Preprocesare

Preprocesarea este prelucrarea textului sursa al programului inaintea etapei de compilare. In limbajul C++ preprocesarea se realizeaza cu ajutorul directivelor de preprocesare. Acestea sunt recunoscute de compilator prin prezenta caracterului “#”.

Includere fisiere

Includerea fisierelor sursa se realizeaza prin intermediul directivei *#include* care are doua forme:

#include <fisier_sursa>
#include "fisier_sursa"

Ambele forme au ca efect includerea totala a fisierului sursa in fisierul care contine directiva. Diferentele intre cele doua forme se refera la locatia unde este cautat fisierul sursa. In cazul in care se foloseste prima varianta, fisierul este cautat in directoarele standard (specificate prin optiuni sau prin variabile de mediu in functie de compilator). In in cazul celei de-a doua varianta fisierul este cautat intai in directorul curent, iar dupa aceea, daca nu este gasit, in directoarele standard. Forma cu „” permite si specificarea caii complete catre fisierul inclus; in acest caz, nu se mai face cautarea si in directoarele standard.

Exemple:

```
// include fisierul stdio.h
// din directoarele standard
#include <stdio.h>

// include fiserul ListeSimple.h;
// cautarea se face intai in directorul
// curent si dupa aceea in directoarele standard
#include “liste.h”


// include fisierul Masive.cpp din directorul
// c:\Biblioteci; daca fisierul nu exista nu
// mai este cautat in alta parte si se genereaza
// o eroare de compilare
#include "C:\Biblioteci\Masive.cpp"
```

Constante simbolice

Definirea constantelor simbolice se face prin intermediul directivei *#define* cu sintaxa:

#define simbol valoare

Directiva are ca efect inlocuirea tuturor aparitiilor lui *simbol* in codul sursa (cu exceptia aparitiilor in cadrul unor constante de tip sir, in comentarii sau in componenta unui alt identificator) cu *valoare* inaintea compilarii textului sursa.

v*aloare* este cosiderata intreaga portiune pana la sfarsitul liniei. Daca se doreste definirea unei valori pe mai multe linii, atunci se va folosi caracterul \ la sfarsitul fiecarei linii ce urmeaza a fi continuata. In cazul in care *valoare* lipseste, textul specificat prin *simbol* va fi eliminat din codul sursa.

O constanta simbolica poate fi redefinita in cadrul celuiasi fisier folosind inca o data directiva define: *#define simbol alta_valoare*.

Valabilitatea unei definiri se incheie in urmatoarele cazuri:

- la sfarsitul fisierului sursa
- la invalidarea simbolului prin intermediul directivei *#undef simbol*

Exemplu de utilizare:

```
// definire parametru
#define DIM_VECTOR 20

// definire tip vector
#define TIP double

// definire mesaj
#define MESAJ "Calcul suma"

// definire cod pe mai multe linii
#define SEPARATOR cout \
      << "----------------" \
      << endl;


TIP suma(TIP v[], int n)
{
      // validare lungime
      if (n > DIM_VECTOR)
            // DIM_MAX nu este inlocuit in sirul de caractere
            // cau in comentariu
            cout << "dimensiunea este mai mare ca DIM_VECTOR" << endl;

      // calcul suma
      TIP suma = 0;
      for (int i = 0; i < n; i++)
            suma += v[i];

      return suma;
}

void main()
{
      // declarare vector folosind
      // simbolurile specificate
```

```
	TIP v[DIM_VECTOR] = {1.1, 3.23, 6.62};
	int n = 3;

	// utilizare simboluri pentru mesaj
	cout << MESAJ;

	// inserare cod prin preprocesor
	SEPARATOR;

	// apel functie
	suma(v, 3);
}
```

Codul sursa rezultat in urma preprocesarii va fi:

```
// definire parametru

// definire tip vector

// definire mesaj

// definire cod pe mai multe linii

double suma(double v[], int n)
{
	// validare lungime
	if (n > 20)
		// DIM_MAX nu este inlocuit in sirul de caractere
		// cau in comentariu
		cout << "dimensiunea este mai mare ca DIM_VECTOR" <<
endl;

	// calcul suma
	double suma = 0;
	for (int i = 0; i < n; i++)
		suma += v[i];

	return suma;
}

void main()
{
	// declarare vector folosind
	// simbolurile specificate
	double v[20] = {1.1, 3.23, 6.62};
	int n = 3;

	// utilizare simboluri pentru mesaj
	cout << "Calcul suma";

	// inserare cod prin preprocesor
	cout << "----------------" << endl;;

	// apel functie
	suma(v, 3);
}
```

Compilare conditionata

Compilarea conditionata permite includerea/excluderea unor sectiuni din codul sursa in functie de anumite criterii. In C++ se folosesc directivele de compilare conditionata *#if*, *#else*, *#elsif* si *#endif*.

Sintaxa este:

#if expresie_1
sectiune_1
#elif expresie_2
sectiune_2
…
#else
sectiunie_n
#endif
unde:

- *expresie_n* este o expresie formata din constante sau simboluri definite cu *#define*; expresia este considerata adevarata daca este diferita de 0;
- partile *#elif* si *#else* sunt optionale;
- *sectiune_n* reprezinta o secventa de cod de inclus in codul sursa al programului in functie de valoarea expresiei;

In cadrul expresiilor se poate folosi cuvantul cheie **defined** pentru a testa daca un anumit simbol de preprocesare a fost definit (de exemplu: #if defined(DIM_MAX) ….). In cazul in care se doreste doar testarea existentei unei variabile se pot folosi directivele *#ifdef* (echivalent cu *#if defined*) sau #ifndef (echivaled cu *#if !defined*)

Exemplu:

```
#if defined DIM_VECTOR && DIM_VECTOR < 20
      // este inclus in codul sursa doar
      // daca simbolul DIM_VECTOR este
      // definit are o valoare mai mica de 20
      cout << "Suma vector de intregi cu " << DIM_VECTOR << "elemente";
#else
      // va fi inclus daca DIM_VECTOR nu
      // este dafinit sau este mai mare
      // de 20
      cout << "Vectorul trebuie sa aiba mai putin de 20 elemente." <<
endl;
      return;
#endif
```

Cele mai uzuale utilizari pentru directivele de compilare conditionata sunt:

a) compilarea unor portiuni din codul sursa doar in pentru versiunile de depanare:

```
int CalculSuma(int v[], int n)
{
// se va afisa doar in versiunea de DEBUG
#if _DEBUG
	cout << "Calcul suma pentru vector cu "
		<< n << " elemente." << endl;
#endif

	// calcul suma
	int suma = 0;
	for (int i = 0; i < n; i++)
		suma += v[i];
#if _DEBUG
	cout << "Suma calculata este " << suma << endl;
#endif

	return suma;
}
```

Mesajele vor fi afisate doar daca programul este compilat in versiunea de depanare. Simbolul _DEBUG este definit automat de Visual Studio in cazul in care se selecteaza versiunea Debug din *Build->Set active configuration.*

b) evitarea dublei incluziuni a fisierelor header:

```
#ifndef LISTE_H
#define LISTE_H

// definitii obiecte
// ...

#endif //LISTE_H
```

In acest fel se poate evita includerea fisierului de mai multe ori. La prima includere conditia este *true* si se va include continutul format din declararea simbolului si codul sursa. La cea de-a doua includere conditia va fi *false* si continutul nu va mai fi inclus.

Macrodefinitii

Macrodefinitiile sunt secvente de cod parametrizate care pot fi inlocuite in textul sursa. Pentru definirea lor se foloseste o extensie a directivei *#define*:

#define macro(param) corp

Unde:

- macro – numele macrodefinitiei
- param – lista de parametri, fara tip
- corp – corpul macrodefinitiei

Macrodefinitiile sunt expandate in textul sursa inainte de compilare in doua etape:

- apelul macrodefinitiei din codul sursa este inlocuit cu corpul acesteia
- parametrii macrodefinitiei sunt inlocuiti cu valorile primite ca parametrii

Exemplu:

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

// exemplu de utilizare
int a = 7, b = 10, c;
c = MAX(a,b);
```

Desi apelul de macrodefinitii este similar unui apel de functie, cele doua constructii sintactice nu sunt echivalente. In cazul macrodefinitiilor, parametrii nu sunt evaluati inaintea apelului, fapt care poate crea confuzii:

```
// macrodefinitie
#define PATRAT(x) x*x;

int x = 2, y;

// problema 1: folosirea parantezelor pentru a impune prioritatilor
y = PATRAT(x+5) // => va fi expandata ca y = x+5*x+5; <=> y = 17

// problema 2: constructii cu efecte colaterale
int x = 2;
y = PATRAT(++x) // => va fi expandata ca y = ++x*++x de unde
// va rezulta x = 4 (preincrementat de 2 ori) si y=16
```

In cadrul corpului macrodefinitiilor de pot utiliza doi operatori speciali:
#param → operator de sirificare; converteste parametrul intr-o constanta de tip sir de caractere
##param → operator de concatenare pentru simboluri; permite crearea de simboluri (nume de functii sau de variabile pe baza parametrului)

Exemple de utilizare pentru macrodefinitii

1. Macrodefinitie pentru afisarea de mesaje de atentionare in cazul in care apare o conditie. Mesajul de va afisa doar in cazul in care aplicatia este compilata in modul „Debug”:

```
#ifdef _DEBUG
	// in cazul in care compilam in mod debug afisam atentionarile
	#define ATENTIONARE(EXPRESIE) \
		if (EXPRESIE) \
			cerr << "ATENTIE: "#EXPRESIE << endl;
#else
	// in modul release le ignoram
	#define ATENTIONARE(EXPRESIE)
#endif


// exemplu de utilizare
double CalculMedie(int note[], int numarNote)
{
	ATENTIONARE(numarNote == 0)

	double suma = 0;
	for (int i = 0; i < numarNote; i++)
		suma += note[i];
```

```
	return suma / numarNote;
}
```

2. Generare de functii similare diferite tipuri de date folosind macrodefinitii:

```
#define FUNCTIE_SUMA(TIP) \
	TIP suma_##TIP(TIP vector[], int nrElem)	\
	{
	\
		TIP suma = 0;	\
		for (int i = 0; i < nrElem; i++)	\
			suma += vector[i];	\
		return suma;	\
	}

// exemplu de utilizare pentru generarea de functii
// (va genera functiile suma_int si suma_double
FUNCTIE_SUMA(int)
FUNCTIE_SUMA(double)
```

3. Evitarea repetarii unor constructii similare:

```
#define CICLU(NR_ITERATII,CORP) \
	for (int i = 0; i < NR_ITERATII; i++) { CORP; }

// exemplu de utilizare
CICLU(10,cout << "Un mesaj..." << endl;)
```