Kurs języka C++

C++ powstał we wczesnych latach osiemdziesiątych. Jego twórcą jest Bjarne Stroustrup z AT&T Bell Laboratories. Język ten powstał jako rozwinięcie języka C. Głównym 'dodatkiem' są tzw. klasy, które czynią C++ językiem obiektowym. Klasa to po prostu wydzielona część kodu, która może zawierać w sobie funkcje i zmienne. Można deklarować egzemplarze klas, uzyskując dostęp do obiektów wykonujących przeróżne zadania, co znacznie upraszcza proces tworzenia programów. W C++ można także (w odróżnieniu od C) deklarować zmienne wewnątrz kodu funkcji, a nie na jej początku. Można także przeciążać funkcje (tzn. tworzyć kilka ich wersji) i operatory (nadawać im nowe znaczenie). Rozwinięciem w stosunku do C jest także nowy sposób dynamicznej alokacji pamięci za pomocą operatora **new**. Jeśli chodzi o komentarze, to dodano możliwość ich tworzenia za pomocą znaków **//**, z tym, że komentarz tego typu dotyczy tylko tekstu w linii, w której występuje.

1. Wprowadzenie do języka C++

1.1. Porównanie struktury programu w Pascalu i w C++

Typowy program w obu językach ma następującą strukturę:

```
Program nazwa;                    int main ()
Var {definicje zmiennych}         {
BEGIN                              /*def. zmiennych
                                         i deklaracje*/
 {instrukcje}                     }
END.
```

Wykonywanie programu zaczyna się od wykonania pierwszej instrukcji z funkcji **main (winmain** – w środowisku graficznym).

Definicje zmiennych mogą występować między instrukcjami, ale zawsze użycie zmiennej musi być poprzedzone jej definicją.

1.2. Podstawowe typy danych

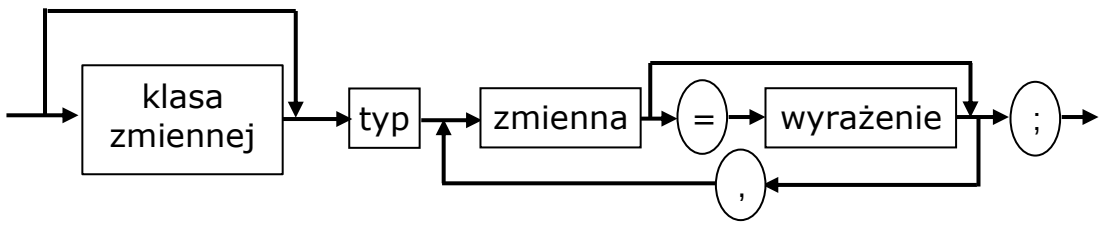
Podstawowe typy danych to:

char – zawiera znaki z tablicy kodu ASCII, stałe są ujmowane w apostrofy ‘a’ , ‘1’;
int – typ całkowity zawiera liczby z <-32768,32767>;
double – typ rzeczywisty;
bool – typ logiczny zawiera wartości true, false.

W przypadku wartości liczbowych w warunkach przyjmuje się wartości true (różny od 0) albo false (0).

1.3. Zmienne

Nazwa zmiennej powinna się zaczynać od litery lub znaku podkreślenia. W nazwie mogą wystąpić jeszcze cyfry. Małe i duże litery są rozróżniane. Zmienna podczas definiowania może zostać zainicjowana, a nawet jest to gorąco zalecane.



Rys. Składnia definiowania zmiennych

Przykłady

```
int liczba=20,k; double x;
char zn;
```

1.4. Operacje arytmetyczne

Dostępne operacje to: +, -, *, /, % (reszta z dzielenia całkowitego).

Przykłady

```
5/3       //1
5%3       //2
5./2       //2.5
```

1.5. Operacje porównania

Dostępne operacje to: <, <=, >, >=, !=, ==.

Przykłady

```
3!= 4    // true
3== 4   // false
```

1.6. Operacje zmiany wartości o 1

Zwiększenie o 1:

```
++ zmienna
lub zmienna ++
```

Zmniejszenie o 1:

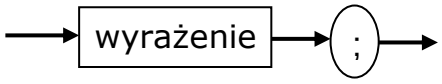
```
-- zmienna
lub zmienna --
```

Przykład

```
k++
```

1.7. Instrukcja wyrażeniowa

Najczęściej instrukcja wykorzystujemy podczas przypisywania wartości zmiennym lub przy wywoływaniu funkcji.



Rys. Składnia instrukcji wyrażeniowej

Przykłady

```
k++;
getch();
```

1.8. Instrukcja złożona

```
{
  definicje zmiennych lub instrukcje
}
```

Instrukcje złożoną stosujemy w sytuacji, kiedy chcemy użyć kilka instrukcji, a składnia dopuszcza tylko jedną instrukcję.

W momencie napotkania definicji zmiennej przydzielone jest jej miejsce w pamięci.

Zmienna istnieje do końca wykonywania instrukcji złożonej.

1.9. Wypisywanie danych do standardowego strumienia wyjściowego

Jeśli chcemy, aby jakieś informacje pojawiły się na ekranie musimy je przesłać do strumienia **cout** – w tym celu korzystamy z operatora wstawiania **<<**.

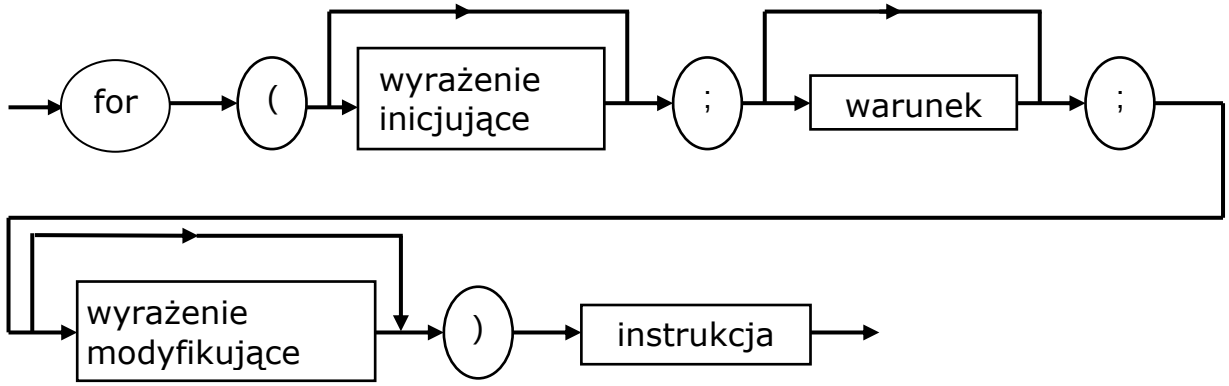
```
cout << wyrażenie;
```

Przykłady

```
cout<<15;
```

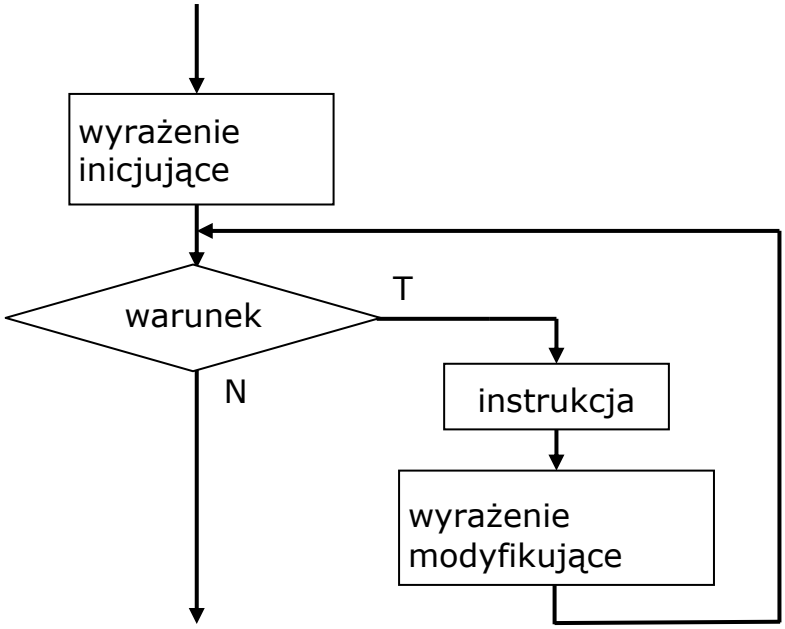
```
cout<<'a';
cout<<"komunikat";
cout<<"a="<<a<<endl;
cout<<'\n';  \\ znak nowego wiersza
```

1.10. Instrukcja for



Rys. Składnia instrukcji for

Wykonanie instrukcji rozpoczyna się od obliczenia wyrażenia inicjującego, jeśli warunek jest spełniony, to zostaje wykonana instrukcja wewnętrzna, potem obliczane jest wyrażenie modyfikujące i następuje powrót do sprawdzenia warunku. Wyjście z pętli następuje, kiedy warunek jest niespełniony.



Rys. Wykonanie instrukcji for

Przykład

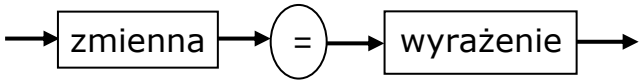
```
cout<<" Kwadraty 20 liczb:";
for (int i=1; i<=20; i++)
     cout<< i*i <<' ';
```

Wyrażenie inicjujące zawiera operacje przypisania wartości początkowych zmiennym używanym w warunku.

Wyrażenie modyfikujące zawiera operacje zmieniające wartości tych zmiennych.

1.11. Operacja przypisania

W wyniku wykonania operacji zmiennej przypisana jest nowa wartość – stanowi ona też wynik operacji.



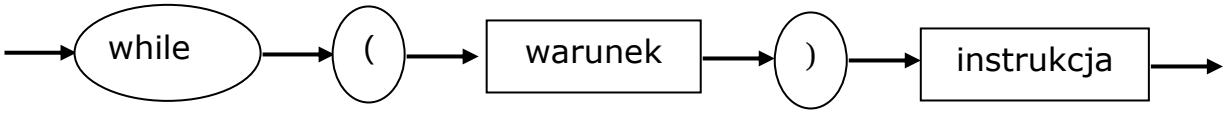
Rys. Składnia operacji przypisania

Przykłady

```
c= (a = 2) * 10;
a = b = 2;
```

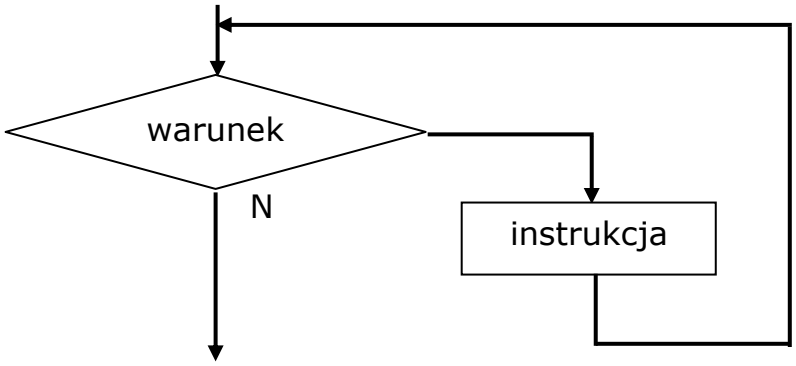
1.12. Instrukcja while

Kiedy nie ma wyrażenia inicjującego i modyfikującego należy sięgnąć po instrukcję iteracyjną while.



Rys. Składnia instrukcji while

Wykonanie instrukcji rozpoczyna się od obliczenia wyrażenia inicjującego, jeśli warunek jest spełniony, to zostaje wykonana instrukcja wewnętrzna, potem obliczane jest wyrażenie modyfikujące i następuje powrót do sprawdzenia warunku. Wyjście z pętli następuje, kiedy warunek jest niespełniony.



Rys. Wykonanie instrukcji while

2. Operacje dla znaków

2.1. Przechowywanie znaków w zmiennych typu char

Dla poszczególnych znaków typu char w zmiennych zapisane są ich numery z tablicy kodów ASCII.

unsigned char – <0,255>
signed char – <–128,127>

Można korzystać ze znaków specjalnych zaczynających się ukośnikiem odwrotnym:

\n – znak nowego wiersza
\r – znak powrotu karetki
\a – sygnał dźwiękowy

\b – backspace
\t – znak tabulacji poziomej
\v – znak tabulacji pionowej
\f – przejście do nowej strony
\0 – znak końca tekstu
\\ – pojedyncza kreska (ukośnik odwrotny)
\' – apostrof
\" – cudzysłów
\0ccc – numer w kodzie ósemkowym
\xhh – numer w kodzie szesnastkowym

Przykłady

```
'A' ≡ '\0101' ≡ '\0x41'
char zn=65;  // zn = 'A'
cout<<zn;    // A
cout<<65;    // 65
cout<< char (65);    // A – operacja jawnej konwersji
cout<<'\n'; //≡ cout<< endl; – przejście do nowego wiersza
cout<<"\n\n\n";
```

2.2. Operacja dla wartości typu char

Operacje porównania

zn1 < zn2 jest prawdą, jeżeli znak z zn1 występuje wcześniej w tablicy kodów ASCII niż z zn2

zn1<= zn2 itd.

Operacje arytmetyczne

zn – n (zn +n) – znak leży o n pozycji przed (za) znakiem z zn

```
zn + 2 //'A'+2;    // 67
cout<<char('A' + 2);   // 'C'
```

Operacje zmiany wartości

zn++ – następny znak z tablicy kodów ASCII

zn-- – poprzedni

zn2 - zn1 – wynikiem jest liczba dzieląca oba znaki

Przykłady

```
'Z' – 'A' //25
cout<<"Wszystkie znaki";
for (int i=0 ; i<=255; i++)
   cout<<char(i);
```

2.3. Funkcje dla konsoli (C++ Builder)

clrscr () – czyszczenie ekranu

gotoxy (x,y) – ustawienie kursora w kolumnie określonej przez pierwszy argument i wierszu określonym przez drugi argument

putch (char zn) – wysyłanie znaku bezpośrednio na konsole

int getch () – pobranie znaku z konsoli bez echa

int getche () – pobranie znaku z konsoli z echem

Przykłady

```
cout<<"Podaj znak:";
char zn =getch();    //znak jest niewidoczny
cout<<"Podałeś :";
putch (zn);
putch ('A');
```

putch ('\n');
putch (65);

Funkcje te mają zdefiniowane prototypy, czyli informacje w typie parametru i typie wyniku w pliku **conio.h**

2.4. Włączanie plików

Składnia języka wymaga, aby przed wywołaniem funkcji wystąpiła jej definicja lub przynajmniej prototyp. Dotyczy to także funkcji standardowych, które są pogrupowane w plikach nagłówkowych o rozszerzeniu **h.** Należy w tym momencie użyć polecenia include dla danego pliku, kiedy chcemy skorzystać z funkcji, której prototyp jest w tym pliku.

#include <plik>
#include "plik"

Jest to polecenie dla procesora, czyli wykonywane przed kompilacją. W miejscu, w którym znajduje się to polecenie wstawiana jest zawartość tego pliku.

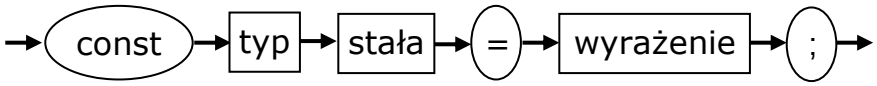
W pierwszym przypadku plik szukany jest w standardowym miejscu. W drugim w katalogu bieżącym.

Przykłady

```
#include <conio.h> //C++ Builder
#include <iostream>
#include "C:\\PROGRAM\\dane.h"
```

2.5. Definiowanie stałych (zmiennych niemodyfikowalnych)

Definicja stałych może wystąpić w dowolnym miejscu przed jej użyciem. Składnia jest taka sama jak składnia definicji zmiennej inicjowanej z dodaniem na początku słowa **const.**



Rys. Składnia definicji stałej

Przykłady

```
const double PI =3.14;
const int MAX=50;
const char koniec ='K',koniec2='k';
```

2.6. Wczytywanie danych ze standardowego strumienia wejściowego

Standardowy strumień wejściowy **cin** zawiera znaki wpisywane z klawiatury. Można je wczytywać za pomocą operatora wydzielania **>>**.

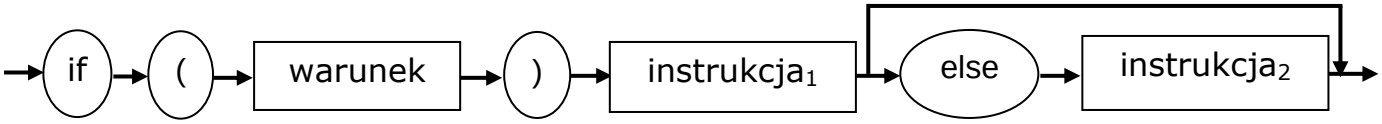
cin>>zmienna;

Przykłady

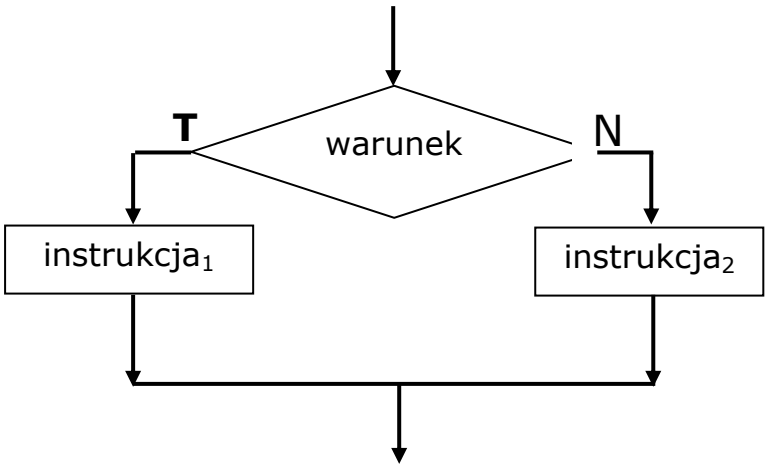
```
cout<<"Podaj znak, liczbę całkowitą, liczbę rzeczywistą:";
char znak; double x; int k;
cin>> znak >> k >> x;
```

2.7. Instrukcja warunkowa

Wykonanie instrukcji **if** zaczyna się od sprawdzenia warunku, jeśli jest spełniony wykonywana jest pierwsza instrukcja, w przeciwnym wypadku wykonywana jest instrukcja druga, pod warunkiem, że występuje.



Rys. Składnia instrukcji warunkowej if



Rys. Wykonanie instrukcji warunkowej if

Przykłady

```
if(x>=0) cout<<"Liczba nieujemna:";
else cout<<"Liczba ujemna:";

if('0'<=zn) //zagnieżdżanie instrukcji if
   if(zn<='9') cout<<"To jest cyfra";
   else cout<<"To nie jest cyfra";
else cout<<"To nie jest cyfra";

if(x>=0)
{
   if(x>0) cout<<"Liczba dodatnia";
}
else cout<<"Liczba ujemna";

if(x>=0)
   if(x>0) cout<<"Liczba dodatnia";
   else:   // x ≡ 0
else cout<<"Liczba ujemna";
```

2.8. Operacje logiczne

&& – koniunkcja
|| – alternatywa
! – negacja

Przykłady

```
if(!('0'<=zn && zn<='9')) cout<<"To nie jest   cyfra";
else cout<<"To jest cyfra";
if(zn == ' ' || zn == '\t'  ||zn == '|n ' )
cout << "To jest znak niewidoczny":
```

Koniunkcja ma wyższy priorytet niż operacja alternatywy.

Przykład

```
if (‘a’ <= zn && zn<= ‘z’ || ‘A’ <= zn && zn <= ‘Z’)
cout << „To jest litera”:
```

Operacje logiczne służą do zapisu złożonych warunków.

2.9. Prawa de Morgana

Te prawa stosujemy do zamiany instrukcji wewnętrznych w instrukcji warunkowej (negowanie warunku)

~ (a lub b) ≡ ~ a i ~ b

~ (a i b) ≡ ~a lub ~ b

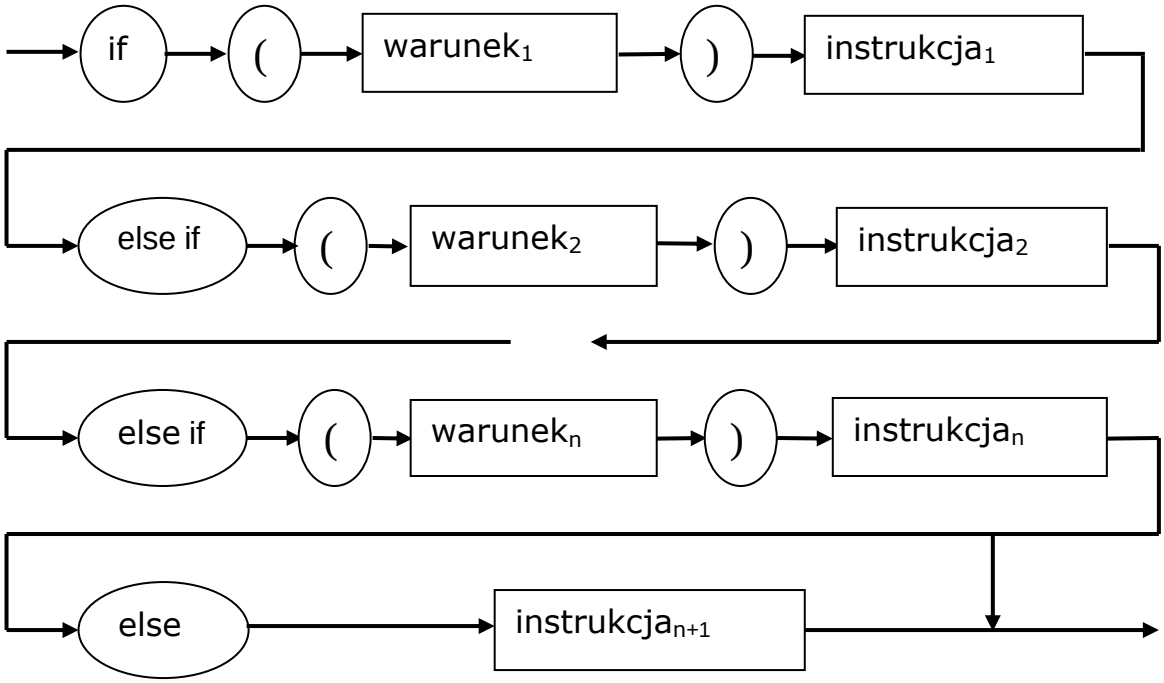
Przykłady

```
if (! (‘0‘ <= zn && zn <= ‘9’))
   cout << “To nie jest cyfra”;
else cout<<”To jest cyfra”;
   równoważne
if (‘0’>zn || zn >’9’) cout << “To nie jest cyfra”;
else cout<<”To jest cyfra”;

if(! (zn == ‘0‘ || zn == ‘ \ t’ || zn== ‘\n’)) równoważne
if(!(zn == ‘0‘) && !(zn == ‘\t) && !(zn == ‘\n’)) równoważne
if(zn != ‘0‘ && zn != ‘\t && zn != ‘\n’))
```

2.10. Konstrukcja else if

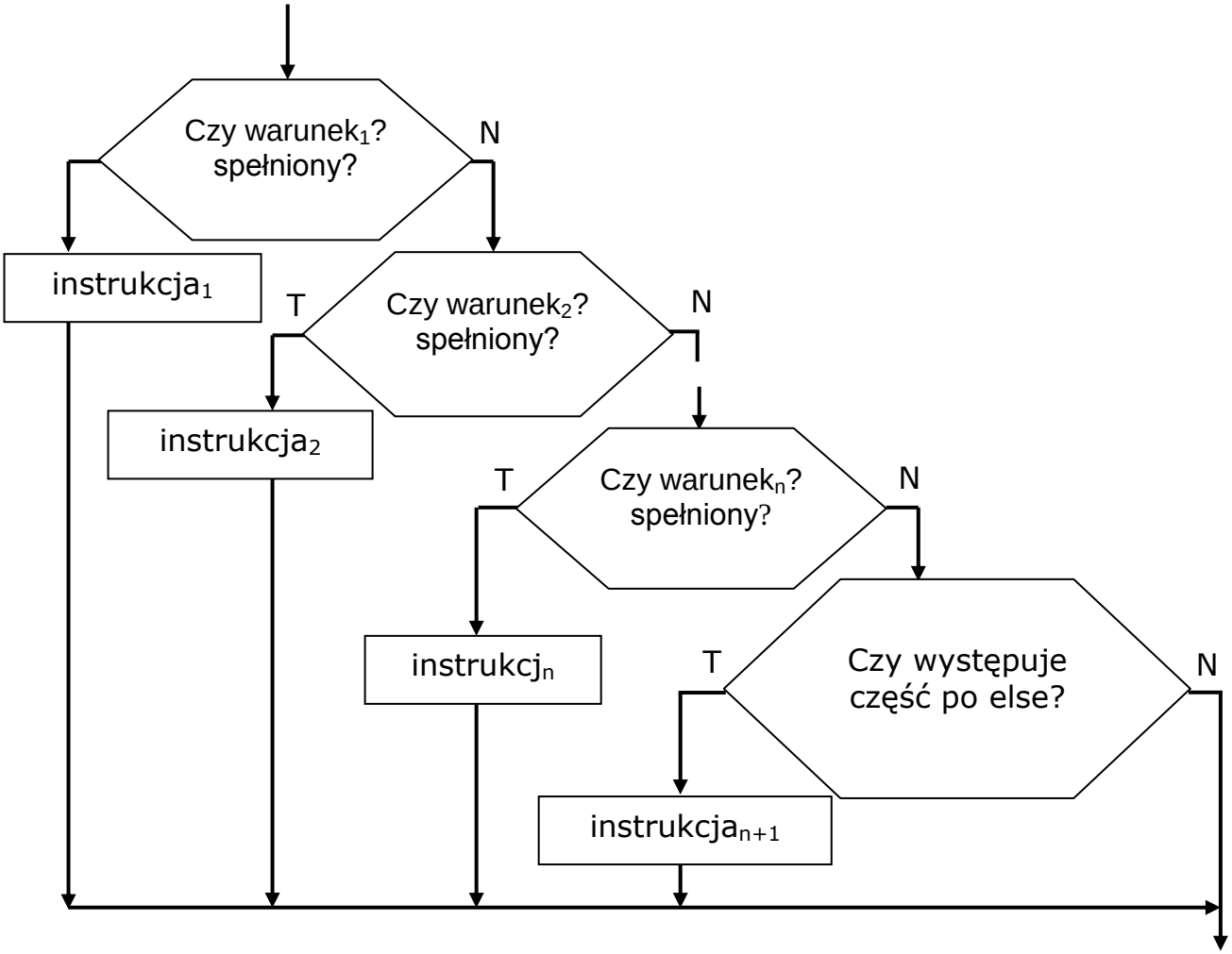
Konstrukcja else if służy do zapisu decyzji wielowariantowych



Rys. Składnia konstrukcji else if

Przykład

```
if (‘0’<=zn &&  zn <=’9’) cout << “To jest cyfra”;
else if (zn ==’ ‘ || zn == ‘\t’  || zn == ‘\n’)
   cout<< “To jest znak niewidoczny”;
else if (zn ==’ ; ‘ || zn ==‘,’ || zn ==‘.’ ||zn ==’.’)
   cout<< “To jest znak interpunkcyjny”;
else cout<< “To jest inny znak”;
```



Rys. Wykonanie konstrukcji else if

2.11. Klasyfikacja znaków

Przy sprawdzaniu zawartości zmiennej znakowej w typowej sytuacji można skorzystać z gotowych funkcji:

int **isalnum** (char zn) – zwraca 1 jeśli w zn jest litera lub cyfra, w przeciwnym wypadku zwraca 0
int **isalpha** (char zn) – sprawdza czy znak jest literą;
int **isdigit** (char zn) – sprawdza czy znak jest cyfrą;
int **isupper**(char zn) – sprawdza czy znak jest dużą litera;
int **islower**(char zn) – sprawdza czy znak jest mała litera;
int **isgraph**(char zn) – sprawdza czy jest to znak widoczny;
char **tolower** (char zn) – zamienia dużą literę na małą literę, a pozostałe znaki pozostawia bez zmian;
char **toupper** (char zn) zamienia małą literę na dużą, a pozostałe znaki pozostawia bez zmian;

Nagłówki tych funkcji są w pliku **ctype.h**.

Przykład

```
if ( isdigit (zn)) cout<<"To jest cyfra";
```

2.12. Zmienne licznikowe

Zmienne licznikowe służą do zliczania ilości wystąpień w określonym badanym zjawisku. Zmienne licznikowe na ogół są typów całkowitych. Na początku należy je zainicjować, potem często mają modyfikowane wartości za

pomocą operacji zwiększenia wartości o 1. Na końcu występuje wykorzystanie wartości, które ostatecznie przyjęła zmienna licznikowa.

Przykłady

```
int  l_ml= 0, l_dl= 0 , char  zn;
cout << ”Podaj tekst”;
while ( cin>> zn )
   if ( isupper (zn))  l_dl ++ ;
   else if (islower (zn))  l_ml ++;
cout<< ”Dużych liter było:” << l_dl << endl;
cout<<”Małych liter było:”<< l_ml << end << endl;
```

2.13. Wczytanie znaku za pomocą funkcji get

Funkcja get jest funkcją przeciążoną, czyli można ją wywoływać korzystając z różnych form:

1) *int get( )*

2) *ifstream & get (char &zn)*

W wyniku wywołania funkcji zostaje wczytany jeden znak ze strumienia **cin**. Znak ten może być znakiem niewidocznym.

Przy wczytywaniu z pomocą operatora wydzielania znaki niewidoczne są pomijane.

Przykłady

```
int zn1; zn1 = get( );       // użycie pierwszej formy
int zn2; get ( zn2 ); //użycie drugiej formy
while ((zn = get( ))!=’\n’)
   //instrukcje
```

W przypadku użycia pierwszej formy, jeśli użytkownik naciśnie (Ctrl Z) jako wartość funkcji będzie zwrócona wartość stałej EOF.

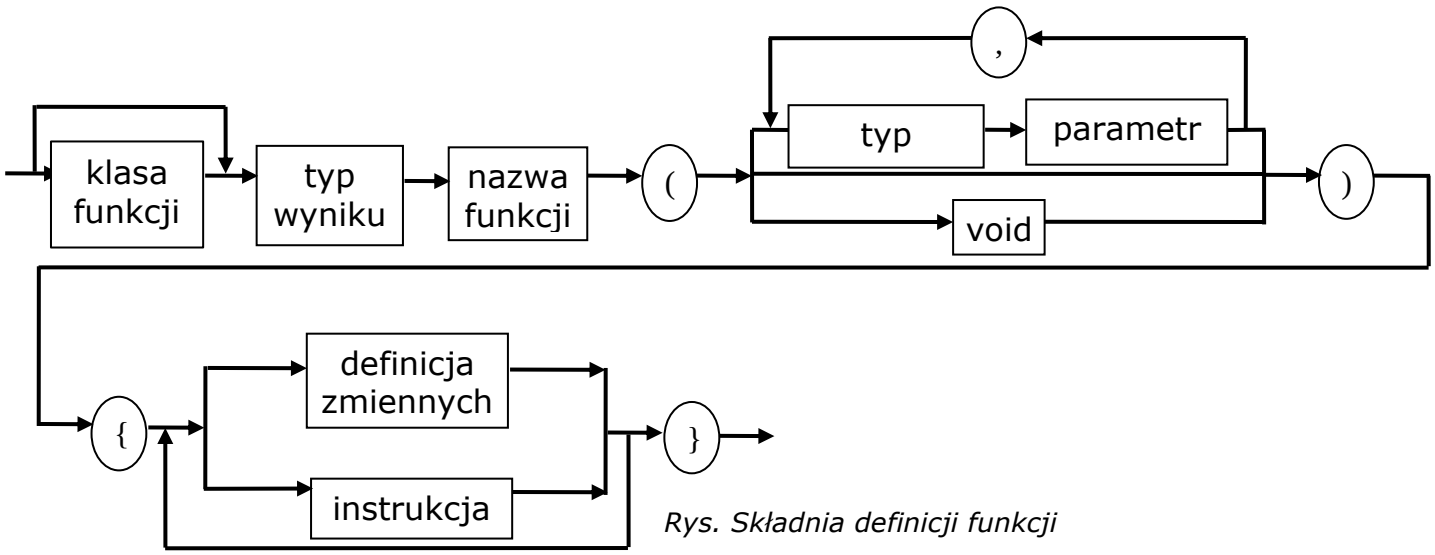
Przykład

```
while (( tolower (zn=get( ))!=’k’)
   //instrukcje
```

3. Funkcje i tablice

3.1. Składnia definicji funkcji

Funkcje w **C++** są odpowiednikami funkcji i procedur z Pascala. Jeśli funkcja nie zwraca wyniku, to jako typ wyniku piszemy **void** (*pusty*). Instrukcja złożona stanowi treść funkcji – w niej mogą znajdować się obok instrukcji, definicje zmiennych lokalnych z tej funkcji.

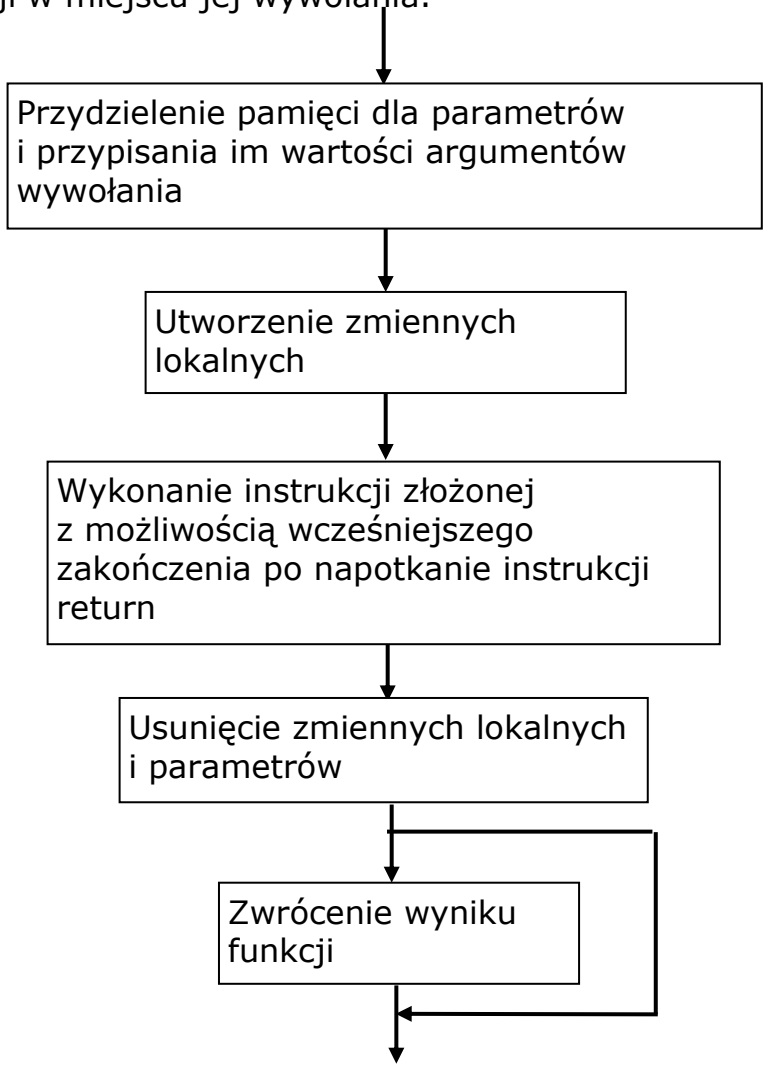


Rys. Składnia definicji funkcji

Prototyp jest to nagłówek zakończony średnikiem. Kiedy nie wystąpiła jeszcze definicja funkcji, a chcemy ją wykorzystać powinien być umieszczony jej prototyp przed wywołaniem. Definicja funkcji nie może zawierać definicji innej funkcji.

3.2. Wykonanie funkcji

Wykonanie funkcji rozpoczyna się od przydzielenia jej miejsca w pamięci dla parametrów (na stosie), następnie obliczane są wartości argumentów (parametrów faktycznych) i przypisywane są do parametrów (formalnych). Potem wykonywana jest instrukcja złożona stanowiąca treść funkcji, czyli tworzone są zmienne lokalne występujące
w tej funkcji i wykonywane są instrukcje wewnętrzne. Wykonanie może się wcześniej zakończyć w momencie napotkania instrukcji **return**. Na końcu wykonania usuwane są zmienne lokalne, parametry i ewentualnie zwracany jest wynik funkcji w miejscu jej wywołania.



Rys. Wykonanie funkcji

Przykład

```
void litery( ):
void main( )
{
  litery( );
 }
```

```
void litery( )
{  //wyświetla tylko litery
   int zn;
   while ((zn = get( ))! = EOF)
      if (isalpha(zn))  cout<<zn;
}
```

3.3. Instrukcja return

Postacie:

1) ***return****;*
2) ***return*** *wyrażenie;*

W wyniku wykonania instrukcji w pierwszej postaci kończy się wykonanie funkcji.

Wykonanie instrukcji w drugiej postaci zaczyna się od wyliczenia wartości wyrażenia, zakończenia wykonywania funkcji i zwrócenia wartości jako wyniku funkcji w miejscu jej wywołania.

Pierwsza postać instrukcji return stosowana jest w funkcjach nie zwracających wyniku. Druga postać stosowana jest w funkcjach zwracających wynik.

Przykład

```
int ile_liter ( )
{
   while ((zn = get( ))! = EOF);
   if (isalpha(zn)) ile++;
   return ile;
}
…
int n;
n= ile_liter();
cout<< ”Liter było:”<< n;
```

3.4. Przekazywanie parametrów przez wartość

Parametry przekazywane są do funkcji przez wartość, czyli funkcja otrzymuje jedynie wartości parametrów faktycznych, a nie może modyfikować znajdujących się w tych parametrach wartości. Parametry służą do komunikacji pomiędzy funkcją a otoczeniem, w którym jest ona wywoływana. Przekazywanie kilku wartości na zewnątrz jest możliwe dzięki parametrom będącym zmiennymi wskaźnikowymi lub referencyjnymi.

Przykład

```
void zamiana ( int a, int b)
{ //przykład nieskutecznej zamiany
   int c;
   c = a;
   a = b;
   b = c;
//return
}
```

Przykład

```
int suma_a_b (int a, int b)
{ /*funkcja sumuje wartości całkowite
   z przedziału <a,b>*/
  int s = 0;
  for (int l=a; l<=b; l++)
  s=s+l;
  return s;
}
```

Dzięki parametrom można zapisywać funkcje w ogólniejszych postaciach.

3.5. Tablice

Tablica jest to ciąg znaków tego samego typu.

Składnia definicji

```
typ_elementu nazwa_tablicy [rozmiar]
```

Przykłady

```
int liczba [10],i, ab[15], j;
double pomiar [100];
```

Numeracja elementu tablicy zaczyna się od zera.

Do elementów tablicy można odwoływać się w sposób tradycyjny za pomocą operacji indeksowania.

Składnia operacji

```
tablica [nr_elementu]
```

Przykłady

```
cout<<pomiar[j]<<"- j- ty pomiar";
for (int i = 0; i < n; i++)
  cin>>ab[i];
```

3.6. Tablice znakowe

Tablice znakowe służą do przechowywania tekstu.

Stała tekstowa jest stałą tablicową zakończoną za pomocą znaku końca tekstu.

Przykłady

```
"tekst"
't' 'e' 'k' 's' 't' '\0' – znak końca tekstu
char wiersz [81], nazwisko [20], imię [15];
```

Tekst w tablicach znakowych musi być zakończony znakiem o kodzie 0.

3.7. Inicjowanie tablic

Wartości początkowe przypisywane elementom tablicy muszą być ujęte w nawiasy klamrowe.

Przykłady

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},n;
double trójkąt [3] = {2.5, 4.1, 3.2}; //długości
  boków trójkąta
```

W liście wartości początkowych można pominąć wartości końcowe albo na odwrót rozmiar.

Przykłady

```
int b[ ] = {1, 2, 3, 4, 5, 6, 7}; // rozmiar = 7
char miasto[20] = "Biłgoraj";
```

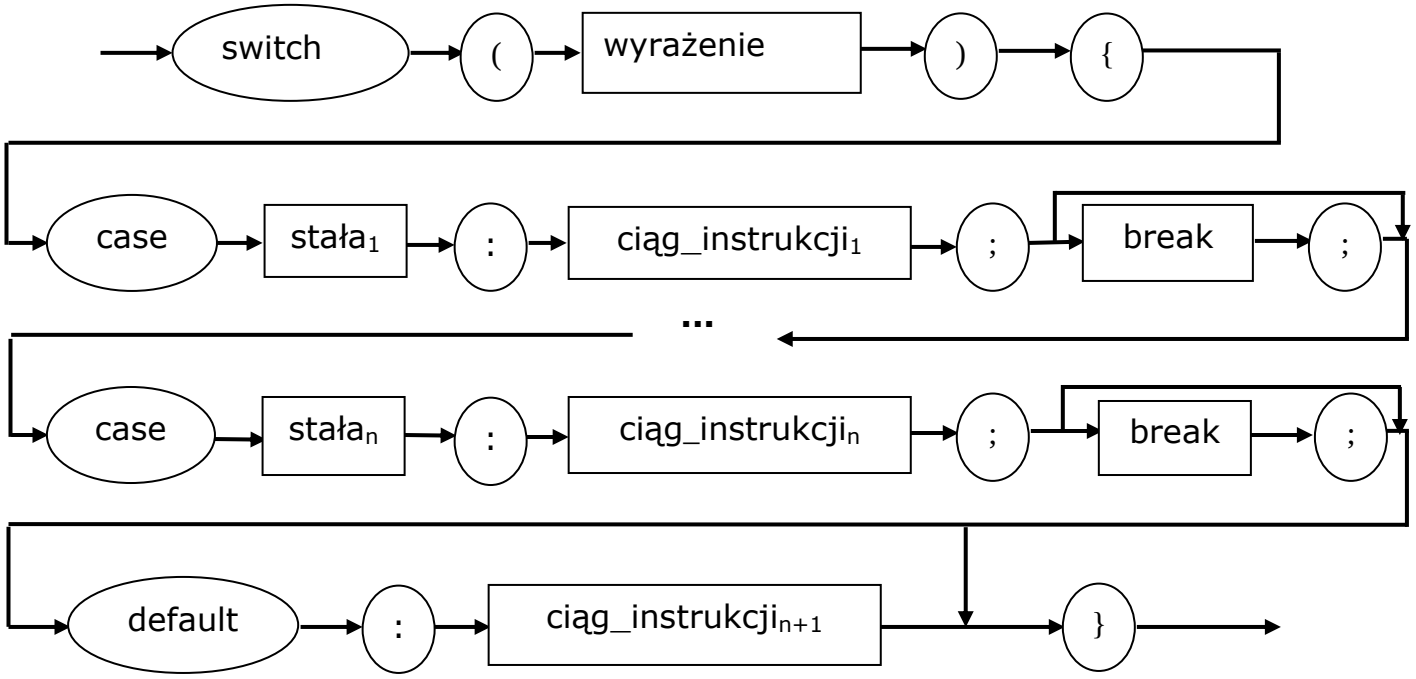
3.8. Wczytywanie i wyświetlanie tekstu dla tablic

Wczytując tekst do tablicy należy pamiętać o możliwym przepełnieniu tablicy. Do tablicy wczytywany spójny widoczny ciąg znaków.

Przykłady

```
cout<<"Podaj nazwisko";
cin>>nazwisko;
cout<<"Podaj wiersz";
cin>>get (wiersz, 80);
cin>>getline (wiersz, 80);  // wczytywanie, ciągu znaków nie
   dłuższego niż 80 znaków
```

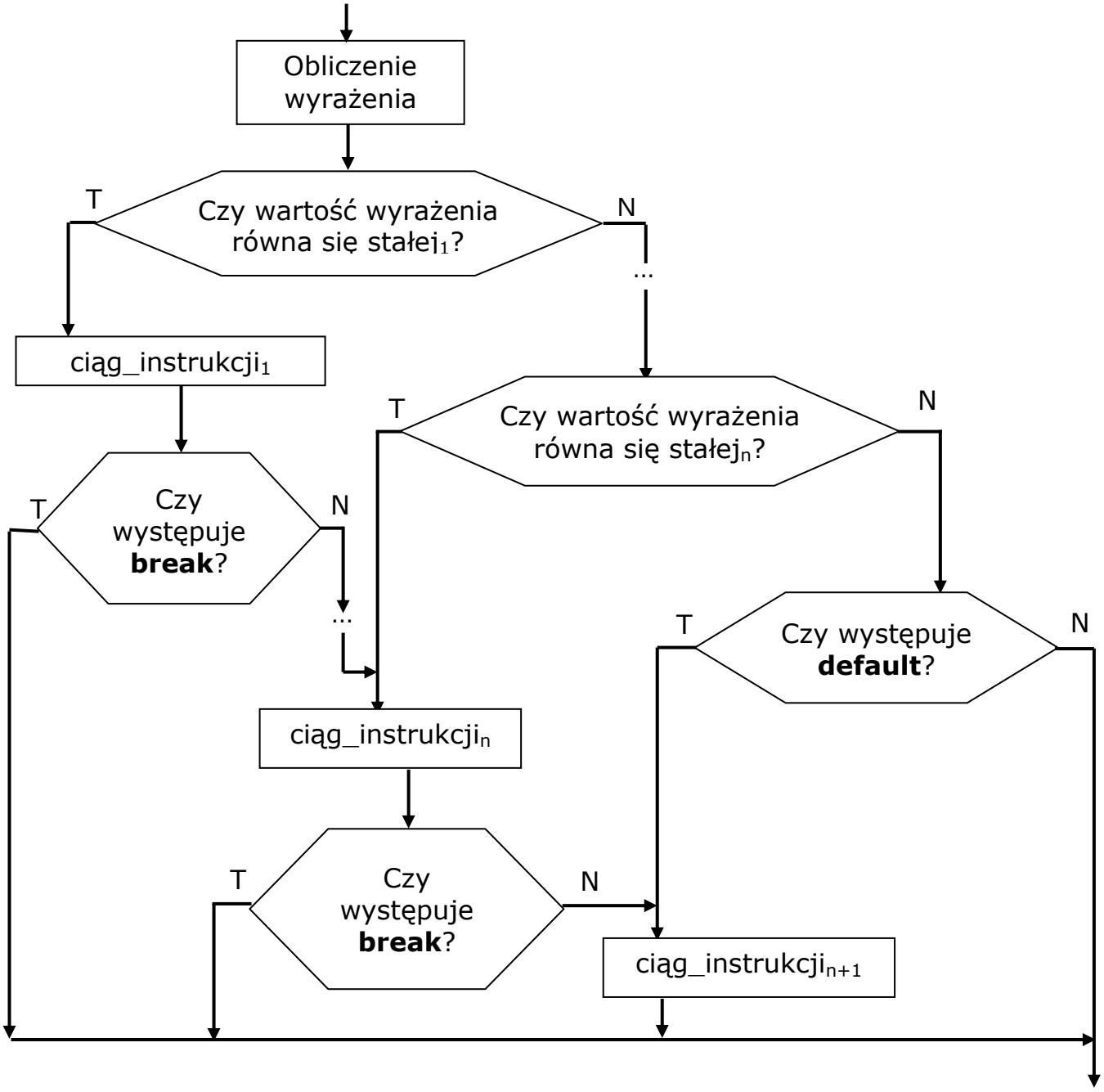
3.9. Instrukcja wyboru



Rys. Składnia instrukcji wyboru switch

Wykonanie instrukcji wyboru zaczyna się od obliczenia wartości wyrażenia, następnie ta wartość porównywana jest z wartością pierwszej stałej, jeśli jest jej równa, to wykonywany jest ciąg instrukcji znajdujący się za nią do momentu napotkania instrukcji **break** albo końca instrukcji wyboru, w przeciwnym wypadku wartość wyrażenia będzie porównywany z kolejną stałą. W momencie wystąpienia równości będzie wykonywany ciąg instrukcji za nią się znajdujący do napotkania instrukcji break albo końca instrukcji wyboru, itd.

Jeśli wartość wyrażenia nie była równa żadnej stałej, to jeśli występuje przypadek domyślny **default**, to wykonywany jest ciąg instrukcji znajdujący się po słowie default do momentu napotkania instrukcji break albo końca instrukcji wyboru.



Rys. Wykonanie instrukcji wyboru switch

3.10. Zastosowanie instrukcji wyboru

Instrukcje wyboru stosujemy do zapisania decyzji wielowariantowej, gdy wybór wariantu zależy od przyjęcia przez wyrażenie konkretnej wartości liczbowej lub znakowej.

Przykład

```
void main( )
{ //program zlicza ilość cyfr, znaków niewidocznych, pozostałych
   znaków
  int  cyfry[10] = {0,0,0,0,0,0,0,0,0,0}, niewid=0;
  int pozost=0, zn; clrscr( );
  while ((zn=cin.get( ))!= EOF)
    switch (zn)
```

```
    {
      case ’0’: case ’1’: case ’2’: case ’3’: case ’4’:
      case ’5’: case ’6’: case ’7’: case ’8’: case ’9’:
        cyfry [zn – ‘0’]++; break;
      case ‘ ‘: case ’\t’: case ’\n’:
      case ’\a’: niewid++;break;
      default: pozost++;
    }
  for (int i = 0; i < 10; i++)
    cout<<”Cyfra ”<< i <” wystąpiła ”<<cyfry[i]<<
      ” raz(y)” << endl;
  cout<<”Znaki niewidoczne wystąpiły ” << niewid << ” raz(y)” <<
   endl;
  cout<< ”Pozostałe znaki wystąpiły”<< pozost <<
      ”raz(y)” <<endl;
  getch ( );
}
```

3.11. Porównywanie tekstu

Funkcja **strcmp** porównuje teksty znajdujące się w dwóch tablicach.

int strcmp (char tekst1[ ], char tekst2 [ ])

Jeżeli tekst pierwszy jest wcześniejszy niż tekst drugi zwracana jest wartość większa od zera, jeśli są identyczne zwracane jest zero, jeśli tekst pierwszy jest późniejszy niż tekst drugi, to zwracana jest wartość większa od zera.

Przykład

```
char operacja [20];
if (strcmp (operacja, ”odejmowanie”)==0)
{
  cout<<”Podaj odjemną i odjemnik”; ...
```

3.12. Kopiowanie tekstu

W wyniku wykonywania funkcji tekst z drugiej tablicy skopiowany jest do tablicy pierwszej.

strcpy (char dokad[ ], char skad[ ])

Przykład

```
char osoba[20];
strcpy (osoba,”Kowalski”);
```

3.13. Dołączanie tekstu

W wyniku wykonywania funkcji do tekstu znajdującego się w tablicy pierwszej będzie dołączony tekst z tablicy drugiej.

strcat (char tekst[ ], char dodatek[ ])

Przykład

```
strcat (osoba, ” Jan”);
```

3.14. Operacje zmiany wartości o jeden

Jeśli operacja zmiany wartości o jeden występuje w wyrażeniu w postaci przedrostkowej, to najpierw wykonywana jest zmiana wartości zmiennej, a potem użycie nowej wartości w wyrażeniu. Jeśli natomiast operacja zmiany wartości o jeden występuje w postaci przyrostkowej, to najpierw w wyrażeniu użyta jest stara wartość zmiennej, a potem wykonywana jest zmiana wartości o jeden.

Przykłady

```
                    n = 5;
1) k = ++n ; // k = 6      2) k = n++; // k= 5
                    m = 0
l = m-- ;  // l = 0          l = --m;   //l = -1
n = k + m; // n=6      n = k + m; // n=4
```

3.15. Instrukcja do while

Wykonanie pętli rozpoczyna się od wykonania instrukcji wewnętrznej następnie sprawdzony jest warunek, jeśli jest spełniony następuje powrót do wykonywania instrukcji. Wyjście z pętli następuje przy nie spełnionym warunku.

Przykład

```
do // wypisanie kolejnych cyfr liczby rząd od tyłu
{
  cout<< char (n %10 + '0');
  n = n /10;
} while (n !=0);
```

3.16. Wykorzystanie poszczególnych pętli

Pętle for stosujemy, kiedy występuje wyrażenie inicjujące i wyrażenie modyfikujące. W ten sposób w nagłówku pętli umieszczone są wszystkie operacje mające wpływ na wartość zmiennych używanych w warunku, co przyczynia się do czytelności instrukcji iteracyjnej.

Pętle do while stosujemy, kiedy nasza pętla ma obrócić się przynajmniej jeden raz – w pozostałych przypadkach stosujemy pętle while**.**

3.17. Zastosowanie instrukcji break w pętlach

Wykonanie instrukcji break w dowolnej pętli powoduje zakończenie jej wykonywania.

Przykład

```
int liczba;
for (int i = 0; i < 100; i++)
{ //przerwanie wczytywania po napotkaniu liczby
  ujemnej
  cout<<"Podaj liczbe:";
  cin>>liczba:
  if (liczba < 0) break;
  cout<<"podałeś: "<<liczba;
}
```

4.Typy liczbowe

4.1. Typy całkowite

Liczby zapisane w typie, **int** interpretowane są w kodzie uzupełniającym U2. Natomiast w przypadku typu **unsigned** w kodzie dwójkowym wartość zapisana jest na dwóch bajtach. Wpływa to na zakresy wartości:

int – < -32768, 32767 >, unsigned – < 0, 65535 >.

Zmienne typu unsigned mogą być wykorzystywane jako zmienne licznikowe. Stałe tego typu mają na końcu literkę **U**. Na wartościach tego typu dopuszczalne jest wykonywanie poznanych operacji arytmetycznych pod warunkiem, że wynik należy do zakresu wartości tego typu.

Wartość typu **long** zapisana jest na czterech bajtach,
w ten sposób uzyskujemy przedział wartości <21478483648, 21478483647>.
Stałe tego typu mają dodawaną na końcu literę **l**. Można wykonywać na wartościach tego typu te same operacje, co dla typu **int**

Typ **long unsigned** oferuje wartości nieujemne zapisane na czterech bajtach, czyli od 0 do ok. 4 mld. Stałe tego typu mają na końcu dodane litery **lu.** Dla wartości **long unsigned** wyniki operacji arytmetycznych są poprawne pod warunkiem, że należą do przedziału wartości tego typu.

Przykład

```
Program wypisujący max ilość wyrazów ciągu Fibonacci'ego
long unsigned f0 = 0, f1 =1,fn;
cout<<"f0 ="<<f0; cout<<"\nf1 ="<<f1;
for (int i = 2; i <1000; i++)
{
  fn = f0 + f1;
  cout <<endl; cout<<'f'<<i<<" ="<<fn;
  if (fn < 200000lu)
  {
    f0=f1; f1=fn;
  }
  else break;
}
```

4.2. Układ szesnastkowy

Przejście z zapisu szesnastkowego do zapisy w układzie dziesiętnym wymaga uwzględnienia wag poszczególnych cyfr zapisu.

$$S_n\, S_{n-1} \ldots S_1\, S_0 = S^n * 16_n + S_{n-1} * S^{n-1} + \ldots + S_1 * 16^1 + S_0 * 16^0$$

Przykład

$$1fa_{(16)} = 256+15*16+10= 256+240+10=506$$

Liczby w zapisie szesnastkowym w programie C++ poprzedzamy parą znaków **0x**.

Przykład

```
int a = 0x1fa;  //  a = 506
```

4.3. Operacje logiczne na bitach

W C++ można skorzystać z następujących operacji: negacja bitów (**~**),koniunkcja (**&**),alternatywa (**|**),
alternatywa wykluczająca (**^**).

Wykonanie operacji realizowane jest na poszczególnych bitach czyli w przypadku operacji dwuargumentowych brane są odpowiadające sobie pary bitów z argumentów.

Przykłady

```
char a = 0XEC;      // 1 1 1 0 1 1 0 1
char c = ~a;        // 0 0 0 1 0 0 1 0
char b = 0X4F;      // 0 1 0 0 1 1 1 1
char d = a &b;      // 0 1 0 0 1 1 0 1
char e = a | b;     // 1 1 1 0 1 1 1 1
char f = a ^ b;     // 1 0 1 0 0 0 1 0
```

Bitowe operacje logiczne wykorzystywane są przy zapisywaniu informacji na ciągu bitów krótszych niż jeden bajt – kiedy zależy nam na oszczędności miejsca w pamięci.

4.4. Przesunięcie bitów

a << *n* w lewo o n bitów
a >> *n* w prawo o n bitów

Przy przesunięciu bitów o n pozycji w lewo z prawej strony wchodzą zera.

Przy przesunięciu bitów w prawo, jeśli wartości są typów bezznakowych to z lewej strony wchodzą zera w przypadku wartości ze znakiem powielany jest najstarszy bit.

4.5. Typ wyliczeniowy

Typ wyliczeniowy pozwala nam na definiowanie nowych wartości.

enum nazwa_typu {nazwa1 [=nr1], nazwa2 [=nr2],...};

Przykład

```
enum dzien {poniedzialek, wtorek, sroda,
  czwartek, piatek, sobota, niedziela}
```

Typ wyliczeniowy służy do zwiększania czytelności programu poprzez użycie identyfikatorów identycznych jak w rzeczywistości.

Przykłady

```
dzien d = wtorek;
if ( d == sobota || d == niedziela)
  cout<<"Dzień wolny";
```

4.6. Nowe operacje przypisania

Nowe operacje to: +=, -=, *=, /=, %=, &=, |&, ^|, <<=, >>=.

Interpretowane są w sposób następujący:

zm. op. = wyr. ⇔ zm. = zm. op. (wyr.)

Przykłady

```
i = c + 2;
x = x * 3;
i + = 2;       // i = i + 2;
x * = 3;       // x = x * 3;
y * = z + 1; // y = y *(z + 1);
```

Nowe operacje przypisania pozwalają skrócić zapis instrukcji, a jednocześnie odpowiadają sposobowi myślenia programisty.

4.7. Operacja warunkowa

Operacja warunkowa w pewnych sytuacjach może zastępować instrukcję warunkową.

warunek? wyrażenie1: wyrażenie2

Wykonanie operacji zaczyna się od sprawdzenia warunku, jeśli jest spełniony obliczane jest wyrażenie1 jego wartość jest wynikiem operacji, w przeciwnym wypadku obliczane jest wyrażenie2 i jego wartość jest wynikiem operacji.

Przykłady

```
double wart_bez_x= x > = 0? x : -x;
max = a > b  ? a : b;
//wypisanie liczb z tablicy po 10 w wierszu
for (int i = 0; i < n; i++)
  cout<< a [ i ]<< char (( i+1) % 10==0? '\n':'  ');
```

4.8. Konwersje jawne

Konwersja czyli zmiana typu wartości.

1) *typ (wyrażenie)*
2) *(typ) wyrażenie*

Po obliczeniu wartości wyrażenia, jest ona przekształcona na taką samą lub zbliżoną wartość w typie docelowym. Nie zawsze taka wartość istnieje. Jeżeli taka sytuacja wystąpi, programista nie będzie ostrzeżony o błędnym wyniku konwersji.

Konwersje jawną stosujemy w celu wykonania konwersji na określony typ lub też w celu zwiększenia czytelności programu.

```
int iloraz_calk (double x, double y)
{
  return int ( x/y);
}
```

4.9. Konwersje wśród typów całkowitych

Kierunki konwersji:

int → long int
↕ ↕
unsigned → long unsigned

Przykłady

```
long(12)   // 12 l
long unsigned (60000 u)  //  60000 lu
int (10 l)  //  10
int (1000000 l)  //  ?
```

Wynik konwersji wśród typ całkowitych jest dokładny pod warunkiem, że wartości przekształcone należą do typu docelowego.

4.10. Konwersje wśród typów rzeczywistych

Kierunki konwersji:

float ↔ double ↔ long double

Przykłady

```
double (0.1234567 f)  //  0.1234567
float (0.123456789) //  0.12345678
```

Konwersje wśród typów rzeczywistych przebiegają poprawnie pod warunkiem, że wartość przekształcona należy do typu docelowego. W takiej sytuacji czasami dodatkowo może być ona zaokrąglana.

4.11. Konwersja między typem całkowitym a rzeczywistym

Kierunki konwersji:

typ całkowity ↔ typ rzeczywisty

Przykłady

```
double (12)  //  12.0
```

Podczas konwersji liczby całkowitej na rzeczywistą dodawana jest część ułamkowa równa 0. Natomiast przy konwersji liczby rzeczywistej na całkowitą, obcinana jest część ułamkowa bez wykonania zaokrąglenia.

W przypadku wartości nie mieszczących się w typie całkowitym otrzymujemy niepoprawny wynik konwersji.

4.12. Przyczyny konwersji niejawnej

Konwersje niejawne przeprowadzane są
z następujących powodów:

1) w wyrażeniach występują wartości różnych typów,

2) typ zmiennej z lewej strony operatora przypisania jest inny niż typ wartości wyrażenia stojącego z prawej strony,

3) typy parametrów faktycznych różnią się od typów parametrów formalnych,

4) typy wartości wyrażenia w instrukcji return jest inny niż typ wyniku funkcji.

Podczas obliczania wyrażenia najpierw wszystkie wartości typu char przekształcane są na typ int, a wartości typu float na typ double. Potem, gdy dla jakiejś operacji jej argumenty są różnych typów, to konwersje wykonywane są w kierunku obszerniejszego typu (long double $\supseteq$ double $\supseteq$ long unsigned $\supseteq$ long $\supseteq$ unsigned).

Przykład

```
long l, char zn, int i;
l = (zn + i ) * l;
l = (int (zn) + i ) * l;
l = long (int (zn) + i) * l;
```

4.13. Wykonywanie obliczeń na liczbach rzeczywistych z podwójną precyzją

Przykład

```
float f1, f2, f;
f = f1 + f2;
f = double (f1) + double (f2);
f = float(double (f1) + double (f2));
```

Jak wynika z powyższego przykładu obliczenia na liczbach rzeczywistych wykonywane są przynajmniej
w arytmetyce z podwójną precyzją czyli w obrębie typu double. Gwarantuje to minimalizację pojawiających się błędów przy prowadzeniu obliczeń na liczbach rzeczywistych w komputerze.

4.14. Operacje występujące w języku C++

Operacja	Łączność
( ) [ ] . ->	l
+- ++ -- & * (typ) sizeof ! ~	p
* / %	l
+ -	l
<< >>	l
< <= > >=	l
!= ==	l
&	l
^	l
|	l
&&	l
||	l
?:	p
= += -= *= /= itd.	p
,	l

4.15. Wpływ priorytetu na kolejność wykonywania operacji

Podczas obliczania wyrażenia najpierw są wykonywane operacje zmiany wartości o 1 w postaci przedrostkowej potem o kolejności wykonywania operacji decyduje priorytet, czyli operacje o priorytecie wyższym wykonywana jest wcześniej na końcu wykonywane są operacje zmiany wartości o 1 w postaci przyrostkowej.

Przykład

$a =_4 b --_5 +_3 c *_2 ++_1 d$

4.16. Wpływ łączności na kolejność wykonywania operacji o takim samym priorytecie

Jeśli operacje mają taki sam priorytet o kolejności ich wykonania decyduje łączność. W przypadku operacji lewostronnie łącznych znajdujących się na tym samym poziomie struktury nawiasowej najpierw są wykonywane operacje z lewej strony.

W przypadku operacji prawostronnie łącznych są one wykonywane od prawej strony.

Można zauważyć, że priorytety najpopularniejszych operacji przedstawiają się następująco:

arytmetyczne > porównania > logiczne > przypisania.

4.17. Obliczanie argumentów operacji

Jeśli po obliczeniu jednego argumentu znany jest już wynik operacji logicznej, to drugi warunek już nie jest obliczany. W przypadku obliczania argumentu dla wielu operacji kolejność jest nieustalona poza operacją

warunkową i operacją wywołania funkcji, gdzie parametry faktyczne wyliczane są począwszy od prawej strony.

Przykład

```
void fun (int a, int b)
.....
i = 5;
fun (i, i ++);  // a = 6, b = 5
a = ile_biale (tekst) + usun_biale(tekst);
if (ujemne (t1, m) != 0 || ujemne (t2, n))
   cout <<"W tablicy są liczby ujemne";
```

W pierwszym przypadku nie jest pewna kolejność wywołania funkcji, a w drugim czy dojdzie do drugiego wywołania.

Rozwiązania

```
a = ile_biale(tekst);
a+= usun_biale(tekst);
i = ujemne(t1, m); j = ujemne (t2, n);
if (i != 0 || j != 0)
   cout <<"W tablicy są liczby ujemne";
```

5.Wskaźniki zmiennych i dostępne dla nich operacje

5.1. Zmienne wskaźnikowe

Nazwy typów wskaźnikowych składają się z nazwy typu, którego zmienne mają być wskazywane oraz ***.**

Przykłady

```
int *, double *, char *
```

Wartości typu wskaźnikowego są wskaźnikami, czyli adresami innych zmiennych.

Przykłady

```
int *w1, *w2, a;
double x, *wx, y;
```

Zmienna wskaźnikowa zajmuje 2 bajty.

5.2. Operacja adresu

Pobranie adresu jest możliwe dzięki operacji adresu

& zmienna

Wynikiem operacji wskaźnik do danej zmiennej.

Przykłady

```
w1 = &a;
wx = &x;
```

5.3. Operacja wyłuskania

Mając adres zmiennej możemy modyfikować jej wartość nie korzystając z jej nazwy.

** zmienna*

Wynikiem tej operacji jest zmienna wskazywana przez podany wskaźnik.

Przykłady

```
*w1 = 5;
*wx = y;
cout <<"x= "<< *wx;
cin >> *w1;  //  >>a
int n, *wn = &n;
```

5.4. Przekazywanie kilku wartości na zewnątrz przy pomocy wskaźników

Jeśli parametr funkcji ma służyć do przekazywania wartości na zewnątrz, musi być on zmienną wskaźnikową inicjowaną na początku adresem zmiennej, w której będzie umieszczona ta wartość.

Przykłady

```
void zamiana (int *wa, int *wb)
{ //skuteczna zamiana
  int rob = *wa;
  *wa = *wb;
  *wb = rob;
}

void suma_iloczyn(double x, double y, double
*wsuma, double *wiloczyn)
{
  *wsuma = x + y;
  *wiloczyn = x * y;
}
```

5.5. Wskaźniki elementów tablicy

Zakładając, że **wa** wskazuje nam jakiś element tablicy, to **(wa + k)** oznacza wskaźnik elementu, który leży o k pozycji za wskazywanym przez wa;

***(wa – k)** oznacza wskaźnik elementu, który leży o k pozycji przed elementem wskazywanym przez wa;

***(wa + k)** oznacza element, który leży o k pozycji za elementem wskazywanym przez wa**.**

***(wa - k)** oznacza element, który leży o k pozycji przed elementem wskazywanym przez wa**.**

Do elementów tablicy w C++ można odwoływać się na dwa sposoby:

1) za pomocą operacji indeksowania,

2) korzystając ze wskaźników i operacji wyłuskania.

5.6. Ścisły związek między tablicami a wskaźnikami w C++

Nazwa tablicy oznacza w wyrażeniach adres zerowego elementu

Wskaźnik może się pojawić w miejscu nazwy tablicy w operacji indeksowania.

Przykłady

```
wa = a; //niedopuszczalny zapis odwrotny
a [i] ≡ *(a + i)
*(wa +i) ≡ wa [i]
```

5.7. Przekazywanie tablic do funkcji

Jeśli tablica jest przekazywana do funkcji, to tak naprawdę parametr jest zmienną wskaźnikową, w której zostaje umieszczony adres zerowego elementu.

Podczas kompilacji funkcji odwołanie do elementu tablicy z użyciem operacji indeksowania zastępowane są odwołaniami do elementów tablicy z użyciem operacji wyłuskania i wskaźników.

Przykład

```
void czytaj (int a [ ], int n)
{  //          int *a
   cout << ”Podaj liczby”;
   for (int i = 0; i < n; i++)
      cin >> a [i];  // cin >> *(a + i);
}
...
int liczby[100];
czytaj (liczby,100);
//       &liczby[0]
```

W funkcjach w C++ można modyfikować elementy tablic – jest to możliwe dzięki przekazaniu do funkcji adresu zerowego elementu tablicy.

5.8. Tablica tablic

Elementami tablicy mogą być inne tablice. Takie tablice mogą posłużyć do przechowywania macierzy itd. Do elementów tablicy tablic odwołujemy się za pomocą operacji indeksowania.

Przykłady

```
int tab[4][5];
int i,j,tabpom[5][7],k;
tab[0][3]=23; k=tab[0][4];
cout << tab[i][j];
```

Zależności

```
tab=&tab[0], int (*)[10]
tab[i]=&tab[i][0]
```

Jeśli tablica tablic przekazana jest do funkcji, to może ona je modyfikować. Postać nagłówka takiej funkcji wygląda następująco

```
fun (int a[][10], int m, int n)
```

5.9. Inicjowanie tablicy tablic

Inicjując tablicę tablic umieszczamy dane początkowe dla tablic wewnętrznych w nawiasach klamrowych

Przykład

```
int ai[4][3]={{1, 2, 3}, {4, 5, 6}, {7, 8, 9},
   {10, 11, 12};
```

Nawiasy wewnętrzne, o ile nie prowadzi to do sprzeczności, można pominąć.

Przykłady

```
int ai[4][3]={{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
                          12};
int a2[4][3]={{1},{2},{3},{4}};
```

5.10. Stałe tekstowe a wskaźniki

Stała tekstowa traktowana jest jako stała tablicowa, czyli w miejscu jej wystąpienia wstawiany jest wskaźnik do jej początku.

Przykłady

```
char *wt = ”luty”;
strcmp (char text1[], char text2[]);
strcmp (char *text1, char *text2);
if (strcmp (miesiąc, ”luty”)==0)
```

5.11. Tablice wskaźników

Ciąg wskaźników przechowujemy w tablicach liczbowych.

typ_wskaźnikowy nazwa_tab [rozmiar];

Przykłady

```
char *wskmiesiące [12];
double *wliczby [30];
```

Najczęściej stosuje się tablice wskaźników do wskazywania tekstu i rekordów nazywanych strukturami.

Przykłady

```
wskmiesiące [0] = ”styczeń”;
double x;
wliczby [0] =&x;
cout<< ”x= ”<<*wliczby [0];
```

5.12. Inicjowanie tablicy wskaźników

Przykłady

```
char *wskdni [7] ={”niedziela”,”poniedziałek”,
”wtorek”,”środa”, czwartek”,”piątek”,”sobota”};
cout<< ”Dni robocze”;
for(int i =1; i < 6; i ++)
  cout<< wskdni [i]<< ‘ ‘;
```

5.13. Porównanie tablicy tablic i tablicy wskaźników

Kiedy piszemy programy dotyczący tekstu teoretycznie możemy skorzystać zarówno z tablicy tablic jak też z tablicy wskaźników jednak dokładnie przyglądając się obu typom można wskazać następujące zalety przemawiające za tablicą wskaźników:

1) tablice wewnętrzne mogą być różnych długości,
2) tablice wewnętrzne można w łatwy sposób sortować.

6. Struktury

6.1. Definicja typu strukturowego

Struktura składa się ze składowych niekoniecznie tego samego typu.

```
struct nazwa_typu
{
  //definicja składowych
};
```

Przykłady

```
struct data
{
  int dzień, miesiąc, rok;
};
struct punkt
{  double x, y; };
```

6.2. Definicja zmiennej strukturowej

Strukturę definiuje się tak jak zmienną prostego typu.

Przykłady

```
data  dzisiaj; punkt p, q;
```

6.3. Zagnieżdżanie struktur

Składowe mogą być strukturami. Pozwala to na oddanie wewnętrznej ”struktury” obiektów z rzeczywistości.

Przykłady

```
struct okrąg
{
  punkt środek;
  double r;
};
struct osoba
{
  char nazwisko[15], imie[12];
  data data_ur;
  int wzrost;
};
```

6.4. Operacja składowej

Do składowych dostajemy się za pomocą operacji składowej.

struktura . składowa

Wynikiem operacji jest składowa z podanej struktury.

Przykłady

```
okrąg ok; osoba os;
dzisiaj.dzień=2; dzisiaj.miesiąc=4; dzisiaj.rok=2004;
p.x =0; p.y =0;
q.x =1.2; q.y =2.5;
cout<< ”Podaj nazwisko:”;
cin>> os.nazwisko;
cout<<”Podaj imię:”;
cin>> os.imie;
cout<<” Podaj datę urodzenia dd mm rr:”;
cin>> os.data_ur.dzien >> os.data_ur.miesiąc >> os.data_ur.rok;
cout<< ”Dane o okręgu”<<endl;
cout<< ”Środek = (”<<ok.srodek.x <<’,’<<ok.srodek.y<<’)’;
cout<< ”Promień = ”<<ok.r;
```

Struktura składa się z elementów, które nie muszą być tego samego typu.

6.5. Wskaźniki struktur

Wskaźniki struktur są wykorzystywane przy przekazywaniu struktur do funkcji (jako parametry).

Przykłady

```
punkt p, *wp = &p;
data d, *wd = &d;
osoba os, *wosb =&os;
okrąg ok, *wok =&ok.;
void czytaj (osoba *wos);
```

6.6. Operacja wyłuskania składowej

Mając wskaźnik struktury, kiedy chcemy się odwołać do składowej zamiast operacji wyłuskania możemy zastosować operację wyłuskania składowej.

Przykłady

```
(*wp).x = 1.5;
(*wp).y = 2.2;
wskaźnik struktury -> składowa;
wp -> x =1.5;
wp -> y =2.2;
```

Wynikiem operacji jest podana składowa ze struktury wskazywanej przez pierwszy argument operacji.

Przykład

```
void czytaj (osoba *wos)
{
  cout<< ”Podaj nazwisko:”;
  cin>> wos -> nazwisko;
  cout<< ”Podaj imię:”;
  cin>> wos -> imie;
  cout<< ”Podaj datę urodzenia dd mm rr:”;
  cin>> wos -> data_ur.dzien >>
   wos -> data_ur.miesiąc >>  wos -> data_ur.rok;
  cout<< ”Podaj wzrost”;
  cin>> wos -> wzrost;
}
void main ( )
{
  osoba os;
  .............
  czytaj  (& os);
}
```

6.7. Tablice struktur

Tablica struktur pozwala na umieszczenie w jednym miejscu zbioru obiektów.

Przykłady

```
punkt  wielokąt [7];
osoba  firma [20];
wielokąt [0].x =0.0;
wielokąt [0].y =0.0;
firma [5].data_ur.rok =1980;
```

6.8. Inicjowanie struktur

Inicjowanie struktury jest podobne do inicjowania tablicy. Przy zagnieżdżeniu stosujemy poznane rekurencyjnie.

Przykłady

```
punkt np.= {2.1, 2.7};
data urodziny ={3, 10, 1980};
okrąg ok2 ={{2.3, 2.5} 3.14};
osoba klient ={„Kowalski”,”Julian”,{1,6,1957} 178};
```

6.9. Inicjowanie tablicy struktur

Podczas inicjowania tablicy struktur stosujemy poznane zasady inicjowania tablicy i struktury jednocześnie.

Kiedy nie prowadzi to do błędu, można opuszczać wewnętrzne nawiasy.

Przykłady

```
punkt wielokąt[7] = {{0,0}, {1,2.3}, {3,6}, {4,2.4}, {-3,-6},
   {-10,-12}, {0,0}};
// lub = {0,0,1,2.3,3,6,4,2.4,-3,-6,-10,-12,0,0};
```

6.10. Zmienne niemodyfikowalne

Jeśli definicja zmiennej poprzedzona jest słowem **const**, to oznacza, że wartość takiej zmiennej nie może być zmieniona.

Przykład

```
const int k =5;
```

6.11. Zmienne referencyjne

Zmienna referencyjna jest synonimem zmiennej, która została podana w miejscu jej definicji, czyli znajduje się w tym samym miejscu w pamięci co oryginalna zmienna.

typ_podstawowy &zmienna=zmienna_typu_podst.

Przykłady

```
int k;
int &rk = k;
double x, &x = x;
k = 5;    cout<< rk; // 5
rk = 3; cout<< k; // 3
```

6.12. Zmienne referencyjne w roli parametrów

Zmienne referencyjne zaleca się stosować przy przekazywaniu struktur do funkcji, gdy wartość składowych ma być nie modyfikowana. Stosujemy wtedy operację składowej przy dostępie do składowych w strukturze.

Przykłady

```
void wyswietl (const osoba &os1)
{
  cout<<os1.nazwisko<<' '<<os1.imie<<endl;
  cout<<"ur."<<os1.data_ur.dzien<<'.'
  os1.data_ur.miesiac << '.' << os1.data_ur.rok;
}
void main ( )
{
  osoba os;
        ...
  wyswietl (os);
}
```

6.13. Operator new

Zmienne dynamiczne tworzymy za pomocą operacji **new**.

1) ***new*** *typ*
2) ***new*** *typ [rozmiar]*

W wyniku wykonania operacji utworzona zostaje zmienna podanego typu i zwrócony zostaje wskaźnik do tej zmiennej.

Kiedy jest użyta druga forma, to utworzona zostaje tablica dynamiczna o podanym rozmiarze i zostaje zwrócony wskaźnik na początek tej tablicy. Wskaźnik pozwala na dostęp do zmiennej, gdyż nie ma ona nazwy.

Przykłady

```
double * wx;
int * wi, *w10;
osoba *wos = new osoba;
wi = new int;
wx = new double;
w10 = new int [10];
```

6.14. Operator delete

Zmienne dynamiczne istnieją do momentu usunięcia za pomocą operacji **delete**.

1) ***delete*** *wskaźnik*
2) ***delete*** *[ ] wskaźnik*

Przykłady

```
delete wi;
delete [ ] w10;
```

6.15. Funkcje rekurencyjne

Funkcja jest funkcją rekurencyjną, jeśli wywołuje samą siebie bezpośrednio lub pośrednio (za pomocą innej funkcji)

Przykład

```
0! = 1;
n! = n(n-1)!
long unsigned silnia (int n)
{
  if (n== 0) return 1;
  else return n * silnia (n-1);
}
```

6.16. Tablice dynamiczne

Stosowanie tablic dynamicznych w programach pozwala nam na oszczędne wykorzystanie pamięci – dokładnie tyle ile jest nam potrzebne.

Przykład

```
void main ( )
{
  int n;
  cout<<"Ile będzie liczb"; cin>>n;
  double *tab_dyn =new double [n];
  for (int i = 0; i < n; i++)
  {
    cout<<"Podaj liczbe:"; cin >>tab_dyn[i];
  }
       ...
  delete [ ] tab_dyn;
}
```

6.17. Tablice wskaźników do struktur

Jeśli chcemy sortować tablice struktur, stosowanie tablicy struktur jest wręcz nie do przyjęcia. Optymalnym rozwiązaniem w takiej sytuacji jest tablica wskaźników struktur.

Przykład

```
osoba *tab_wsk [50];
tab_wsk[0] = new osoba ;
```

6.18. Listy

Szybkie wstawianie i usuwanie nowych elementów jest możliwe w liście (jedno lub dwukierunkowej). Elementy są strukturami, w których jedna składowa jest wskaźnikiem do następnego elementu. W programie należy zdefiniować wskaźnik do początku listy.

Przykład

```
struct osoba_w;
{
  //składowe informacyjne
  osoba *nast; //wskaźnik do kolejnego elementu
} *pocz;
```

6.19. Drzewa binarne

Jeśli musimy sortować zmienne dynamiczne, to oszczędne algorytmy związane są z drzewami binarnymi.

Drzewo jest wskazywane za pomocą wskaźnika zwanego korzeniem.

```
struct węzeł
{
  // składowe informacyjne
  węzeł *lewe, *prawe; //składowe wskaźnikowe
} *korzeń =0;
```

Do obsługi drzew binarnych stosuje się funkcje rekurencyjne.

7. Klasy

7.1. Dzielenie programu na moduły i klasy

Podczas pisanie dużego programu przez kilka osób można podzielić go na kilka modułów. Każdy moduł może być samodzielnie kompilowany. Zawiera funkcje logicznie ze sobą powiązane. Podczas tworzenia modułów można korzystać z różnych klas zmiennych i funkcji.

Klasa zmiennej jest jej atrybutem (obok nazwy, typu, adresu itd.) decydującym o czasie istnienia, położeniu w pamięci i zasięgu. W języku C++ zmienne mogą być klasy **auto**, **register**, **static** i **extern**.

Klasa funkcji decyduje o jej zasięgu. Funkcje mogą być klasy **static** i **extern**.

7.2. Zmienne klasy auto

Zmienne klasy auto mogą być parametrami lub zmiennymi lokalnymi funkcji. Umieszczane są na stosie. Istnieją podczas wykonywania funkcji i są poprzez nazwę tylko w niej dostępne. Ewentualne inicjowanie występuje przy każdym wywołaniu funkcji. Podanie nazwy klasy nie jest obowiązkowe.

Przykład

```
auto int n=20;
```

7.3. Zmienne klasy register

Zmienne klasy register uzyskają miejsce w szybkich rejestrach procesora. Jest to jedyna różnica w stosunku do zmiennych klasy auto.

Przykład

```
double sumakwad(double liczby[], int n)
{
  int i; double s=0; register double *wl=liczby;
  for (i=0;i<n;i++)
  {
    s+=*wl * *wl; wl++;
  }
  return s;
}
```

Można optymalizując czas działania funkcji zmienną często używane zdefiniować jako zmienną klasy register.

Nie zawsze przydział miejsca w rejestrze jest możliwy ze względu na ograniczoną ilość takich rejestrów.

7.4. Zmienne klasy static wewnętrzne

Zmienne klasy static wewnętrzne mogą być zmiennymi lokalnymi funkcji i są poprzez nazwę tylko w niej dostępne. Umieszczane są na stercie. Istnieją od pierwszego wywołania funkcji do końca wykonywania programu (czyli także między wywołaniami funkcji). Stanowią pamięć funkcji. Inicjowanie występuje przy pierwszym wywołaniu funkcji.

Przykład

```
static int k=0;
```

7.5. Zmienne klasy static zewnętrzne

Zmienne klasy static zewnętrzne definiowane są na zewnątrz funkcji. Umieszczane są na stercie. Istnieją przez cały czas działania programu. Stanowią pamięć wspólną funkcji modułu czyli są dostępne poprzez nazwę tylko w module, w którym wystąpiła ich definicja. Inicjowanie podaną wartością występuje przed wykonaniem pierwszej instrukcji w programie.

Można stosować te zmienne:

1) gdy używając funkcji, wykorzystującej takie zmienne nie musimy o nich wiedzieć,

2) gdy skraca to postać wywołań funkcji bez zmniejszenia czytelności.

7.6. Zmienne klasy extern

Zmienne tej klasy definiujemy w jednym module na zewnątrz funkcji, a w pozostałych, w których chcemy z nich korzystać, podajemy deklarację zaczynającą się słowem extern. Te zmienne stanowią pamięć wspólną kilku modułów programu. Pozostałe cechy zmienne mają identyczne ze zmiennymi klasy static zewnętrzne. Łatwy dostęp do tych zmiennych w programie może być przyczyną trudnych do wykrycia błędów.

7.7. Funkcje klasy static

Nagłówek zaczyna się słowem static. Można z tych funkcji korzystać tylko w module, w którym wystąpiła ich definicja. Są to funkcje pomocnicze w module.

Przykład

```
static int fpom( );
```

7.8. Funkcje klasy extern

Można z tych funkcji korzystać w module, w którym wystąpiła ich definicja oraz w modułach, w których wystąpił ich prototyp. W tym ostatnim przypadku prototyp zaczyna się słowem extern. Są to funkcje główne w module.

Przykład

```
int fpodst ( );
```

7.9. Funkcje przeciążone

Jeśli funkcja posiada kilka definicji z różnymi sekwencjami typów parametrów, to taką funkcję nazywa się przeciążoną. W zależności od typów argumentów wywołania kompilator wybiera wersję, która ma być użyta.

Przykład

```
void wyświetlaj (int k);
void wyswietlaj (double x);
void wyswietlaj (const char tekst[ ])
```

7.10. Pojęcie klasy jako nowego typu w C++

Klasa jako typ zmiennych składa się ze składowych, które mogą być zarówno zmiennymi (pola), jak też funkcjami (metody). Domyślnie w klasie umieszcza się metody wykorzystujące pola czyli w jednym miejscu mamy zebrane dane i działające na nich funkcje. Przyczynia się to większej czytelności programu.

```
class  nazwa_klasy
{
  public:
    // definicja składowych
  private:
    // definicje składowych
};
```

7.11. Porównanie sekcji public i private

Składowe umieszczane w sekcji **private** mogą być wykorzystywane tylko w metodach danej klasy. Są to funkcje pomocnicze oraz pola chronione przed dostępem z zewnątrz.

Sekcja **public** zawiera metody udostępniane na zewnątrz klasy, stanowiące jej interfejs. Nie poleca się umieszczać w niej pól, gdyż w ten sposób trudno będzie wykryć związane z nimi błędy.

Definicja klasy może jeszcze zawierać sekcję **protected**.

7.12. Definiowanie metod

Na ogół w definicji klasy umieszcza się jedynie prototyp funkcji, natomiast jej pełna definicja występuje poniżej. Od definicji zwykłej funkcji nagłówek różni się poprzedzeniem nazwy funkcji nazwą klasy i operatorem zakresu.

Przykład

```
void  Stos :: push ( int _element)
```

7.13. Obiekty

Obiekty są to zmienne jakiejś klasy.

Przykład

```
Stos s;
```

Obiekty mogą parametrami i wynikiem funkcji.

W zależności od potrzeb tworzone obiekty mogą być obiektami poznanych klas auto, static lub extern (ale nie register) lub obiektami dynamicznymi.

7.14. Odwoływanie się do metod i pól

Wewnątrz klasy z jej składowych korzystamy za pośrednictwem ich nazw. Na zewnątrz do składowych sekcji public odwołujemy się za pomocą operacji składowej po podaniu obiektu.

Przykład

```
s. push (10)
```

7.15. Funkcja wstawiania

Nagłówek funkcji wstawianej w definicji zaczyna się słowem **inline**.

Przykład

```
inline int f( )
{
   ...
}
```

Jest to komunikat dla kompilatora, aby w miejscu wywołania funkcji wstawił jej treść. Jeśli funkcja jest dostatecznie krótka, to kompilator tak zrobi.

Ponadto jeśli w definicji klasy zamiast prototypu wystąpi definicja metody, to kompilator domyślnie przyjmie, że chodzi o funkcję wstawianą.

7.16. Wskaźniki obiektów

Wskaźniki obiektów są użyteczne przez przekazywaniu obiektów do funkcji, podobnie jak to było w przypadku struktur. Mając wskaźnik obiektu do składowych będziemy się dostawać za pomocą operacji wyłuskania składowej.

Przykład

```
Stos st;
st.push (5);
Stos *ws1= &st, *ws2;
  …
ws1 -> init( );
ws1 -> push(10);
ws2 = &st;ws2 -> push(15);
ws1 -> print ( );  //  15  10
```

7.17. Tablice obiektów

W pewnych sytuacjach chcemy korzystać ze zbioru obiektów – użyteczne są wtedy tablice obiektów.

Przykład

```
Stos st5[5];
for (int i = 0; i< 5; i++)
   st5 [i].init( );
     …
for (i = 0; i < 5; i++)
{  st5 [i].push (i); st5 [i].push (i * i); }
```

7.18. Obiekty dynamiczne

Obiekty dynamiczne tworzymy i usuwamy jak zwykłe zmienne za pomocą operatorów new i delete.

Przykłady

```
Stos *wdst;
wdst = new Stos;
wdst -> init ( );
wdst -> push (100); wdst -> push (200);
wdst -> print ( );
delete wdst;
```

7.19. Tablice dynamiczne

Połączeniem tablic i obiektów dynamicznych są tablice dynamiczne tworzone w specjalny sposób za pomocą new.

Przykłady

```
Stos * wdtst5;
wdst5 =new Stos[5];
for (int i = 0; i< 5; i++)
{
  wdtst5[3].init ( );
  wdtst5[3].push (100*i);
  wdtst5[3].push (200*i);
  wdtst5[3].print ( );
}
delete [ ] wdtst5;
```

7.20. Struktura programu z klasami

Zaleca się umieszczanie definicji klasy w osobnym module. W plikach o rozszerzeniu h umieszczamy polecenia włączenia innych plików, definicję stałych, typów, klasy oraz metod wstawianych.

Natomiast w pliku o rozszerzeniu cpp umieszczamy definicje poszczególnych metod.

Taka struktura pozwala na łatwe dodawanie nowych metod rozszerzających możliwości klasy.

Zakres metod ogranicza się do danej klasy, czyli jeśli w innej klasie występuje metoda o takiej samej nazwie, to nie oznacza, że ta metoda jest funkcją przeciążoną.

8. Konstruktory i destruktory

8.1. Wartości domyślne parametrów

Jeśli wartości niektórych parametrów często otrzymują tę samą wartość, to można podać ją w nagłówku funkcji.

W wywołaniu ją pomijamy chyba, że chcemy przekazać inną wartość do funkcji.

Tak postępujemy, zaczynając od prawej strony nagłówka. Składnia języka nie dopuszcza pominięcia danego parametru, kiedy z jego prawej strony jest parametr nie pominięty.

Przykłady

```
void petla (int poczatek=1, int koniec=10, int rok=1);
void main ( )
{
  petla (40, -40, -5);
  petla (-10, 20); // petla (-10, 20, 1);
  petla (-20);      //  petla (-20, 10, 1);
   petla ();          //  petla (1, 10, 1);
}
```

8.2. Konstruktor obiektu

Konstruktor obiektu posiada taką samą nazwę jak nazwa klasy. Nie ma określonego wyniku (nawet typu void).

Jego zadaniem jest nadanie wartości wszystkim polom danego obiektu. Może posiadać parametry, którym wartości przekazujemy w miejscu definicji obiektu.

Wywoływany jest automatycznie w momencie tworzenia obiektu. Powinien być zdefiniowany w każdej klasie.

Przykład

```
class Stos
{
  public:
    Stos ( );     // konstruktor bezparametrowy
    Stos ( int); //konstruktor z jednym parametrem
    Stos (int, int); //konstruktor z dwoma par.
    ...
}
inline Stos :: Stos(int _element)
{
  tablica [0] = _element;
  szczyt = 1; stan = NIEPELNY;
}
void main( )
{
  Stos st(10);
  cout << "Stos:" << st.print ( );
}
```

Kiedy w klasie występują tylko konstruktory z parametrami, nie można zdefiniować obiektu nie podając wartości dla parametrów tego konstruktora. Wyjściem jest napisanie definicji konstruktora z wartościami domyślnymi.

8.3. Destruktor

Nazwa destruktora składa się z nazwy klasy poprzedzonej znakiem tyldy (~). Nie ma podanego wyniku oraz nigdy nie posiada parametrów. Jest wywoływany w momencie, gdy dany obiekt przestaje istnieć. Jego zadaniem jest zwolnienie pamięci dynamicznej wskazywanej przez pola obiektu, do której, w przeciwnym przypadku, stracilibyśmy dostęp.

Przykład

```
inline String :: ~String;
{
  delete [ ] str;
}
```

8.4. Pola const i lista inicjująca konstruktora

W klasie mogą wystąpić definicje pól **const**. Nie mogą one być modyfikowane – nawet za pomocą instrukcji konstruktora czyli w żadnej metodzie.

W tym celu inicjowanie pól, zwłaszcza const należy skorzystać z listy inicjującej konstruktora.

```
Konstruktor (parametry):pole_1(wyr_1),pole_2(wyr_2)
    {
      // instrukcje
    }
```

Przykład

```
class Liczby
{
  public:
    Liczby::Liczby (int _cn, double _x);
  private:
    const int cn;
    int n;
    float x;
};
Liczby::Liczby (int _cn, double _x):
     cn(_cn), x(_x)   //inicjowanie pól
{
  n = 100;
}
```

8.5. Obiekty i funkcje const

Jeśli zdefiniujemy jakiś obiekt jako zmienną niemodyfikowalną, to w trakcie kompilacji metody wywoływane dla tego obiektu, nawet jeśli nie chcą modyfikować jego pól, traktowane są przez kompilator jako funkcje, które mogą to zrobić.

Aby zapewnić kompilator, że dana metoda nie zmienia wartości pól, dodajemy na końcu słowa const (w prototypie i w definicji).

Przykład

```
class Stos
{
  public
    void Stos::print() const;
  ...
};
const Stos  s(15, 20);
s.print( );
```

Ogólnie, definiując klasę warto zaznaczyć, które metody nie modyfikują pól obiektu. Kompilator sprawdzi czy programista o tym nie zapomniał i zwiększy to czytelność programu.

8.6. Pola obiektowe

Klasy można zagnieżdżać czyli pola mogą być obiektami wcześniej zdefiniowanych klas.

Przykład

```
class Data
{
  public:
    Data (int _dzień, int _miesiac, int _rok);
     void print ( ) const;
  private:
    int dzien, miesiac, rok;
}
class String
{
  public:
    String (const char _str[]);
    void print ( ) const;
  private:
    char *str;
    int rozmiar;
};
class Osoba
{
  public:
    Osoba (const char _nazwisko[], int _dzien,  int _miesiac, int _rok);
    void print ( ) const;
  private:
    String nazwisko;
    Data data_ur;
}
Osoba::Osoba (const char _nazwisko[ ], int _dzien,
   int _miesiac, int _rok): nazwisko (_nazwisko), data_ur (_dzien,
   _miesiac, _rok)
{ }
void Osoba :: print ( ) const
{
  nazwisko.print ( ); data_ur. print ( );
}
```

W liście inicjującej konstruktora klasy Osoba do zainicjowania pól nazwisko, data_ur zostały użyte konstruktory klas Data i String.

8.7. Wskaźnik this

W przypadku korzystania w programie z wielu obiektów tej samej klasy metody generowane są w jednym egzemplarzu i udostępniane wszystkim obiektom, równocześnie każdy obiekt dostaje własny komplet pól.

Metody należące do danej klasy posiadają ukryty parametr **this**, który zawiera wskaźnik obiektu, dla którego metoda została wywołana. Można go w sposób jawny wykorzystać w metodach.

Przykłady

```
Stos st1, st2;
st1. push (10);  // st1. push (&st1, 10);
st2. push (20);  // st2. push (&st2, 20);
// void Stos :: push (Stos *this, int _element);
```

8.8. Funkcje zaprzyjaźnione

Jeśli chcemy jakiejś funkcji szybko udostępnić pola z wszystkich sekcji danej klasy, to możemy ją uczynić funkcją zaprzyjaźnioną tej klasy. W definicji klasy umieszczamy jej prototyp poprzedzony słowem **friend**. Ponieważ nie dostaje ona wskaźnika this, należy jej w sposób jawny przekazać wskaźnik obiektu danej klasy.

Przykłady

```
class K
{
  friend void f ( K *wx);
  public:
    void g ( );
  ...
};
class L
{
  friend void X :: g ( );
  ...
};
```

8.9. Pola i funkcje statyczne

W klasie można definiować pola statyczne rozpoczynając definicję słowem **static**.

Przykład

```
class Stos
{
  private:
    static int licznik;
  ...
};
```

Pola statyczne występują w jednym egzemplarzu dla wszystkich obiektów danej klasy. W metodach korzystamy z nich poprzez ich nazwę.

Przykład

```
Stos::Stos ( )
{
  licznik++;
}
```

Inicjowanie tych pól umieszczamy przed definicją pierwszej funkcji.

Przykład

```
int Stos::licznik=0;
```

Dodatkowo do obsługi tych pól można wykorzystywać metody static – w danej klasie poprzedzamy je słowem static.

Przykład

```
cout<<"Jest "<<Stos::ile( )<<" stosów";
```

Metody statyczne mogą odwoływać się tylko do pól statycznych z danej klasy (nie mają wskaźnika this).

9. Spis treści

1. Wprowadzenie do języka C++ 1
1.1. Porównanie struktury programu w Pascalu i w C++ 1
1.2. Podstawowe typy danych 1
1.3. Zmienne 1
1.4. Operacje arytmetyczne 2
1.5. Operacje porównania 2
1.6. Operacje zmiany wartości o 1 2
1.7. Instrukcja wyrażeniowa 2
1.8. Instrukcja złożona 2
1.9. Wypisywanie danych do standardowego strumienia wyjściowego 2
1.10. Instrukcja for 3
1.11. Operacja przypisania 4
1.12. Instrukcja while 4
2. Operacje dla znaków 4
2.1. Przechowywanie znaków w zmiennych typu char 4
2.2. Operacja dla wartości typu char 5
2.3. Funkcje dla konsoli (C++ Builder) 5
2.4. Włączanie plików 6
2.5. Definiowanie stałych 6
2.6. Wczytywanie danych ze standardowego strumienia wejściowego 6
2.7. Instrukcja warunkowa 6
2.8. Operacje logiczne 7
2.9. Prawa de Morgana 8
2.10. Konstrukcja else if 8
2.11. Klasyfikacja znaków 9
2.12. Zmienne licznikowe 9
2.13. Wczytanie znaku za pomocą funkcji get 10
3. Funkcje i tablice 10
3.1. Składnia definicji funkcji 10
3.2. Wykonanie funkcji 11
3.3. Instrukcja return 12
3.4. Przekazywanie parametrów przez wartość 12
3.5. Tablice 13
3.6. Tablice znakowe 13
3.7. Inicjowanie tablic 13
3.8. Wczytywanie i wyświetlanie tekstu dla tablic 14
3.9. Wykonanie instrukcji wyboru 14
3.10. Zastosowanie instrukcji wyboru 15
3.11. Porównywanie tekstu 16
3.12. Kopiowanie tekstu 16
3.13. Dołączanie tekstu 16
3.14. Operacje zmiany wartości o jeden 16
3.15. Instrukcja do while 17
3.16. Wykorzystanie poszczególnych pętli 17
3.17. Zastosowanie instrukcji break w pętlach 17
4.Typy liczbowe 17
4.1. Typy całkowite 17
4.2. Układ szesnastkowy 18

4.3. Operacje logiczne na bitach 18
4.4. Przesunięcie bitów 19
4.5. Typ wyliczeniowy 19
4.6. Nowe operacje przypisania 19
4.7. Operacja warunkowa 19
4.8. Konwersje jawne 20
4.9. Konwersje wśród typów całkowitych 20
4.10. Konwersje wśród typów rzeczywistych 20
4.11. Konwersja między typem całkowitym a rzeczywistym 20
4.12. Przyczyny konwersji niejawnej 21
4.13. Wykonywanie obliczeń na liczbach rzeczywistych z podwójną precyzją .. 21
4.14. Operacje występujące w języku C++ 22
4.15. Wpływ priorytetu na kolejność wykonywania operacji 25
4.16. Wpływ łączności na kolejność wykonywania operacji o takim samym priorytecie 25
4.17. Obliczanie argumentów operacji 22
5. Wskaźniki zmiennych i dostępne dla nich operacje 23
5.1. Zmienne wskaźnikowe 23
5.2. Operacja adresu 23
5.3. Operacja wyłuskania 23
5.4. Przekazywanie kilku wartości na zewnątrz przy pomocy wskaźników 24
5.5. Wskaźniki elementów tablicy 24
5.6. Ścisły związek między tablicami a wskaźnikami w C++ 29
5.7. Przekazywanie tablic do funkcji 24
5.8. Tablica tablic 25
5.9. Inicjowanie tablicy tablic 25
5.10. Stałe tekstowe a wskaźniki 25
5.11. Tablice wskaźników 26
5.12. Inicjowanie tablicy wskaźników 26
5.13. Porównanie tablicy tablic i tablicy wskaźników 26
6. Struktury 26
6.1. Definicja typy strukturowego 26
6.2. Definicja zmiennej strukturowej 26
6.3. Zagnieżdżanie struktur 27
6.4. Operacja składowej 27
6.5. Wskaźniki struktur 27
6.6. Operacja wyłuskania składowej 28
6.7. Tablice struktur 28
6.8. Inicjowanie struktur 28
6.9. Inicjowanie tablicy struktur 29
6.10. Zmienne niemodyfikowalne 29
6.11. Zmienne referencyjne 29
6.12. Zmienne referencyjne w roli parametrów 29
6.13. Operator new 29
6.14. Operator delete 30
6.15. Funkcje rekurencyjne 30
6.16. Tablice dynamiczne 30
6.17. Tablice wskaźników do struktur 31
6.18. Listy 31

6.19. Drzewa binarne 31
7. Klasy 31
7.1. Dzielenie programu na moduły i klasy 31
7.2. Zmienne klasy auto 31
7.3. Zmienne klasy register 32
7.4. Zmienne klasy static wewnętrzne 32
7.5. Zmienne klasy static zewnętrzne 32
7.6. Zmienne klasy extern 32
7.7. Funkcje klasy static 33
7.8. Funkcje klasy extern 33
7.9. Funkcje przeciążone 33
7.10. Pojęcie klasy jako nowego typu w C++ 33
7.11. Porównanie sekcji public i private 33
7.12. Definiowanie metod 33
7.13. Obiekty 34
7.14. Odwoływanie się do metod i pól 34
7.15. Funkcja wstawiania 34
7.16. Wskaźniki obiektów 34
7.17. Tablice obiektów 34
7.18. Obiekty dynamiczne 35
7.19. Tablice dynamiczne 35
7.20. Struktura programu z klasami 35
8. Konstruktory i destruktory 35
8.1. Wartości domyślne parametrów 35
8.2. Konstruktor obiektu 36
8.3. Destruktor 36
8.4. Pola const i lista inicjująca konstruktora 37
8.5. Obiekty i funkcje const 37
8.6. Pola obiektowe 37
8.7. Wskaźnik this 38
8.8. Funkcje zaprzyjaźnione 39
8.9. Pola i funkcje statyczne 39
9. Spis treści 40