# Tutoriat 6
# Template

1. Ce este?
   Presupune scrierea unei singure clase/funcții a cărei comportament este asemănător și se modifică, doar dacă se modifică și un anumit tip de date.

   Obs:
   Template este o formă de **polimorfism** la compilare.

2. La ce se folosește?
   Template este un instrument prin care se poate evita rescrierea unor blocuri de cod.

3. Despre **typename**
   Poate fi înlocuit începând cu standardul C++17 cu **class(cel folosit în definirea template)** fără vreo diferență semnificativă.

4. Funcții template
   a. Pași la compilare:

      1. Când e întâlnită o funcție template, este compilată, fără a se ține cont de tipul de date necunoscut.
      2. În momentul în care se apelează o funcție template, compilatorul creează o nouă funcție obișnuită în care tipul de date necunoscut este înlocuit cu cel specificat în apel.

      f<int>(3) ===> void f (int x) {...}
      f<char>('c') ===> void f (char x) {...}

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
void g (int x) {
    cout << "Functie obisnuita"<<endl;
}
int main () {
    f<int>(3); // Functie template
    f<char>('c'); // Functie template
    g(2); // Functie obisnuita
```

```
    return 0;
}
```

b. Specializarea funcțiilor template

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
template <>
void f (int x) {
    cout << "Functie  specializata"<<endl;
}
int main () {
    f(3); // Functie  specializata
    f<int>(3); // Functie  specializata
    f('c'); // Functie template
}
```

c. Prioritatea la supraîncarcare(overloading)

Cum procedează compilatorul când caută o funcție care se potrivește cu un apel:

1. Se caută o **funcție normală** care să aibă parametrii potriviți.
2. Dacă nu s-a găsit la punctul 1, se caută o **specializare** cu parametrii potriviți.
3. Dacă nici punctul 2 nu a furnizat un rezultat, se caută **o funcție template** cu numărul de parametrii potriviți.
4. Dacă nici punctul 3 nu a furnizat un rezultat, se întoarce o **eroare la compilare.**

Mai pe scurt, o ordine de prioritate ar fi:

1. Funcțiile normale
2. Funcțiile specializate
3. Funcțiile template

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
template <>
void f (int x) {
    cout << "Functie specializata"<<endl;
}
void f (char c) {
```

```
    cout << "Functie normala (char)"<<endl;
}
void f (int c) {
    cout << "Functie normala (int)"<<endl;
}
int main () {
    f(3); // Functie normala (int)
    f('c'); // Functie normala (char)
    f<int>(3); // Functie specializata
    f<char>('c'); // Functie template
}
```

5. Clase template
    a. Exemplu

Obs:
Spre deosebire de funcțiile template, aici este obligatorie specificarea tipului de date la declararea unui obiect (între <>).

```
template <typename T>
class A {
// clasa template
    T x;
    int y;
public:
    A(){cout<<"A"<<endl;}
};
class B {
// clasa obisnuita fara template
    char x;
    int y;
public:
    B(){cout<<"B"<<endl;}
};
int main () {
    A<int> a1; // A
    A<char> a2; // A
    B b; // B
    return 0;
}
```

    b. Specializarea claselor template

```
template <typename T>
class A {
    T x;
```

```
public:
    A() { cout << "A template"; }
};
template <>
class A<int> {
    int x;
public:
    A<int>() { cout << "A specializata "; }
};
int main () {
    A<int> a1; //A specializata
    A<char> a2; // A template
}
```

c. Metode template

```
template <typename T>
class A {
    T x;
public:
    T getX () const;
    void setX (T);
};
template <typename T>
T A<T>::getX () const {
    return x;
}
template <typename T>
void A<T>::setX (T _x) {
    x = _x;
}
int main(){
    A<int> object;
    object.setX(2);
    cout<<object.getX(); // 2
}
```