Template-urile se pot face pe mai multe tipuri

App.cpp

```
template <class T1, class T2, class T3>
T1 Suma(T2 x, T3 y)
{
    return (T1)(x + y);
}

void main(void)
{
    int x = Suma<int,char>(1, 10.5);
    double d = Suma<int,double,double>(1.5, 2.4);
}
```

- ▶ Pentru primul caz (Suma<int,char>) compilatorul traduce in Suma<int,char,double> care face match cu parametri)
- ▶ Compilatorul incearca sa deduca formula pe baza parametrilor daca nu sunt specificat. Trebuie insa specificat tipul de return pentru ca nu poate fi dedus de compilator din parametri

Template-uri

- ▶ Template-urile pentru functii accepta si parametri cu valori default

App.cpp

```
template <class T1, class T2, class T3>
T1 Suma(T2 x, T3 y = T3(5))
{
    return (T1)(x + y);
}

void main(void)
{
    int x = Suma<int, char, int>(10);
}
```

- ▶ Codul de mai sus pune in x valoarea 15 (10+5 → valoarea default pentru y).
- ▶ Utilizarea parametrilor default presupune existenta unui constructor pentru tipul clasei utilizate. Formulari de genul “T3 y = 10” nu sunt valide decat daca tipul T3 accepta egalitate cu un int.



► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ **Template-uri pentru clase**
- ❖ Template-uri specializate
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

Template-uri

- ▶ Template-urile pentru clase se definesc in urmatoarul fel:

App.cpp

```
template <class T>
class MyClass {
    ...
};

sau
template <typename T>
class MyClass {
    ...
};
```

- ▶ Variabila T poate fi folosita atat pentru definirea unor membri a claselor, cat si pentru definirea unor parametri in cadrul unor metode, sau a unor variabile locale in cadrul metodelor.

Template-uri

- ▶ Template-urile pentru clase se definesc in urmatoarul fel:

App.cpp

```
template <class T>
class Stiva
{
    T Lista[100];
    int count;
public:
    Stiva() : count(0) {}
    void Push(T value) { Lista[count++] = value; }
    T Pop() { return Lista[--count]; }
};

void main(void)
{
    Stiva<int> s;
    s.Push(1); s.Push(2); s.Push(3);
    printf("%d", s.Pop());
}
```

- ▶ Codul de mai jos afiseaza 3

Template-uri

- Atentie la cum parametrizati functiile dintr-o clasa bazate pe un template:

App.cpp

```
template <class T>
class Stiva
{
    T Lista[100];
    int count;
public:
    Stiva() : count(0) {}
    void Push(T &value) { Lista[count++] = value; }
    T& Pop() { return Lista[--count]; }
};

void main(void)
{
    Stiva<int> s;
    s.Push(1); s.Push(2); s.Push(3);
    printf("%d", s.Pop());
}
```

- Codul de mai sus nu compileaza. Functia Pop e corecta, dar functia Push trebuie sa primeasca o referinta → **s.Push(1) nu ii da o referinta !!!**

Template-uri

- Atentie la cum parametrizati functiile dintr-o clasa bazate pe un template:

App.cpp

```
template <class T>
class Stiva
{
    T Lista[100];
    int count;
public:
    Stiva() : count(0) {}
    void Push(T &value) { Lista[count++] = value; }
    T& Pop() { return Lista[--count]; }
};

void main(void)
{
    Stiva<int> s;
    for (int tr = 1; tr <= 3; tr++)
        s.Push(tr);
    printf("%d", s.Pop());
}
```

- Codul compileaza si afiseaza 3.

Template-uri

- La fel ca si la functii, clasele pot avea mai multi parametri templetizati:

App.cpp

```
template <class T1,class T2>
class Pereche
{
    T1 Key;
    T2 Value;
public:
    Pereche() : Key(T1()), Value(T2()) {}
    void SetKey(const T1 &v) { Key = v; }
    void SetValue(const T2 &v) { Value = v; }
};

void main(void)
{
    Pereche<const char*, int> p;
    p.SetKey("nota_examen");
    p.SetValue(10);
}
```

- “Pereche() : Key(T1()), Value(T2())” → apeleaza constructorul implicit pentru Key si Value (pentru “p” Key va fi NULL si Value va fi 0).

Template-uri

- La fel ca si la functii, clasele pot avea mai multi parametrii templetizati:

App.cpp

```
template <class T1,class T2>
class Pereche
{
    T1 Key;
    T2 Value;
public:
    Pereche() : Key(T1()), Value(T2()) {}
    Pereche(const T1 &v1, const T2& v2) : Key(v1), Value(v2) {}
    void SetKey(const T1 &v) { Key = v; }
    void SetValue(const T2 &v) { Value = v; }
};

void main(void)
{
    Pereche<const char*, int> p("nota_examen",10);
}
```

- Se poate define si un constructor explicit pentru o clasa templetizata. In cazul de mai sus constructorul primeste un string (const char *) si un numar.

Template-uri

► Un exemplu mai complex:

App.cpp

```
template <class T>
class Stiva
{
    ...
    void Push(T* value) { Lista[count++] = (*value); }
    T& Pop() { return Lista[--count]; }
};

template <class T1,class T2>
class Pereche
{
    ...
};

void main(void)
{
    Stiva<Pereche<const char*, int>> s;

    s.Push(new Pereche<const char*, int>("nota_asm", 10));
    s.Push(new Pereche<const char*, int>("nota_poo", 9));
    s.Push(new Pereche<const char*, int>("nota_c#", 8));
}
```

Template-uri

- Macro-urile pot fi folosite impreuna cu template-urile:

App.cpp

```
#define P(k,v) new Pereche<const char*, int>(k, v)  
#define Stack Stiva<Pereche<const char*, int>>
```

```
template <class T>  
class Stiva  
{  
    ...  
};  
template <class T1,class T2>  
class Pereche  
{  
    ...  
};  
  
void main(void)  
{  
    Stack s;  
    s.Push(P("nota_asm", 10));  
    s.Push(P("nota_poo", 9));  
    s.Push(P("nota_c#", 8));  
}
```

Template-uri

- ▶ Template-urile pentru clase accepta si parametrii fara tip (constant)

App.cpp

```
template <class T,int Size>
class Stiva
{
    T Lista[Size];
    int count;
public:
    Stiva() : count(0) {}
    void Push(const T& value) { Lista[count++] = (value); }
    T& Pop() { return Lista[--count]; }
};
void main(void)
{
    Stiva<int, 10> s;
    Stiva<int, 100> s2;
    for (int tr = 0; tr < 5; tr++)
        s.Push(tr);
}
```

- ▶ In cazul de fata, “s” va avea 10 elemente, iar “s2” 100 de elemente
- ▶ Se vor crea doua clase diferite in acest caz (cu 10 si 100 de elemente)

Template-uri

- ▶ Template-urile pentru clase accepta si parametrii fara tip (constant)

App.cpp

```
template <class T, int Size = 100>
class Stiva
{
    T Lista[Size];
    int count;
public:
    Stiva() : count(0) {}
    void Push(const T& value) { Lista[count++] = (value); }
    T& Pop() { return Lista[--count]; }
};

void main(void)
{
    Stiva<int, 10> s;
    Stiva<int> s2;
    for (int tr = 0; tr < 5; tr++)
        s.Push(tr);
}
```

- ▶ In cazul de fata, “s” va avea 10 elemente, iar “s2” 100 de elemente. In cazul lui s2 se foloseste valoarea default pentru Size

Template-uri

- ▶ Template-urile pentru clase accepta valori default si pentru tipuri

App.cpp

```
template <class T = int>
class Stiva
{
    T Lista[100];
    int count;
public:
    Stiva() : count(0) {}
    void Push(const T& value) { Lista[count++] = (value); }
    T& Pop() { return Lista[--count]; }
};
void main(void)
{
    Stiva<double> s;
    Stiva<> s2;
}
```

- ▶ In cazul de fata, “s” va fi o lista de 100 de double-uri, iar s2 (pentru ca nu am specificat tipul) va fi de tipul int.

Template-uri

- ▶ Template-urile pentru clase accepta valori default si pentru tipuri

App.cpp

```
template <class T = int>
class Stiva
{
    T Lista[100];
    int count;
public:
    Stiva() : count(0) {}
    void Push(const T& value) { Lista[count++] = (value); }
    T& Pop() { return Lista[--count]; }
};
void main(void)
{
    Stiva<double> s;
    Stiva s2;
}
```

- ▶ Codul de mai sus NU compileaza - chiar daca tipul parametrizat T este default int, trebuie specificat cand definit o variabila de tipul Stiva ca este un template (trebuie sa folosim <>)

Template-uri

- Clasele pot folosi si ele la randul lor functii template

App.cpp

```
class Integer
{
    int value;
public:
    Integer() : value(0) {}
    template <class T>
    void SetValue(T v) { value = (int)v; }
};

void main(void)
{
    Integer i;
    i.SetValue<float>(0.5f);
    i.SetValue<double>(1.2);
    i.SetValue<char>('a');
}
```

- In cazul de mai sus, Integer are 3 functii templetizate (pentru float, double si char)

Template-uri

- ▶ Clasele pot folosi si ele la randul lor functii template

App.cpp

```
class Integer
{
    int value;
public:
    Integer() : value(0) {}
    template <class T>
    void SetValue(T v) { value = (int)v; }
};

void main(void)
{
    Integer i;
    i.SetValue(0.5f);
    i.SetValue('a');
}
```

- ▶ Specificarea tipului in template nu este neaparat obligatorie. In cazul de fata, Integer are doua functii templetizate (una pentru float → dedusa de compilator din 0.5f) si alta pentru char (dedusa din 'a')



► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ **Template-uri specializate**
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

Template-uri

- ▶ Template-urile insa vin si cu o serie de limitari
 - ▶ Cea mai importanta este ca fiecare functie dintr-o clasa templetizata are exact acelasi comportament (singurul lucru care difera este tipul parametrilor)
 - ▶ De exemplu daca definim o functie Suma cu doi parametri x si y , iar in corpul functiei returnam $x+y$, acest lucru nu il putem schimba daca x si y sunt de tipul `char` (de exemplu sa returnam $x*y$)
- ▶ Acest lucru se poate insa face, folosind template-uri specializate
- ▶ Template-urile specializate reprezinta o modalitate cand pentru o clasa mai scriem inca o definitie in care suprascriem codul initial din template cu altul care sa fie specific pentru anumiti parametri de un anumit tip

Template-uri

- Fie urmatorul exemplu:

App.cpp

```
template <class T>
class Number
{
    T value;
public:
    void Set(T t) { value = t; };
    int  Get() { return (int)value; }
};

void main(void)
{
    Number<int> n1;
    n1.Set(5);
    Number<char> n2;
    n2.Set('7');
    printf("n1=%d, n2=%d", n1.Get(), n2.Get());
}
```

- Codul compileaza - dar afiseaza pe ecran “n1=5, n2=55” desi noi pe n2 l-am initializat cu ‘7’ → si am dori sa afiseze n2 = 7

Template-uri

- Fie urmatorul exemplu:

App.cpp

```
template <class T>
class Number
{
    T value;
public:
    void Set(T t) { value = t; };
    int  Get() { return (int)value; }
};

template <>
class Number <char>
{
    char value;
public:
    void Set(char t) { value = t-'0'; };
    int  Get() { return (int)value; }
};

void main(void)
{
    Number<int> n1;
    n1.Set(5);
    Number<char> n2;
    n2.Set('7');
    printf("n1=%d, n2=%d", n1.Get(), n2.Get());
}
```

Template
specializat
pentru char

- Codul functioneaza corect si afiseaza n1=5, n2=7

Template-uri

- ▶ Template-urile specializate se aplica si la functii in acelasi mod:

App.cpp

```
template <class T>
int ConvertToInt(T value) { return (int)value; }

template <>
int ConvertToInt<char>(char value) { return (int)(value-'0'); }

void main(void)
{
    int x = ConvertToInt<double>(1.5);
    int y = ConvertToInt<char>('4');
}
```

Template specializat
pentru char

- ▶ Codul compileaza - x va avea valoare 1, iar y va avea valoarea 4



► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ Template-uri specializate
- ❖ **Membri statici in clase templetizate**
- ❖ Functii “friend” in clasele templetizate

Template-uri

- Clasele templetizate pot avea si membri statici:

App.cpp

```
template<class T>
class Number
{
    T Value;
public:
    static int Count;
};

int Number<int>::Count = 10;
int Number<char>::Count = 20;
int Number<double>::Count = 30;

void main(void)
{
    Number<int> n1;
    Number<char> n2;
    Number<double> n3;

    printf("%d,%d,%d", n1.Count, n2.Count, n3.Count);
}
```

Template-uri

- Clasele templetizate pot avea si membri statici:

App.cpp

```
template<class T>
class Number
{
    T Value;
public:
    static T Count;
};

int Number<int>::Count = 10;
char Number<char>::Count = 'a';
double Number<double>::Count = 30;

void main(void)
{
    Number<int> n1;
    Number<char> n2;
    Number<double> n3;

    printf("%d,%c,%lf", n1.Count, n2.Count, n3.Count);
}
```




► Macro-uri

► Template-uri

- ❖ Template-uri pentru functii
- ❖ Template-uri pentru clase
- ❖ Template-uri specializate
- ❖ Membri statici in clase templetizate
- ❖ Functii “friend” in clasele templetizate

Template-uri

- Clasele templetizate pot avea si functii friend:

App.cpp

```
template<class T>
class Number
{
    T Value;
    int IntValue;
public:
    friend void Test(Number<T> &t);
};
void Test(Number<double> &t)
{
    t.Value = 1.23;
}
void Test(Number<char> &t)
{
    t.Value = 0;
}
void main(void)
{
    Number<char> n1;
    Number<double> n2;
    Test(n1);
    Test(n2);
}
```

Template-uri

- Clasele templetizate pot avea si functii friend:

App.cpp

```
template<class T>
class Number
{
    T Value;
    int IntValue;
public:
    friend void Test(Number<T> &t)
    {
        t.Value = 5;
    }
};

void main(void)
{
    Number<char> n1;
    Number<double> n2;
    Test(n1);
    Test(n2);
}
```