Wykład 7

7. Preprocesor i dyrektywy kompilatora

7.1. Makrodefinicje proste

7.2. Makrodefinicje parametryczne

7.3. Usuwanie definicji makra

7.4. Włączanie innych zbiorów do tekstu programu

7.5. Dyrektywy kompilacji warunkowej

7.6. Kompilacja modułów

7.7. Dyrektywa #pragma

7.8. Wybrane operacje na jednostkach leksykalnych

Preprocesor języka C++ umożliwia:

- definiowanie makrodefinicji pozwalających uprościć postać kodu źródłowego programu;
- dołączanie tekstów plików, np. zbiorów nagłówkowych zawierających prototypy standardowych funkcji bibliotecznych;
- sterowanie przebiegiem kompilacji.

7.1. Makrodefinicje proste

Makrodefinicje proste mają następującą postać:

#define *identyfikator makro_tekst_lub_wartość*

Każde wystąpienie *identyfikatora* w tekście programu zostanie zastąpione ciągiem *makro_tekst_lub_wartość*. Na końcu makra nie dajemy średnika, gdyż znalazłby się on w jego rozwinięciu.

Za pomocą makrodefinicji prostych można:

- definiować stałe,
- zastępować zarezerwowane słowa lub symbole innymi,
- tworzyć identyfikatory typów danych przy użyciu standardowych typów danych,
- tworzyć skróty poleceń.

Przykłady.

```
// definiowanie stałych

#define IMIE ”Adam”

#define PI 3.14159

#define TRUE 1
```

```
// definiowanie słów kluczowych

#define BEGIN {

#define END }

#define POCZ main()

// definiowanie typów danych

#define BOOLEAN char

#define BYTE unsigned char

#define REAL double

// tworzenie skrótów poleceń

#define WRITE printf

#define  READ scanf

#define ReadKey getch()

#define WRITELN printf(”\n”)
```

Makrodefinicje umożliwiają zdefiniowanie nowej nazwy dla standardowego typu danych, ale nie realizują tego samego co specyfikator typu typedef, który pozwala zadeklarować nowy typ danych.

```
Np.  typedef  int  * WINT;      // deklaracja typu WINT – wskaźniki int *

WINT p1, p2;    // dwie zmienne typu WINT
```

Ostatnia definicja odpowiada definicji:

```
WINT p1;       // int * p1;

WINT p2;       // int * p2;
```

Zmienne p1 i p2 są wskaźnikami na int .

Natomiast deklaracja

```
#define WINT int *
```

zastępuje w kodzie źródłowym WINT wyrażeniem int *. Dlatego definicja postaci

```
WINT p1, p2;
```

odpowiada definicji

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

Zmienna p1 jest wskaźnikiem na int, natomiast zmienna p2 jest typu int.

7.2. Makrodefinicje parametryczne

Do makra można przekazywać parametry. Ogólna postać dyrektywy *#define* wykorzystującej parametry jest następująca:

#define *identyfikator*(id_par1, id_par2, ...) *wyrażenie_makro*

Należy pamiętać, że w definicji makra nawias otwierający musi następować bezpośrednio po identyfikatorze makrodefinicji.

Wywołanie makrodefinicji przypomina wywołanie funkcji. Identyfikatory parametrów zostaną zastąpione nazwami parametrów. Należy jednak pamiętać, że w odróżnieniu od funkcji nie jest sprawdzana zgodność typu parametrów makra i przekazywanych argumentów.

Makrodefinicja DZIEL:

```
#define DZIEL(a,b) ((a) / (b))
```

Wywołanie DZIEL(x,y)

W efekcie rozwinięcia otrzymamy tekst ((x) / (y)). Umieszczenie tekstu w nawiasach pozwala ustrzec się od błędów podczas przekazywania wyrażeń.

W przypadku, gdyby zdefiniowano DZIEL(a,b) (a / b) wówczas rozwinięcie wywołania

DZIEL(x+1, y+2)

prowadziłoby do wyrażenia x+1 / y+2 = x + 1/y + 2. Jest to wyrażenie różne od oczekiwanego.

Wywoływanie makrodefinicji

Wywołanie makrodefinicji jest podobne do wywołania funkcji. Różnice występują w tych przypadkach, w których wyrażenie będące parametrem wywołania jest obliczane w kilku etapach. Np.

```
#define KWADRAT(a)((a)*(a))

int i=5;   long w;

w = KWADRAT(i++);      // w = ((i++)*(i++))
```

Wartość zmiennej i zostanie zwiększona dwurotnie, tzn. i = 7, w = 5*6 zamiast i = 6, jak można by oczekiwać tylko na podstawie wywołania makra.

```
#define MIN(a,b) (a) < (b) ? (a) : (b)
```

MIN(x++,y++) zostanie rozwinięte: (x++) < (y++) ? (x++) : (y++).

Zmienna o mniejszej wartości zostanie zwiększona dwa razy, natomiast zmienna o większej wartości jednokrotnie.

Np. x=0 i y=2;
(x++) < (y++); // (0) < (2); x++ = 1; y++ = 3;
wartość wyrażenia w = (x++) < (y++) ? (x++) : (y++) = x =1;
następnie x++ = 2;
ostatecznie: w=1; x=2; y=3.

Wniosek: w odwołaniach do makrodefinicji należy unikać przekazywania parametrów, których wartość jest obliczana wieloetapowo.

Przykład 7.1. Wykorzystanie makrodefinicji.

```
#define IMIE "Adam"

// makro TRUE FALSE
#define boolean(x) ((x) ? "TRUE" : "FALSE")

// jednoliniowe pseudofunkcje
#define abs(x) ( ( (x) >= 0 ) ? (x) : (-x) )
#define max(x,y) ( ( (x) > (y) ) ? (x) : (y) )
#define min(x,y) ( ( (x) > (y) ) ? (y) : (x) )
#define kwadrat(x) ( (x) * (x) )

// testowanie znakow
#define mala(c) (c >='a' && c <= 'z')
#define duza(c) (c >='A' && c <= 'Z')
#define cyfra(c) (c >='0' && c <= '9')

// konwersja znakow
#define mala_litera(c) (c - 'A'+'a')
#define duza_litera(c) (c - 'a'+'A')

void main(void)
{  char zn;     double x,y;     int ww;

   clrscr();  printf("\nImie : %s", IMIE);    x = -2.0;  y = 1.0;
   printf("\n\nLiczba x: %6.4lf", x);
   printf("\nABS(x) : %6.4lf ",abs(x));
   printf("\nKWADRAT(x) : %6.4lf ",kwadrat(x));
   printf("\n\nLiczba y: %6.4lf", y);
   printf("\nMAX(x,y) : %6.4lf ",max(x,y));         //1.0000
   printf("\nMIN(x,y) : %6.4lf ",min(x,y));         //-2.0000

   getch();    clrscr();    zn = 'S';     ww = mala(zn);
   printf("\nZnak %c jest mala litera %s", zn, boolean(ww));
   ww = duza(zn);
   printf("\nZnak %c jest duza litera %s", zn, boolean(ww));
   ww = cyfra(zn);
   printf("\nZnak %c jest cyfra %s", zn, boolean(ww));

   printf("\n\nMala litera %c jest %c", zn, mala_litera(zn));
}
```

7.3. Usuwanie definicji makra

Dyrektywa **#undef** umożliwia usunięcie definicji makra.

#undef id_makrodefinicji

Np.

```
#define IMIE ”Adam”
puts(IMIE);

#undef IMIE

#define IMIE ”Tomek”
puts(IMIE);
```

7.4. Włączanie innych zbiorów do tekstu programu

Dyrektywa #include pozwala włączyć do aktualnie kompilowanego zbioru zawartość innych zbiorów. Najczęściej są to zbiory nagłówkowe, zawierające makrodefinicje oraz deklaracje funkcji bibliotecznych lub zdefiniowanych w innych modułach programu.

Dyrektywa

#include <nazwa_zbioru>

jest wykorzystywana do włączania zbiorów nagłówkowych związanych ze standardowymi bibliotekami języka. Zbiór będzie poszukiwany w kartotece zawierającej standardowe zbiory systemu, która jest określona za pomocą opcji *Options*|*Directories*|*Includedirectories.*

Dyrektywa

#include ”nazwa_zbioru” lub dyrektywa

#include ”ścieżka_dostępu”

powoduje, że zbiór nagłówkowy jest poszukiwany w aktualnej kartotece (kartotece określonej przez ścieżkę dostępu), a jeżeli nie zostanie znaleziony to jest poszukiwany w kartotece zawierającej standardowe zbiory systemu.

Np.

```
// początek  plik.h

void funkcja1(int);
void funkcja2(double);

//  koniec plik.h

//  początek mplik.cpp

#include  ”plik.h”              // dołączenie plik.h

void main()
{  ...  }
void funkcja1(int)
{ ... }
void funkcja2(double)
{ ... }
//  koniec  mplik.cpp
```

7.5. Dyrektywy kompilacji warunkowej

Dyrektywy kompilacji warunkowej #if, #elif umożliwiają wykonanie lub nie kompilacji pewnych fragmentów kodu źródłowego w zależności od spełnienia określonych warunków.

Składnia dyrektyw kompilacji warunkowej jest podobna do konstrukcji if-else:

```
#if  wyr_stałe_1
       sekwencja instrukcji kompilowanych jeżeli wyr_stałe_1
       ma wartość różną od zera (prawda)
#elif  wyr_stałe_2
        sekwencja instrukcji kompilowanych jeżeli wyr_stałe_2
        ma wartość różną od zera (prawda)
...
#elif  wyr_stałe_N
        sekwencja instrukcji kompilowanych jeżeli wyr_stałe_N
        ma wartość różną od zera (prawda)
#else  sekwencja instrukcji kompilowanych jeśli żaden z powyższych
        warunków nie jest prawdziwy
#endif
```

Przetwarzane instrukcje są zawarte między dyrektywą #if i #endif. Wyrażenia stałe użyte w dyrektywach #if oraz #elif muszą mieć wartość całkowitą. Jeśli któreś z wyrażeń wyr_stale_i jest prawdziwe, to odpowiadający mu kod jest analizowany, natomiast fragmenty kodu związane z pozostałymi wyrażeniami są pomijane.

Dyrektywy **#elif** i **#else** są opcjonalne. Każda dyrektywa powinna zaczynać się od nowej linii.

Przykład 7.2. Wykorzystanie dyrektyw warunkowych.

```
#define WYK 4

#pragma argsused    // nie wysyłaj ostrzeżenia jeżeli wewnątrz
                    // funkcji nie są wykorzystywane jej argumenty
double potega(double x)
{
 #if  WYK < 0
 #error Ujemny wykladnik!   // przy próbie kompilacji pojawi się
    // komunikat o błędzie Error directive Ujemny wykładnik!
 #elif  WYK == 0
 return 1;
 #elif  WYK == 1
 return x;
 #elif  WYK == 2
 return x*x;
 #else
   int k=0;
   double s = 1;

   for (k=0; k<WYK; k++) s*=x;
   return s;
 #endif
}

void main(void)
{  double x = 2;

  clrscr();
  cout << potega(x) << endl;
  getch();
}
```

Dyrektywy umożliwiające stwierdzenie czy dany identyfikator został już zdefiniowany dyrektywą #define i w zależności od tego wykonanie lub nie kompilacji fragmentu programu.

```
#ifdef  IDENT
        <instrukcje podlegające kompilacji jeżeli wcześniej
          wystąpiła definicja  #define IDENT ... >
#endif

#ifndef  IDENT
        <instrukcje podlegające kompilacji jeżeli wcześniej
          nie wystąpiła definicja  #define IDENT ... >
#endif
```

Np.

```
#ifdef  WYK
cout << "WYK = " << WYK << endl;
#endif
```

Możliwy jest również alternatywny sposób sprawdzania czy dany identyfikator został zdefiniowany.

```
#if defined (IDENT)
        <instrukcje podlegające kompilacji jeżeli wcześniej
          wystąpiła definicja  #define IDENT ... >
#endif
```

Defined jest operatorem preprocesora, który zwraca wartość 1, jeśli jego argument jest zdefiniowany, oraz wartość 0 w przeciwnym przypadku. Zaletą przedstawionej formy jest to, iż można ją stosować w połączeniu z dyrektywą #elif.

Np.

```
#if defined(MODEL1)
   sp = farcoreleft();
 #elif defined(MODEL2)
   sp = coreleft();
 #else
 #error Blad - Bad memory model
 #endif
```

Dyrektywa #error przerywa kompilację i wypisuje komunikat Blad – Bad memory model, jeśli nie zdefiniowano MODEL1 lub MODEL2.

7.6. Kompilacja modułów

Dobrze zaprojektowany zbiór nagłówkowy powinien być zabezpieczony przed wielokrotnym włączeniem go do tekstu programu.

// plik.h - zbiór nagłówkowy

```
#ifndef PLIK_H

#define PLIK_H
        < deklaracje pliku  plik.h >
#endif
```

Jeśli makrodefinicja PLIK_H nie została jeszcze zdefiniowana wówczas kompilator musi przeanalizować wszystkie dyrektywy i instrukcje znajdujące się między #ifndef a #endif. W wyniku przetworzenia wspomnianego obszaru zbadane zostaną wszystkie deklaracje znajdujące się w zbiorze nagłówkowym plik.h oraz zdefiniowane zostanie makro PLIK_H. Przy ponownym włączeniu zbioru plik.h jego zawartość nie będzie analizowana.

Sposób wykorzystania dyrektyw kompilacji warunkowej w programach wielomodułowych (prog. główny + 4 moduły) ilustruje kolejny przykład.

Przykład. 7.3. Kompilacja programów wielomodułowych.

Wariant 1. Dołączanie kodu modułów do programu za pomocą dyrektywy #include.

Wariant 2. Tworzenie projektu (zbioru modułów do kompilacji); łączenie kodów wynikowych modułów.

```
/*  plik nagłówkowy zawierający deklaracje typów oraz prototypy funkcji */

#ifndef PROT.H
#define PROT.H
void pisz_wynik(double);                    // prototyp funkcji pisz_wynik
double suma1(double, double);
void    suma2(double, double, double *);
void    suma3(double, double, double &);
#endif
```

```
/*   plik  pisz.cpp zawierający definicję funkcji pisz_wynik  */

#ifndef PROT.H
#include "PROT.H"
#endif
#include <stdio.h>
void pisz_wynik(double w)
{
  printf("Modul 4  ");    printf("Obliczona suma = %10.2lf\n",w);
}
```

```
/*    plik mod1.cpp zawierający definicję funkcji suma1  */

#ifndef PROT.H
#include "PROT.H"
#endif
#include <stdio.h>

double suma1(double x, double y)
{
   printf("Modul 1\n");  pisz_wynik(x+y);
   return x+y;
}
```

```
/*  plik mod2.cpp zawierający definicję funkcji suma2  */

#ifndef PROT.H
#include "PROT.H"
#endif
#include <stdio.h>

void suma2(double x, double y, double *w)
{
   printf("Modul 2\n");
   pisz_wynik(x+y);
   *w = x+y;
}
```

```
/*  plik mod3.cpp zawierający definicję funkcji suma3  */

#ifndef PROT.H
#include "PROT.H"
#endif
#include <stdio.h>

void suma3(double x, double y, double &w)
{
   printf("Modul 3\n");
   pisz_wynik(x+y);
   w = x+y;
}

/* ----------  Program główny GLOWNY.CPP - wariant 1 ---------- */

#ifndef PROT.H
#include "PROT.H"
#endif

#include "mod1.cpp"   // dołącz kod modułu 1 do programu
#include "mod2.cpp"   // dołącz kod modułu 2
#include "mod3.cpp"   // dołącz kod modułu 3
#include "pisz.cpp"     // dołącz kod modułu 4

#include <stdio.h>
#include <conio.h>
#include <iostream.h>

/*   Wykorzystywane moduły
     mod1.cpp    mod2.cpp   mod3.cpp   pisz.cpp
     oraz plik deklaracyjny     prot.h
*/

void main(void)
{
  double a,b, wynik;

  clrscr();
  cout << "Wprowadz dana 1  ";  cin >> a; cout << endl;
  cout << "Wprowadz dana 2  ";  cin >> b; cout << endl;
  wynik = suma1(a,b);   // przekazanie argumentów przez wartość
  pisz_wynik(wynik);

  suma2(a,b,&wynik);    // przekazanie argumentów przez wskaźnik
  pisz_wynik(wynik);

  suma3(a,b,wynik);      // przekazanie argumentów przez referencje
  pisz_wynik(wynik);

  getch();
 }
```

```
/* ----------  Program główny GLOWNY1.CPP - wariant 2 ---------- */

projekt:  GLOWNY1.PRJ - łączenie kodów wynikowych modułów

// w projekcie pliki *.cpp lub *.obj
// skompiluj, połącz pliki typu obj i utwórz kod wykonywalny

mod1.cpp          // modul 1-  utwórz mod1.obj
mod2.cpp          // modul 2 - utwórz mod2.obj
mod3.cpp          // modul 3 - utwórz mod3.obj
pisz.cpp          // modul 4 - utwórz mod4.obj
glowny1.cpp      // program główny - utwórz glowny1.obj

#ifndef PROT.H
#include "PROT.H"
#endif

#include <stdio.h>
#include <conio.h>
#include <iostream.h>

/*   Wykorzystywane moduły
     mod1.cpp    mod2.cpp   mod3.cpp   pisz.cpp
     oraz plik deklaracyjny     prot.h
*/
void main(void)
{    ...
}
```

Wyniki:

```
Wprowadz dana 1  20

Wprowadz dana 2  40

Modul 1
Modul 4  Obliczona suma =      60.00
Modul 4  Obliczona suma =      60.00
Modul 2
Modul 4  Obliczona suma =      60.00
Modul 4  Obliczona suma =      60.00
Modul 3
Modul 4  Obliczona suma =      60.00
Modul 4  Obliczona suma =      60.00
```

7.7. Dyrektywa #pragma

Format dyrektywy: **#pragma id_dyrektywy** , gdzie

id_dyrektywy musi być jednym z identyfikatorów dopuszczalnych przez kompilator.
Np.

#pragma *argsused* - wyłącza ostrzeżenie informujące o braku odwołań do parametrów funkcji wewnątrz jej ciała.

Parameter <identyfikator> is never used in function <funkcja>

Np.

```
      int pisz(int x, int y) { return x; }
```

Parameter 'y' is never used

#pragma *startup* id_funkcji < priorytet >

#pragma *exit* id_funkcji < priorytet >

Dyrektywy określają funkcje, które są wywoływane przed wywołaniem funkcji main() lub po jej zakończeniu. Funkcja wskazana w dyrektywie ma postać: **void funkcja(void).**

Parametr priorytet jest opcjonalny i ma wartość z przedziału <64,255>. Im mniejsza wartość tym wyższy priorytet. W przypadku rozpoczęcia programu jako pierwsze wywoływane są funkcje o wyższym priorytecie. W przypadku zakończenia programu funkcje o wyższym priorytecie są wywoływane później. Jeżeli priorytet nie został podany, przyjmowana jest wartość 100. W przypadku kilku dyrektyw bez określonego priorytetu funkcje wymienione w późniejszych dyrektywach mają niższy priorytet.

```
void start1(void)
{ cout << "Funkcja start 1\n"; }

void start2(void)
{ cout << "Funkcja start 2\n"; }
            // funkcje wywoływane przed main()

#pragma startup start1  // wyższy priorytet - wywołana jak pierwsza
#pragma startup start2  // niższy priorytet - wywołana jako druga

void koniec1(void)
{ cout << "Funkcja koniec 1\n"; }

void koniec2(void)
{ cout << "Funkcja koniec 2\n"; }
            // funkcje wywoływane po main()

#pragma exit koniec1   // wyższy priorytet - wywołana jako druga
#pragma exit koniec2   // niższy priorytet - wywołana jako pierwsza
```

7.8. Wybrane operacje na jednostkach leksykalnych

Kontynuacja makrodefinicji w następnym wierszu

Jeśli makrodefinicja jest zbyt długa, to można ją kontynuować w nowej linii poprzez umieszczenie na końcu linii znaku \ (backslash).

```
#define NAPIS ”To jest  \
tekst”

printf(NAPIS);
```

Łączenie jednostek leksykalnych za pomocą znaków ##

Istnieje możliwość połączenia dwóch jednostek leksykalnych w jedną za pomocą znaków ##. Preprocesor usuwa znaki ## i łączy jednostki leksykalne.

```
#define NOWY_ID(nazwa1, nazwa2)(nazwa1##nazwa2)

void main()
{
   int NOWY_ID(i,1);                          // zmienna i1
   i1 = 2;  cout << i1 << endl;

   cout << NOWY_ID("To","jest tekst") << endl;
                                              // nowy tekst: ”To jest tekst”
}
```

Konwersja ciągu znaków do łańcucha za pomocą znaku #

Umieszczenie znaku # przed nazwą parametru w ciągu jednostek leksykalnych powoduje przekształcenie tego parametru w łańcuch poprzez dołączenie cudzysłowów na początku i na końcu nazwy parametru.

```
#define LAN(str) #str

printf(LAN(To jest tekst)"\n");   // rozwijane w:  ”To jest tekst”

char * s = LAN(to jest lancuch);   // rozwijane w:  ”to jest lancuch”
printf("%s\n", s);
```