Wstęp do programowania, część I

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

12 października 2011

Kontakt

- http://www.fuw.edu.pl/~rwys/ws_prog
- rwys@fuw.edu.pl
- tel. 22 55 32 263
- ul. Hoża 69, pok. 142 (dawniej 216)

Materiał na ćwiczenia

1. Algorytmy.
2. Podstawy programowania w C++.
   - Funkcja `main()`, nagłówek i treść funkcji.
   - Wyprowadzanie wyników z programu i wprowadzanie danych do programu.
   - Wyrażenia i zmienne.
   - Pętle.
3. Programowanie prostych obliczeń.
4. Liczby pseudolosowe.
5. Formatowanie wydruków i drukowanie wzorów.
6. Tablice i sortowanie liczb.
7. Wskaźniki i referencje.
8. Wejście/wyjście z wykorzystaniem plików.
9. Złożone typy danych i ich zastosowania.

Plan wykładu

- Dlaczego warto poznać programowanie komputerów.
- Algorytmy – jak je tworzyć i oceniać ich poprawność.
- Złożoność obliczeniowa i równoważność algorytmów.
- Składnia języka C++, elementy składowe programów.
- Proste typy danych w C++.
- Operatory i wyrażenia, priorytety operatorów i konwersje typów danych.
- Pętle, instrukcja warunkowa i instrukcje sterujące w C++.
- Wskaźniki i tablice, operacje na wskaźnikach. Referencje.
- Dynamiczne rezerwowanie i zwalnianie pamięci.
- Tablice wskaźników i argumenty programu.
- Złożone typy danych i ich zastosowania.

Dlaczego warto uczyć się programować komputery

- Żeby zobaczyć na czym to polega.
- Jeżeli mamy nietypowe zadanie, to czasami szybciej jest napisać własny program, niż szukać gotowego.
- Czasami musimy być pewni, że program działa w ściśle określony sposób i wtedy trzeba go samemu napisać.
- Może to być interesujące.
- Stanowiska pracy związane z programowaniem bywają dobrze płatne.

Na czym polega programowanie

1. Określenie problemu (co chcemy osiągnąć).
2. Wybranie algorytmu (*ang. algorithm*), czyli metody poszukiwania rozwiązania (jak będziemy to robić).
3. Tworzenie kodu źródłowego (*ang. source code*), stanowiącego reprezentację wybranego algorytmu (kodowanie).
4. Kompilowanie kodu źródłowego, czyli tworzenie kodu wykonywalnego (*ang. executable code*) gotowego do wykonania (programu).
5. Sprawdzanie i testowanie (*ang. debugging*) programu. Jeżeli znajdziemy błąd, wracamy do etapu tworzenia kodu źródłowego (lub nawet do etapu wyboru algorytmu).

Co to jest algorytm

Algorytm

Skończony zbiór dobrze zdefiniowanych instrukcji przeznaczony do wykonania określonego zadania, który przy ustalonym stanie początkowym pozwala na uzyskanie odpowiedniego, rozpoznawalnego stanu końcowego w skończonym czasie.

Co to jest algorytm

Algorytm

Skończony zbiór dobrze zdefiniowanych instrukcji przeznaczony do wykonania określonego zadania, który przy ustalonym stanie początkowym pozwala na uzyskanie odpowiedniego, rozpoznawalnego stanu końcowego w skończonym czasie.

W uproszczeniu

Metoda poszukiwania (lub tworzenia) rozwiązania zadanego problemu.

Co to jest algorytm

Algorytm

Skończony zbiór dobrze zdefiniowanych instrukcji przeznaczony do wykonania określonego zadania, który przy ustalonym stanie początkowym pozwala na uzyskanie odpowiedniego, rozpoznawalnego stanu końcowego w skończonym czasie.

W uproszczeniu

Metoda poszukiwania (lub tworzenia) rozwiązania zadanego problemu.

Nie dla każdego problemu istnieje skuteczny algorytm pozwalający znaleźć rozwiązanie.

- Problemy nieobliczalne (algorytm nie istnieje, np. *busy beaver*).
- Problemy, dla których algorytmy są zbyt złożone (wykonywanie programu trwałoby zbyt długo).

Poprawność algorytmów

Stan początkowy dla algorytmu

Dane wejściowe (*ang. input data*).

Stan końcowy dla algorytmu

Wynik (*ang. result*).

Poprawność algorytmów

Stan początkowy dla algorytmu

Dane wejściowe (*ang. input data*).

Stan końcowy dla algorytmu

Wynik (*ang. result*).

Definicja poprawności algorytmu

Algorytm jest poprawny (*ang. correct*), gdy dla każdych dopuszczalnych danych wejściowych jednocześnie spełnione są dwa następujące warunki:

1. Wynik jest otrzymywany w skończonej liczbie kroków — problem zatrzymania (stopu).
2. Wynik stanowi rozwiązanie problemu, dla którego algorytm został stworzony.

Powszechnie znane algorytmy

- Obliczanie reszty z dzielenia.
- Algorytm Euklidesa.
- Algorytm Eratostenesa (sito).
- Dodawanie liczb w systemie dwójkowym (binarnym).
- Uniwersalny algorytm mnożenia (*Russian peasant algorithm*).
- Przeszukiwanie binarne (*ang. binary search*) lub bisekcja (*ang. bisection*).

Obliczanie reszty z dzielenia

Niech n i m będą liczbami naturalnymi, $n > m$. Chcemy obliczyć resztę z dzielenia n przez m.

Obliczanie reszty z dzielenia

Niech n i m będą liczbami naturalnymi, $n > m$. Chcemy obliczyć resztę z dzielenia n przez m.

Potrzebne jest jedno miejsce do przechowywania pośrednich wyników, nazwijmy je k.

1. Niech $k = n$.
2. Jeśli $k < m$, to k jest resztą z dzielenia, $k = n \,\%\, m$.
3. Niech $k = k - m$.
4. Przejdź do kroku 2.

Algorytm Euklidesa

Niech n i m będą liczbami naturalnymi, $n > m$. Wyznaczymy największy wspólny dzielnik n i m, czyli $NWD(n, m)$.

Algorytm Euklidesa

Niech n i m będą liczbami naturalnymi, $n > m$. Wyznaczymy największy wspólny dzielnik n i m, czyli $NWD(n, m)$.

Obserwacja

Jeśli r jest resztą z dzielenia n przez m, to $n = km + r$ (dla pewnego całkowitego k), więc $NWD(n, m) = NWD(m, r)$.

Algorytm Euklidesa

Niech n i m będą liczbami naturalnymi, $n > m$. Wyznaczymy największy wspólny dzielnik n i m, czyli $NWD(n, m)$.

Obserwacja

Jeśli r jest resztą z dzielenia n przez m, to $n = km + r$ (dla pewnego całkowitego k), więc $NWD(n, m) = NWD(m, r)$.

Niech a, b i r będą miejscami do przechowywania pośrednich wyników.

1. Niech $a = n$ i $b = m$.
2. Niech $r = a \,\%\, b$ (resztę z dzielenia a/b zapisujemy w r).
3. Jeśli $r = 0$, to $NWD(n, m) = b$.
4. Niech $a = b$ i $b = r$.
5. Przejdź do kroku 2.

Sito Eratostenesa

Chcemy ustalić które z liczb naturalnych nie większych od zadanego N są liczbami pierwszymi.

Sito Eratostenesa

Chcemy ustalić które z liczb naturalnych nie większych od zadanego N są liczbami pierwszymi.

Niech $A = \{2, 3, ..., N\}$ będzie zbiorem liczb, natomiast k i m – miejscami do przechowywania pośrednich wyników.

1. Niech $k = 2$.
2. Jeśli $k^2 > N$, zbiór A zawiera tylko liczby pierwsze.
3. Usuwamy z A wszystkie wielokrotności k, począwszy od k^2.
4. W m zapisz najmniejszą liczbą ze zbioru A większą od k.
5. Niech $k = m$.
6. Przejdź do kroku 2.

Dodawanie liczb w systemie dwójkowym

Mamy dwie liczby N-cyfrowe w zapisie dwójkowym:

$$a = \sum_{j=0}^{N-1} a_j 2^j \,, \quad b = \sum_{j=0}^{N-1} b_j 2^j \,,$$

gdzie $a_j, b_j \in \{0, 1\}$ są cyframi. Chcemy wyznaczyć współczynniki (cyfry) $w_j \in \{0, 1\}$ w rozwinięciu ich sumy:

$$a + b = w = \sum_{j=0}^{N} w_j 2^j \,,$$

dla danych a_j oraz b_j.

Dodawanie liczb w systemie dwójkowym c. d.

Wprowadzamy funkcje:

$$PAR(x, y, z) = (x + y + z) \,\%\, 2$$

$$MAJ(x, y, z) = (x + y + z)/2$$

gdzie $x, y, z \in \{0, 1\}$, a znak / oznacza dzielenie z pominięciem reszty.

Dodawanie liczb w systemie dwójkowym c. d.

Wprowadzamy funkcje:

$$PAR(x, y, z) = (x + y + z) \,\%\, 2$$

$$MAJ(x, y, z) = (x + y + z)/2$$

gdzie $x, y, z \in \{0, 1\}$, a znak / oznacza dzielenie z pominięciem reszty.

Definiujemy ciąg $c_0 = 0, c_j = MAJ(a_{j-1}, b_{j-1}, c_{j-1})$ dla $j = 1, 2, \ldots, N$.

Dodawanie liczb w systemie dwójkowym c. d.

Wprowadzamy funkcje:

$$PAR(x, y, z) = (x + y + z) \,\%\, 2$$

$$MAJ(x, y, z) = (x + y + z)/2$$

gdzie $x, y, z \in \{0, 1\}$, a znak / oznacza dzielenie z pominięciem reszty.

Definiujemy ciąg $c_0 = 0, c_j = MAJ(a_{j-1}, b_{j-1}, c_{j-1})$ dla $j = 1, 2, \ldots, N$.

Wtedy c_j jest przeniesieniem (*ang. carry*) z pozycji $(j - 1)$ na pozycję j podczas dodawania, więc mamy $w_N = c_N$ oraz $w_j = PAR(a_j, b_j, c_j)$ dla $j = 0, 1, \ldots, N - 1$.

Dodawanie liczb w systemie dwójkowym c. d.

Algorytm dodawania

Niech C będzie miejscem służącym do przechowywania pośrednich wartości, natomiast k będzie licznikiem.

1. Niech $C = 0$, $k = 0$.
2. Niech $w_k = PAR(a_k, b_k, C)$.
3. Niech $C = MAJ(a_k, b_k, C)$.
4. Niech $k = k + 1$.
5. Jeśli $k < N$, przejdź do kroku 2.
6. Niech $w_N = C$.
7. Ciąg cyfr $w_0, w_1, w_2, \ldots, w_N$ stanowi wynik.

Uniwersalny algorytm mnożenia

Russian peasant algorithm

Obliczamy wynik mnożenia dwóch nieujemnych liczb całkowitych a i b:

1. Niech $a_0 = a$, $b_0 = b$ i $k = 0$.
2. Niech $k = k + 1$.
3. Niech $a_k = a_{k-1}/2$ (dzielenie bez reszty) i $b_k = 2b_{k-1}$.
4. Jeśli $a_k > 1$, to przejdź do kroku 2.
5. Wynik mnożenia jest sumą wszystkich b_k ($k = 0, 1, 2, \ldots$), dla których odpowiadające im a_k są nieparzyste.

Uniwersalny algorytm mnożenia – przykład

Obliczamy iloczyn 156×18

1. Wyznaczamy ciągi a_k i b_k:

k	a_k	b_k
0	156	18
1	78	36
2	39	72
3	19	144
4	9	288
5	4	576
6	2	1152
7	1	2304

2. $156 \times 18 = 2304 + 288 + 144 + 72 = 2808$

Uniwersalny algorytm mnożenia – przykład c. d.

Obliczamy iloczyn 156×18

1. Można sumować na bieżąco:

k	a_k	b_k	s_k
0	156	18	0
1	78	36	0
2	39	72	72
3	19	144	216
4	9	288	504
5	4	576	504
6	2	1152	504
7	1	2304	2808

2. $156 \times 18 = s_7 = 2808$

Uniwersalny algorytm mnożenia z sumowaniem na bieżąco

Obliczamy iloczyn dwóch nieujemnych liczb całkowitych a i b

Niech A, B i S będą miejscami służącymi do przechowywania pośrednich wyników.

1. Niech $A = a$, $B = b$ i $S = 0$.
2. Jeśli $A \,\%\, 2 = 1$ (wartość w A jest nieparzysta), to $S = S + B$.
3. Jeśli $A = 1$, przejdź do kroku 6.
4. Niech $A = A/2$ (dzielenie bez reszty) i $B = 2B$.
5. Przejdź do kroku 2.
6. Wynik mnożenia jest zapisany w S.

Przeszukiwanie binarne (bisekcja)

Poszukiwanie elementu r w skończonym i uporządkowanym zbiorze B

1. Niech $B_0 = B$ i $k = 0$.
2. Jeśli wszystkie elementy zbioru B_k są jednakowe lub B_k jest zbiorem pustym, przejdź do kroku 6.
3. Niech m_k oznacza medianę zbioru B_k.
4. Jeśli $r \leqslant m_k$, to niech $B_{k+1} = \{b \in B_k : b \leqslant m_k\}$, a w przeciwnym wypadku niech $B_{k+1} = \{b \in B_k : b > m_k\}$.
5. Niech $k = k + 1$ i przejdź do kroku 2.
6. Jeśli $r \in B_k$, to $r \in B$.

Przeszukiwanie binarne (bisekcja) – przykład

Weźmy $B = \{3, 7, 21, 48, 56, 122, 141, 218, 225, 248\}$ i $r = 225$

k	B_k	m_k
0	$\{3, 7, 21, 48, 56, 122, 141, 218, 225, 248\}$	56
1	$\{122, 141, 218, 225, 248\}$	218
2	$\{225, 248\}$	225
3	$\{225\}$	225

Złożoność obliczeniowa algorytmu

Złożoność obliczeniowa algorytmu

Miara liczby operacji przeprowadzanych podczas wykonywania programu stanowiącego realizację (zapis) danego algorytmu.

Złożoność obliczeniowa algorytmu

Złożoność obliczeniowa algorytmu

Miara liczby operacji przeprowadzanych podczas wykonywania programu stanowiącego realizację (zapis) danego algorytmu.

Obserwacje

1. Złożoność obliczeniowa algorytmu jest funkcją rozmiarów jego danych wejściowych, czyli rozmiaru zadania.
2. Zwykle jest ona szacowana (w przybliżeniu) dla rozmiaru zadania dążącego do nieskończoności.
3. Czas przeprowadzania obliczeń jest ściśle zależny od złożoności obliczeniowej algorytmu.

Złożoność obliczeniowa – szczególne przypadki

Dla rozmiaru zadania reprezentowanego przez jedną liczbę N

Stała – liczba operacji nie zależy od N.

Logarytmiczna – liczba operacji jest proporcjonalna do $\log N$.

Liniowa – liczba operacji jest wprost proporcjonalna do N.

Wielomianowa – liczba operacji jest proporcjonalna N^k, gdzie $k > 1$.

NP – weryfikacja rozwiązania jest problemem o wielomianowej złożoności obliczeniowej, ale poszukiwanie rozwiązania może mieć większą złożoność.

Złożoność obliczeniowa – szczególne przypadki

Dla rozmiaru zadania reprezentowanego przez jedną liczbę N

Stała – liczba operacji nie zależy od N.

Logarytmiczna – liczba operacji jest proporcjonalna do $\log N$.

Liniowa – liczba operacji jest wprost proporcjonalna do N.

Wielomianowa – liczba operacji jest proporcjonalna N^k, gdzie $k > 1$.

NP – weryfikacja rozwiązania jest problemem o wielomianowej złożoności obliczeniowej, ale poszukiwanie rozwiązania może mieć większą złożoność.

Często czas działania programu wyraża się poprzez jego złożoność obliczeniową, także na przykład jeśli algorytm ma wielomianową złożoność obliczeniową mówi się, że jest on wykonywany „w czasie wielomianowym".

Złożoność obliczeniowa – klasy *P* i *NP*

Klasa *P* (*ang. polynomial*)

Obejmuje wszystkie algorytmy o wielomianowej złożoności obliczeniowej.

Złożoność obliczeniowa – klasy *P* i *NP*

Klasa *P* (*ang. polynomial*)

Obejmuje wszystkie algorytmy o wielomianowej złożoności obliczeniowej.

Klasa *NP* (*ang. nondeterministic polynomial*)

Obejmuje wszystkie algorytmy (zadania), dla których weryfikacja wyniku jest problemem klasy *P*.

Złożoność obliczeniowa – klasy *P* i *NP*

Klasa *P* (*ang. polynomial*)

Obejmuje wszystkie algorytmy o wielomianowej złożoności obliczeniowej.

Klasa *NP* (*ang. nondeterministic polynomial*)

Obejmuje wszystkie algorytmy (zadania), dla których weryfikacja wyniku jest problemem klasy *P*.

Twierdzenie

Każdy problem klasy *NP* można sprowadzić do innego problemu tej klasy z użyciem algorytmu o wielomianowej złożoności obliczeniowej (klasy *P*).

Złożoność obliczeniowa – klasy *P* i *NP*

Klasa *P* (*ang. polynomial*)

Obejmuje wszystkie algorytmy o wielomianowej złożoności obliczeniowej.

Klasa *NP* (*ang. nondeterministic polynomial*)

Obejmuje wszystkie algorytmy (zadania), dla których weryfikacja wyniku jest problemem klasy *P*.

Twierdzenie

Każdy problem klasy *NP* można sprowadzić do innego problemu tej klasy z użyciem algorytmu o wielomianowej złożoności obliczeniowej (klasy *P*).

Hipoteza $P \neq NP$

Weryfikacja tej hipotezy jest jedenym z tzw. problemów tysiąclecia (*ang. millennium problems*).

Złożoność obliczeniowa – przykłady

Dla wyznaczania reszty z dzielenia

Liniowa względem ilorazu n/m.

Złożoność obliczeniowa – przykłady

Dla wyznaczania reszty z dzielenia

Liniowa względem ilorazu n/m.

Dla dodawania liczb w systemie dwójkowym

Liniowa względem N.

Złożoność obliczeniowa – algorytm Euklidesa

W najgorszym wypadku logarytmiczna względem n (zakładając, że obliczanie reszty z dzielenia ma stałą złożoność).

1. W najgorszym wypadku w każdym kroku mamy

$$b \sim r \sim \frac{a}{2}$$

Złożoność obliczeniowa – algorytm Euklidesa

W najgorszym wypadku logarytmiczna względem n (zakładając, że obliczanie reszty z dzielenia ma stałą złożoność).

1. W najgorszym wypadku w każdym kroku mamy
$$b \sim r \sim \frac{a}{2}$$
2. Zatem (w najgorszym wypadku) w kroku k
$$b \sim r \sim \frac{n}{2^k}$$

Złożoność obliczeniowa – algorytm Euklidesa

W najgorszym wypadku logarytmiczna względem n (zakładając, że obliczanie reszty z dzielenia ma stałą złożoność).

1. W najgorszym wypadku w każdym kroku mamy
$$b \sim r \sim \frac{a}{2}$$
2. Zatem (w najgorszym wypadku) w kroku k
$$b \sim r \sim \frac{n}{2^k}$$
3. Zatem (w najgorszym wypadku) w ostatnim kroku, czyli dla $k = s$, gdzie s jest całkowitą liczbą wykonanych kroków
$$1 \sim r \sim \frac{n}{2^s} \Longrightarrow s \sim \log_2 n$$

Złożoność obliczeniowa – sito Eratostenesa

Kwadratowa względem N.

1. Liczba powtórzeń „usuwania wielokrotności” jest rzędu N.

Złożoność obliczeniowa – sito Eratostenesa

Kwadratowa względem N.

1. Liczba powtórzeń „usuwania wielokrotności” jest rzędu N.
2. W każdym powtórzeniu przeprowadzamy $(N - k^2)/k$ operacji.

Złożoność obliczeniowa – sito Eratostenesa

Kwadratowa względem N.

1. Liczba powtórzeń „usuwania wielokrotności” jest rzędu N.
2. W każdym powtórzeniu przeprowadzamy $(N - k^2)/k$ operacji.
3. Łączna liczba przeprowadzanych operacji s jest proporcjonalna do

$$\frac{N\left(N - \langle k \rangle^2\right)}{\langle k \rangle} = \frac{N^2}{\langle k \rangle} - \langle k \rangle N$$

gdzie $\langle k \rangle$ jest średnią wartością k.

Złożoność obliczeniowa – sito Eratostenesa

Kwadratowa względem N.

1. Liczba powtórzeń „usuwania wielokrotności” jest rzędu N.
2. W każdym powtórzeniu przeprowadzamy $(N - k^2)/k$ operacji.
3. Łączna liczba przeprowadzanych operacji s jest proporcjonalna do

$$\frac{N\left(N - \langle k \rangle^2\right)}{\langle k \rangle} = \frac{N^2}{\langle k \rangle} - \langle k \rangle N$$

   gdzie $\langle k \rangle$ jest średnią wartością k.
4. Przy N dążącym do nieskończoności pierwszy człon jest dominujący, więc mamy

$$s \sim N^2$$

Złożoność obliczeniowa – uniwersalne mnożenie

Logarytmiczna względem a przy założeniu, że dodawanie ma stałą złożoność.

1. Mamy $a_k = a/2^k$.

Złożoność obliczeniowa – uniwersalne mnożenie

Logarytmiczna względem a przy założeniu, że dodawanie ma stałą złożoność.

1. Mamy $a_k = a/2^k$.
2. Zatem, dla k równego liczbie kroków s dostajemy $1 = a_s = a/2^s$.

Złożoność obliczeniowa – uniwersalne mnożenie

Logarytmiczna względem a przy założeniu, że dodawanie ma stałą złożoność.

1. Mamy $a_k = a/2^k$.
2. Zatem, dla k równego liczbie kroków s dostajemy $1 = a_s = a/2^s$.
3. Stąd wynika, że $s \sim \log_2 a$.

Złożoność obliczeniowa – bisekcja

Logarytmiczna względem liczby elementów B.

1. Dla każdego k moc (liczba elementów) zbioru B_{k+1} jest średnio o połowę mniejsza od mocy (liczby elementów) zbioru B_k.

Złożoność obliczeniowa – bisekcja

Logarytmiczna względem liczby elementów B.

1. Dla każdego k moc (liczba elementów) zbioru B_{k+1} jest średnio o połowę mniejsza od mocy (liczby elementów) zbioru B_k.
2. Zatem dla $k = s$ (liczba kroków) moc zbioru B_s jest rzędu mocy zbioru B podzielonej przez 2^s.

Złożoność obliczeniowa – bisekcja

Logarytmiczna względem liczby elementów B.

1. Dla każdego k moc (liczba elementów) zbioru B_{k+1} jest średnio o połowę mniejsza od mocy (liczby elementów) zbioru B_k.
2. Zatem dla $k = s$ (liczba kroków) moc zbioru B_s jest rzędu mocy zbioru B podzielonej przez 2^s.
3. Stąd wynika, że $s \sim \log_2 \bar{B}$.

Równoważność algorytmów

Definicja równoważności algorytmów

Dwa algorytmy można uznać za równoważne (*ang. equivalent*), gdy:

1. Każdy z nich pozwala wyznaczyć rozwiązanie tego samego problemu.
2. Mają one jednakową złożoność obliczeniową (z dokładnością do stałego czynnika).

Równoważność algorytmów

Definicja równoważności algorytmów

Dwa algorytmy można uznać za równoważne (*ang. equivalent*), gdy:

1. Każdy z nich pozwala wyznaczyć rozwiązanie tego samego problemu.
2. Mają one jednakową złożoność obliczeniową (z dokładnością do stałego czynnika).

Kod źródłowy, stanowiący zapis algorytmu, jest jego realizacją (*ang. implementation*), podobnie jak kod wykonywalny (program) powstający po jego skompilowaniu.

Równoważność algorytmów

Definicja równoważności algorytmów

Dwa algorytmy można uznać za równoważne (*ang. equivalent*), gdy:

1. Każdy z nich pozwala wyznaczyć rozwiązanie tego samego problemu.
2. Mają one jednakową złożoność obliczeniową (z dokładnością do stałego czynnika).

Kod źródłowy, stanowiący zapis algorytmu, jest jego realizacją (*ang. implementation*), podobnie jak kod wykonywalny (program) powstający po jego skompilowaniu.

Obserwacja

Wszystkie realizacje (implementacje) tego samego algorytmu są równoważne (tzn. zapis algorytmu w jednym języku programowania jest równoważny jego zapisowi w innym języku programowania).

Równoważność algorytmów c. d.

Obserwacja

Można powiedzieć, że algorytm jest klasą abstrakcji relacji równoważności w zbiorze wszystkich możliwych programów, jakie można zapisać w dowolnym języku programowania.

Instrukcje i bloki w C++

Instrukcja lub zapis (*ang. statememt*)

Ciąg znaków (w kodzie źródłowym), w wyniku skompilowania którego generowany jest ciąg rozkazów dla procesora. Przeważnie zajmuje jeden wiersz i kończy się średnikiem (instrukcje są odpowiednikami zdań).

Instrukcje i bloki w C++

Instrukcja lub zapis (*ang. statememt*)

Ciąg znaków (w kodzie źródłowym), w wyniku skompilowania którego generowany jest ciąg rozkazów dla procesora. Przeważnie zajmuje jeden wiersz i kończy się średnikiem (instrukcje są odpowiednikami zdań).

Blok (*ang. block*)

Ciąg instrukcji w nawiasach klamrowych { . . . }.

Instrukcje i bloki w C++

Instrukcja lub zapis (*ang. statememt*)

Ciąg znaków (w kodzie źródłowym), w wyniku skompilowania którego generowany jest ciąg rozkazów dla procesora. Przeważnie zajmuje jeden wiersz i kończy się średnikiem (instrukcje są odpowiednikami zdań).

Blok (*ang. block*)

Ciąg instrukcji w nawiasach klamrowych { . . . }.

Pojedynczą instrukcję można zawsze zastąpić blokiem

W związku z tym w ogólności blok może być ciągiem instrukcji oraz bloków (zastępujących instrukcje) w nawiasach klamrowych.

Funkcje w C++

Funkcja (*ang. function*)

Wyróżniona część kodu źródłowego, stanowiąca odrębną całość, składająca się z treści (*ang. body*) oraz nagłówka (*ang. header*).

Funkcje w C++

Funkcja (*ang. function*)

Wyróżniona część kodu źródłowego, stanowiąca odrębną całość, składająca się z treści (*ang. body*) oraz nagłówka (*ang. header*).

Treść funkcji (*ang. function body*)

Blok, którego sposób wykonywania w ogólności zależy od pewnej liczby wartości, zwanych parametrami lub argumentami funkcji, pochodzących z innych części programu i kończy się wygenerowaniem wyniku (*ang. result*), przeznaczonego do wykorzystania w innych częściach programu.

Funkcje w C++

Funkcja (*ang. function*)

Wyróżniona część kodu źródłowego, stanowiąca odrębną całość, składająca się z treści (*ang. body*) oraz nagłówka (*ang. header*).

Treść funkcji (*ang. function body*)

Blok, którego sposób wykonywania w ogólności zależy od pewnej liczby wartości, zwanych parametrami lub argumentami funkcji, pochodzących z innych części programu i kończy się wygenerowaniem wyniku (*ang. result*), przeznaczonego do wykorzystania w innych częściach programu.

Nagłówek funkcji (*ang. function header*)

Część definicji funkcji określająca typ danych dla wyniku, nazwę funkcji oraz listę jej argumentów.

Definiowanie funkcji w C++

Przykład definicji funkcji

```
int main()
{
    cout << "Napis" << endl;  // Instrukcja.
    return 0;  // Instrukcja.
}
```

Ta funkcja ma następujące własności:

1. Generowany (zwracany) przez nią wynik jest typu `int`.
2. Jej nazwą jest `main`.
3. Lista jej argumentów jest pusta.
4. Zawsze zwraca wynik 0.

Zasady związane z definiowaniem funkcji

1. Każda funkcja wykorzystywana w programie musi być zdefiniowana lub dostępna w pewnej bibliotece (*ang. library*).
2. Dla funkcji z bibliotek w programie umieszcza się same nagłówki.

Zasady związane z definiowaniem funkcji

1. Każda funkcja wykorzystywana w programie musi być zdefiniowana lub dostępna w pewnej bibliotece (*ang. library*).
2. Dla funkcji z bibliotek w programie umieszcza się same nagłówki.

Plik nagłówkowy (*ang. header file*)

Plik zawierający nagłówki funkcji dostępnych w bibliotekach oraz deklaracje stałych i zmiennych, a także definicje złożonych typów danych.

Zasady związane z definiowaniem funkcji

1. Każda funkcja wykorzystywana w programie musi być zdefiniowana lub dostępna w pewnej bibliotece (*ang. library*).
2. Dla funkcji z bibliotek w programie umieszcza się same nagłówki.

Plik nagłówkowy (*ang. header file*)

Plik zawierający nagłówki funkcji dostępnych w bibliotekach oraz deklaracje stałych i zmiennych, a także definicje złożonych typów danych.

Włączanie plików nagłówkowych do kodu źródłowego

Do tego celu służy dyrektywa `#include`, np. `#include <iostream>` (nawiasy < oraz > oznaczają, że plik powinien znajdować się w jednym z systemowych katalogów z plikami nagłówkowymi).

Przestrzenie nazw w C++

Przestrzeń nazw (*ang. namespace*)

Abstrakcyjny kontekst („pojemnik"), względem którego definiuje się identyfikatory (nazwy) różnych rzeczy (np. funkcji).

Przestrzenie nazw w C++

Przestrzeń nazw (*ang. namespace*)

Abstrakcyjny kontekst („pojemnik"), względem którego definiuje się identyfikatory (nazwy) różnych rzeczy (np. funkcji).

Nazwy funkcji (a także zmiennych i złożonych typów danych) muszą być unikatowe w obrębie używanej przestrzeni nazw, ale dwie różne przestrzenie nazw mogą zawierać tę samą nazwę.

Przestrzenie nazw w C++

Przestrzeń nazw (*ang. namespace*)

Abstrakcyjny kontekst („pojemnik"), względem którego definiuje się identyfikatory (nazwy) różnych rzeczy (np. funkcji).

Nazwy funkcji (a także zmiennych i złożonych typów danych) muszą być unikatowe w obrębie używanej przestrzeni nazw, ale dwie różne przestrzenie nazw mogą zawierać tę samą nazwę.

Deklarowanie używanej przestrzeni nazw

`using namespace std;` oznacza, że w programie (kodzie źródłowym) będą używane symbole (nazwy) zdefiniowane względem standardowej przestrzeni nazw (znajdujące się w tej przestrzeni). Wtedy nazwy funkcji (zmiennych itd.) zdefiniowanych w programie również należą do standardowej przestrzeni nazw.

Co zawierają instrukcje

Literały (*ang. literal*) – wartości (np. napisy, liczby) wpisane w kod źródłowy, kopiowane do kodu wykonywalnego podczas kompilacji.

Stałe (*ang. constant*) – symbole (nazwy) reprezentujące ustalone wartości, zastępowane tymi wartościami podczas kompilacji.

Wyrażenia (*ang. expression*) – zapis obliczeń.

Zmienne (*ang. variable*) lub nazwy zmiennych – symbole reprezentujące wartości, które mogą się zmieniać w czasie wykonywania programu.

Słowa kluczowe (*ang. keyword*) – symbole reprezentujące określone działania (np. `for`, `do`, `while`, `if`, `else`, `return`).

Wywołania funkcji (*ang. function call*) – symbole oznaczające miejsca w programie, w których należy wykonać instrukcje wchodzące w skład treści danej funkcji (wyniki wykonania funkcji też „pojawiają się" w tych miejscach).

Znaki przerwy i komentarze

Znaki przerwy (*ang. white space*)

- Spacje (*ang. space*).
- Tabulacje poziome (*ang. horizontal tab*).
- Przejścia do następnego wiersza (*ang. newline*).

Są pomijane podczas kompilacji, tylko istnieją pewne ograniczenia dotyczące miejsc, w którym może być umieszczone przejście do następnego wiersza (o ile nie zostanie poprzedzone znakiem \ (*ang. backslash*)).

Znaki przerwy i komentarze

Znaki przerwy (*ang. white space*)

- Spacje (*ang. space*).
- Tabulacje poziome (*ang. horizontal tab*).
- Przejścia do następnego wiersza (*ang. newline*).

Są pomijane podczas kompilacji, tylko istnieją pewne ograniczenia dotyczące miejsc, w którym może być umieszczone przejście do następnego wiersza (o ile nie zostanie poprzedzone znakiem \ (*ang. backslash*)).

Komentarze (*ang. comment*)

1. Dowolne znaki po symbolu // do końca wiersza.
2. Dowolne znaki między symbolami /* oraz */.

Są ignorowane przez kompilator.

Zmienne (*ang. variable*)

1. Kod źródłowy – symbole reprezentujące wartości.
2. Program w trakcie wykonywania – obszary pamięci służące do przechowywania danych.

Zmienne (*ang. variable*)

1. Kod źródłowy – symbole reprezentujące wartości.
2. Program w trakcie wykonywania – obszary pamięci służące do przechowywania danych.

Na poziomie kodu źródłowego

1. Zmienna może reprezentować różne wartości w różnych punktach programu.
2. Typ danych określa zbiór wartości, które mogą być reprezentowane.

Zmienne (*ang. variable*)

1. Kod źródłowy – symbole reprezentujące wartości.
2. Program w trakcie wykonywania – obszary pamięci służące do przechowywania danych.

Na poziomie kodu źródłowego

1. Zmienna może reprezentować różne wartości w różnych punktach programu.
2. Typ danych określa zbiór wartości, które mogą być reprezentowane.

Na poziomie programu w trakcie wykonywania

1. Zmienna ma ustalone położenie (w pamięci) i rozmiar (liczbę bitów danych, które można w niej zapisać).
2. Typ danych określa rozmiar zmiennej oraz interpretację zapisywanych w niej danych, w tym także sposób przeprowadzania operacji na nich.

Wartość zmiennej

1. Na poziomie kodu źródłowego – wartość reprezentowana przez zmienną w danym punkcie programu (tzn. po wykonaniu wszystkich poprzedzających instrukcji).
2. Na poziomie programu w trakcie wykonywania – wartość odpowiadająca ciągowi bitów przechowywanemu w zmiennej w danej chwili czasu zgodnie z interpretacją określoną przez odpowiadający jej typ danych.

Wartość zmiennej

1. Na poziomie kodu źródłowego – wartość reprezentowana przez zmienną w danym punkcie programu (tzn. po wykonaniu wszystkich poprzedzających instrukcji).
2. Na poziomie programu w trakcie wykonywania – wartość odpowiadająca ciągowi bitów przechowywanemu w zmiennej w danej chwili czasu zgodnie z interpretacją określoną przez odpowiadający jej typ danych.

Odwołanie (*ang. reference*) do zmiennej

Ma miejsce wtedy, gdy w instrukcji znajduje się nazwa zmiennej.

Odczyt (*ang. read*) – wartość zmiennej jest wykorzystywana w jakiejś operacji (np. arytmetycznej).

Zapis (*ang. write*) – wartość zmiennej podlega modyfikacji.

Modyfikowanie wartości zmiennych

Operacja przypisania =

Jeżeli nazwa zmiennej znajduje się po lewej stronie symbolu przypisania =, to otrzymuje ona nową wartość, np.

```
suma = 1;  // Odtąd zmienna suma będzie mieć wartość 1.
```

Modyfikowanie wartości zmiennych

Operacja przypisania =

Jeżeli nazwa zmiennej znajduje się po lewej stronie symbolu przypisania =, to otrzymuje ona nową wartość, np.

```
suma = 1;  // Odtąd zmienna suma będzie mieć wartość 1.
```

Operacje modyfikacji +=, -=, *=, /=, ...

Jeżeli nazwa zmiennej znajduje się po lewej stronie jendego z symboli modyfikacji +=, -=, *=, /=, ..., to jej wartość jest zmieniana w sposób określony przez ten symbol, np.

```
suma += 1;  // Odtąd zmienna suma będzie większa o 1.
```

Inkrementacja i dekrementacja

Inkrementacja (*ang. incrementation*), ++

Symbol inkrementacji (++) oznacza, że w tym miejscu programu wartość zmiennej ma być zwiększona o 1.

```
suma++;  // Zwiększ wartość zmiennej suma o 1.
```

Inkrementacja i dekrementacja

Inkrementacja (*ang. incrementation*), ++

Symbol inkrementacji (++) oznacza, że w tym miejscu programu wartość zmiennej ma być zwiększona o 1.

```
suma++;  // Zwiększ wartość zmiennej suma o 1.
```

Dekrementacja (*ang. decrementation*), --

Symbol dekrementacji (--) oznacza, że w tym miejscu programu wartość zmiennej ma być zmniejszona o 1.

```
suma--;  // Zmniejsz wartość zmiennej suma o 1.
```

Inkrementacja i dekrementacja w wyrażeniach

Przed nazwą zmiennej lub po nazwie zmiennej

```
suma++;   // Postinkrementacja.
++suma;   // Preinkrementacja.
suma--;   // Postdekrementacja.
--suma;   // Predekrementacja.
```

Inkrementacja i dekrementacja w wyrażeniach

Przed nazwą zmiennej lub po nazwie zmiennej

```
suma++;   // Postinkrementacja.
++suma;   // Preinkrementacja.
suma--;   // Postdekrementacja.
--suma;   // Predekrementacja.
```

Operacje inkrementacji i dekrementacji można łączyć z innymi operacjami

- Zwiększanie wartości zmiennej `k` po obliczeniu wyrażenia:

  ```
  suma += 1.0 / k++;
  ```

- Zwiększanie wartości zmiennej `k` przed obliczeniem wyrażenia:

  ```
  suma += 1.0 / ++k;
  ```

Liczby całkowite – reprezentacja uzupełnienia do 2

b – liczba całkowita (może być ujemna)

$$b = -b_{N-1}2^{N-1} + \sum_{j=0}^{N-2} b_j 2^j = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \ldots + b_1 2^1 + b_0 2^0$$

Typy danych dla reprezentacji uzupełnienia do 2

$N = 8$: `char` (od $-2^7 = -128$ do $127 = 2^7 - 1$)

$N = 16$: `short int` (od $-2^{15} = -32768$ do $32767 = 2^{15} - 1$)

$N = 32$: `int` (od -2^{31} do $2^{31} - 1$)

$N = 64$: `long int` lub `long long int` (od -2^{63} do $2^{63} - 1$)

Dla procesorów 32-bitowych z rodziny x86 typ `long int` jest równoważny typowi `int`.

Liczby całkowite nieujemne – reprezentacja bezznakowa

b – nieujemna liczba całkowita

$$b = \sum_{j=0}^{N-1} b_j 2^j = b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \ldots + b_1 2^1 + b_0 2^0$$

Typy danych dla reprezentacji bezznakowej

$N = 8$: `unsigned char` (od 0 do $255 = 2^8 - 1$)

$N = 16$: `unsigned short int` (od 0 do $65535 = 2^{16} - 1$)

$N = 32$: `unsigned int` (od 0 do $2^{32} - 1$)

$N = 64$: `unsigned long long int` (od 0 do $2^{64} - 1$), dla procesorów 64-bitowych równoważny `unsigned long int`.

Reprezentacja zmiennoprzecinkowa o pojedynczej precyzji

Typ `float`

$$r = \pm S_r \times 2^{W_r}$$

1. Słowo 32-bitowe.
2. b_{31} = znak.
3. $b_{30} \ldots b_{23} = W_r$ (od -126 do 127).
4. Postać znormalizowana:

$$S_r = 1 + \sum_{j=1}^{23} b_{23-j} 2^{-j}$$

5. Postać zdenormalizowana:

$$W_r = -126, \quad S_r = \sum_{j=1}^{23} b_{23-j} 2^{-j}$$

Reprezentacja zmiennoprzecinkowa o podwójnej precyzji

Typ `double`

$$r = \pm S_r \times 2^{W_r}$$

1. Słowo 64-bitowe.
2. b_{63} = znak.
3. $b_{62} \ldots b_{52} = W_r$ (od -1022 do 1023).
4. Postać znormalizowana:

$$S_r = 1 + \sum_{j=1}^{52} b_{52-j} 2^{-j}$$

5. Postać zdenormalizowana:

$$W_r = -1022, \quad S_r = \sum_{j=1}^{52} b_{52-j} 2^{-j}$$

Typ Boole'owski

`bool`

Typ danych reprezentujący wartości wyrażeń, w których występują spójniki logiczne i operacje porównania. Zmienne tego typu mogą przyjmować dwie wartości, `true` oraz `false`.

Typ Boole'owski

`bool`

Typ danych reprezentujący wartości wyrażeń, w których występują spójniki logiczne i operacje porównania. Zmienne tego typu mogą przyjmować dwie wartości, `true` oraz `false`.

Związek typu `bool` z typami liczbowymi

Wartość o dowolnym liczbowym typie danych może zastępować wartość typu `bool`. Wówczas wartość liczbowa 0 odpowiada wartości `false` typu `bool`, natomiast każda liczbowa wartość różna od 0 odpowiada wartości `true` typu `bool`.

Wyrażenia

Wyrażenia reprezentują obliczenia

Zawierają one:

1. Symbole reprezentujące wartości do wykorzystania w obliczeniach.
   - Literały.
   - Zmienne (nazwy zmiennych).
   - Stałe (nazwy stałych).
2. Wywołania funkcji, z których pochodzą wartości do wykorzystania w obliczeniach, np. `s = sin(x);`
3. Operatory.

Wyrażenia

Wyrażenia reprezentują obliczenia

Zawierają one:

1. Symbole reprezentujące wartości do wykorzystania w obliczeniach.
   - Literały.
   - Zmienne (nazwy zmiennych).
   - Stałe (nazwy stałych).
2. Wywołania funkcji, z których pochodzą wartości do wykorzystania w obliczeniach, np. `s = sin(x);`
3. Operatory.

Określoną kolejność przeprowadzania obliczeń można wymusić stosując nawiasy okrągłe (tzn. nawiasy okrągłe służą do grupowania składników wyrażenia). Innych rodzajów nawiasów nie używa się do tego celu.

Wyrażenia – typy danych dla wyników

Typ danych dla wyniku wyrażenia zależy od typów danych dla wartości w wyrażeniu.

Wyrażenia – typy danych dla wyników

Typ danych dla wyniku wyrażenia zależy od typów danych dla wartości w wyrażeniu.

Zasady ustalania typu danych dla wyniku obliczeń

1. Jeżeli wszystkie wartości wykorzystywane w obliczeniach są tego samego typu, wynik też jest tego typu (z wyjątkiem operacji porównania).
2. Wartości różnych typów są sprowadzane do tego samego typu danych przed przeprowadzeniem operacji. Czynność ta nazywana jest konwersją typów danych.
3. Wartości o „prostszych” typach danych są zamieniane (konwertowane) na wartości o „ogólniejszych” typach danych.
4. Ustalanie typu danych dla wyniku jest przeprowadzane dla każdej operacji z osobna.

Hierarchia typów danych przy automatycznych konwersjach

1. `char`
2. `unsigned char`
3. `short int`
4. `unsigned short int`
5. `int`
6. `unsigned int`
7. `long int`
8. `unsigned long int`
9. `long long int`
10. `unsigned long long int`
11. `float`
12. `double`

Przykłady automatycznych konwersji typów danych

- W wyrażeniu $a + b$, jeżeli a jest typu `double`, zaś b jest dowolnego innego prostego typu, to wartość b zostanie skonwertowana do typu `double` i wynik wyrażenia będzie typu `double`.
- W wyrażeniu $a + b * c$, jeżeli a jest typu `double`, b jest typu `int`, natomiast c jest typu `unsigned int`, to
  1. Wartość b zostanie skonwertowana do typu `unsigned int`.
  2. Zostanie obliczony iloczyn $b * c$ i wynik będzie typu `unsigned int`.
  3. Wynik mnożenia otrzymany w poprzednim kroku zostanie skonwertowany do typu `double`.
  4. Zostanie obliczony wynik dodawania, który będzie typu `double`.
- W wyrażeniu $a/(b * (c + d))$, jeżeli a jest typu `double`, a pozostałe wartości są typu `int`, to:
  1. Zostanie obliczony wynik wyrażenia $b * (c + d)$, który będzie typu `int`.
  2. Wynik mnożenia otrzymany w poprzednim kroku zostanie skonwertowany do typu `double`.
  3. Zostanie obliczony wynik dzielenia, który będzie typu `double`.

Domyślne typy danych dla literałów

W czasie kompilacji literały są zastępowane wartościami ustalonych typów

- Liczba całkowita – `int`
- Liczba całkowita z dopisaną literą `U` (np. `100U`) – `unsigned int`
- Liczba całkowita z dopisaną literą `L` (np. `1000L`) – `long int`
- Liczba całkowita z „końcówką” `UL` (np. `1UL`) – `unsigned long int`
- Liczba niecałkowita – `double`
- Znak (np. `'a'`) – `char`

Zapis liczb całkowitych

1. System dziesiętny.
   - Pierwsza cyfra nie może być zerem!
2. System szesnastkowy.
   - `0x` jako prefiks.
   - Cyfry 0...9 i litery A...F (lub a...f) reprezentują wagi 0...9 i 10...15, odpowiednio.
   - Np. zapis `0x1FF` oznacza liczbę 511.
3. System ósemkowy.
   - `0` jako prefiks.
   - Cyfry 0...7 reprezentują wagi 0...7, odpowiednio.
   - Np. zapis `0111` oznacza liczbę 73.

Zapis liczb niecałkowitych

1. Notacja z kropką dziesiętną.
   - Jeśli zapis zaczyna się od kropki, to część całkowita jest równa 0 (np. `.5`).
   - Jeśli zapis kończy się kropką, to część ułamkowa jest równa 0 (np. `1.`).
2. Notacja „naukowa”.
   - Liczba w notacji z kropką, litera `E` lub `e`, liczba całkowita.
   - `E` lub `e` oznacza mnożenie przez 10 do pewnej całkowitej potęgi.
   - Liczba całkowita jest wykładnikiem.
   - Np. `1.0e3` oznacza liczbę 1000.
   - Np. `1.0e-2` oznacza liczbę 0,01.

Konwersje typów danych przy przypisaniu i modyfikacji

Zasady

1. Podczas przeprowadzania operacji przypisania (=) oraz modyfikacji (+=, −=, *=, /=, . . .) wynik wyrażenia po prawej stronie symbolu operacji jest konwertowany do typu zmiennej, której nazwa znajduje się po lewej stronie symbolu operacji.
2. Jeżeli zmienna będąca przedmiotem przypisania lub modyfikacji jest typu całkowitego (np. `int`), a wyrażenie po prawej stronie symbolu operacji jest typu niecałkowitego (np. `double`), to część ułamkowa wyniku tego wyrażenia jest odrzucana.

Konwersje typów danych przy przypisaniu i modyfikacji

Zasady

1. Podczas przeprowadzania operacji przypisania (=) oraz modyfikacji (+=, -=, *=, /=, ...) wynik wyrażenia po prawej stronie symbolu operacji jest konwertowany do typu zmiennej, której nazwa znajduje się po lewej stronie symbolu operacji.
2. Jeżeli zmienna będąca przedmiotem przypisania lub modyfikacji jest typu całkowitego (np. `int`), a wyrażenie po prawej stronie symbolu operacji jest typu niecałkowitego (np. `double`), to część ułamkowa wyniku tego wyrażenia jest odrzucana.

Przykład

- Jeżeli `n` jest typu `int`, a `r` jest typu `double`, to w wyniku przypisania `n = r;` zmienna `n` otrzyma wartość równą części całkowitej `r`.

Czym są operatory?

Operator

Symbol reprezentujący operację do przeprowadzenia.

Czym są operatory?

Operator

Symbol reprezentujący operację do przeprowadzenia.

Rodzaje operatorów w C++

- Arytmetyczne.
- Bitowe (*ang. bit pattern*).
- Porównania.
- Logiczne (*ang. boolean*).
- Operator trójargumentowy.
- Wskaźnikowe (*ang. pointer*).

Argumenty operatorów

Argumenty operatora

Wartości używane do obliczenia wyniku operacji.

Argumenty operatorów

Argumenty operatora

Wartości używane do obliczenia wyniku operacji.

Podział operatorów ze względu na liczbę argumentów

- Jednoargumentowe.
- Dwuargumentowe (większość).
- Trójargumentowe (jeden).

Operatory arytmetyczne

Jednoargumentowy operator arytmetyczny

- : zmiana znaku wartości (po prawej stronie).

Dwuargumentowe operatory arytmetyczne

- + : dodawanie.
- - : odejmowanie.
- * : mnożenie.
- / : dzielenie (jeśli argumenty są całkowite, reszta z dzielenia jest odrzucana).
- % : reszta z dzielenia.

Operatory bitowe

Argumenty operatora są traktowane jako wzory bitowe (*ang. bit pattern*).

Operatory bitowe

Argumenty operatora są traktowane jako wzory bitowe (*ang. bit pattern*).

Jednoargumentowy operator bitowy

`~` : negacja wszystkich bitów argumentu (po prawej stronie).

Np. jeśli zmienna `n` jest typu `int`, to po wykonaniu instrukcji `n = ~0;` będzie ona mieć wartość -1.

Dwuargumentowe operatory bitowe

`|` : suma bitowa (OR).

`&` : iloczyn bitowy (AND).

`^` : różnica symetryczna (XOR).

`<<` : przesunięcie bitowe w lewo.

`>>` : przesunięcie bitowe w prawo.

Operatory OR, AND i XOR

OR, suma bitowa

```
n = 12 | 10;  // 1100 | 1010 = 1110
```

Operatory OR, AND i XOR

OR, suma bitowa

```
n = 12 | 10;  // 1100 | 1010 = 1110
```

Po wykonaniu powyższych instrukcji

1. `n` ma wartość 14.

Operatory OR, AND i XOR

OR, suma bitowa

```
n = 12 | 10;  // 1100 | 1010 = 1110
```

AND, iloczyn bitowy

```
m = 12 & 10;  // 1100 & 1010 = 1000
```

Po wykonaniu powyższych instrukcji

1. `n` ma wartość 14.

Operatory OR, AND i XOR

OR, suma bitowa

```
n = 12 | 10;  // 1100 | 1010 = 1110
```

AND, iloczyn bitowy

```
m = 12 & 10;  // 1100 & 1010 = 1000
```

Po wykonaniu powyższych instrukcji

1. `n` ma wartość 14.
2. `m` ma wartość 8.

Operatory OR, AND i XOR

OR, suma bitowa

```
n = 12 | 10;  // 1100 | 1010 = 1110
```

AND, iloczyn bitowy

```
m = 12 & 10;  // 1100 & 1010 = 1000
```

XOR, różnica symetryczna

```
k = 12 ^ 10;  // 1100 ^ 1010 = 0110
```

Po wykonaniu powyższych instrukcji

1. `n` ma wartość 14.
2. `m` ma wartość 8.

Operatory OR, AND i XOR

OR, suma bitowa

```
n = 12 | 10;  // 1100 | 1010 = 1110
```

AND, iloczyn bitowy

```
m = 12 & 10;  // 1100 & 1010 = 1000
```

XOR, różnica symetryczna

```
k = 12 ^ 10;  // 1100 ^ 1010 = 0110
```

Po wykonaniu powyższych instrukcji

1. `n` ma wartość 14.
2. `m` ma wartość 8.
3. `k` ma wartość 6.

Operatory przesunięć

Przesunięcie bitowe w lewo (*ang. left shift*)

```
y = x << k;
```

1. Dla wszystkich pozycji bitowych $j = N-1, N-2, \ldots, k$ skopiuj na tę pozycję bit z pozycji $(j-k)$.
2. Na pozycje bitowe $0, 1, \ldots, k-1$ wstaw zera.

Dla typów bezznakowych odpowiada to mnożeniu przez 2^k.

Operatory przesunięć

Przesunięcie bitowe w lewo (*ang. left shift*)

```
y = x << k;
```

1. Dla wszystkich pozycji bitowych $j = N-1, N-2, \ldots, k$ skopiuj na tę pozycję bit z pozycji $(j-k)$.
2. Na pozycje bitowe $0, 1, \ldots, k-1$ wstaw zera.

Dla typów bezznakowych odpowiada to mnożeniu przez 2^k.

Przesunięcie bitowe w prawo (*ang. right shift*)

```
y = x >> k;
```

1. Dla wszystkich pozycji bitowych $j = 0, 1, \ldots, N-k-1$ skopiuj na tę pozycję bit z pozycji $(j+k)$.
2. Na pozycje bitowe $N-1, N-2, \ldots, N-k$ wstaw zera.

Dla typów bezznakowych odpowiada to dzieleniu przez 2^k bez reszty.

Operatory porównania

`==` : test równości argumentów.

`!=` : test nierówności argumentów.

`<` : mniejsze.

`<=` : mniejsze lub równe.

`>` : większe.

`>=` : większe lub równe.

Operatory porównania

`==` : test równości argumentów.

`!=` : test nierówności argumentów.

`<` : mniejsze.

`<=` : mniejsze lub równe.

`>` : większe.

`>=` : większe lub równe.

Dla wszystkich operatorów porównania wartości argumentów operacji są sprowadzane do jednakowego typu danych (zgodnie z zasadami przedstawionymi wyżej) przed dokonaniem porównania.

Operatory porównania

`==` : test równości argumentów.

`!=` : test nierówności argumentów.

`<` : mniejsze.

`<=` : mniejsze lub równe.

`>` : większe.

`>=` : większe lub równe.

Dla wszystkich operatorów porównania wartości argumentów operacji są sprowadzane do jednakowego typu danych (zgodnie z zasadami przedstawionymi wyżej) przed dokonaniem porównania.

Dla operatorów porównania wynik operacji jest zawsze typu `bool`.

Operatory logiczne

Argumenty operatora są traktowane jako wartości typu `bool` (a zatem wynik operacji też jest tego typu).

Operatory logiczne

Argumenty operatora są traktowane jako wartości typu `bool` (a zatem wynik operacji też jest tego typu).

Jednoargumentowy operator logiczny

`!` : zaprzeczenie.

Dwuargumentowe operatory logiczne (spójniki logiczne)

`||` : alternatywa.

`&&` : koniunkcja.

Operatory logiczne

Argumenty operatora są traktowane jako wartości typu `bool` (a zatem wynik operacji też jest tego typu).

Jednoargumentowy operator logiczny

`!` : zaprzeczenie.

Dwuargumentowe operatory logiczne (spójniki logiczne)

`||` : alternatywa.

`&&` : koniunkcja.

Przykład

Wyrażenie `!(3 + 5) || (4 - 2)` ma wartość `true`, a wyrażenie `!(3 + 5) && (4 - 2)` ma wartość `false`.

Operator trójargumentowy

`a ? b : c`

1. a, b i c są dowolnymi wyrażeniami.
2. Jeżeli a ma wartość 0 lub `false`, wynikiem operacji jest c.
3. W przeciwnym wypadku wynikiem operacji jest b.

Operator trójargumentowy

`a ? b : c`

1. a, b i c są dowolnymi wyrażeniami.
2. Jeżeli a ma wartość 0 lub `false`, wynikiem operacji jest c.
3. W przeciwnym wypadku wynikiem operacji jest b.

Przykład

Aby przypisać zmiennej z większą z wartości x oraz y, można użyć zapisu:

```
z = x > y ? x : y;
```

Priorytety operatorów

Priorytet (*ang. priority*) operatora

Waga określająca w jakiej kolejności w stosunku do innych operacji w wyrażeniu będzie przeprowadzona operacja reprezentowana przez dany operator.

Priorytety operatorów

Priorytet (*ang. priority*) operatora

Waga określająca w jakiej kolejności w stosunku do innych operacji w wyrażeniu będzie przeprowadzona operacja reprezentowana przez dany operator.

Zasady

1. W pierwszej kolejności przeprowadzane są operacje reprezentowane przez operatory o najwyższych priorytetach.
2. Dla operatorów o jednakowych priorytetach kolejność przeprowadzania operacji nie jest jednoznacznie określona (najlepiej jest używać nawiasów w celu określenia jej).

Tabela priorytetów (siły wiązania) operatorów

Siła wiązania (priorytet) operatorów w porządku malejącym

```
 !     ~     -
 *     /     %
 +     -
<<    >>
 <     <=    >=    >
==    !=
 &
 ^
 |
&&
||
? :
```