Programowanie w języku C++

Adam Szmagliński, p. F215
http://fizyka.szmaglinski.eu

Instytut Fizyki PK

Kraków, 18.12.2019

Funkcja **main**

Wykonywanie programu w języku C++ zaczyna się od funkcji **main**. Ciało tej funkcji zawierające instrukcje objęte jest nawiasami klamrowymi { oraz }.

```
int main(){
    cout << "Hello World!" << '\n';
    system("pause");
    return 0;
}
```

Wstęp

Funkcja **main**

Wykonywanie programu w języku C++ zaczyna się od funkcji **main**.
Ciało tej funkcji zawierające instrukcje objęte jest nawiasami klamrowymi { oraz }.

```
int main(){
    cout << "Hello World!" << '\n';
    system("pause");
    return 0;
}
```

Nazwy

Nazwy tworzy się z liter i cyfr – pierwszym znakiem musi być litera.
Znak podkreślenia „_” jest traktowany jak litera.
W języku C++ rozróżnia się wielkie i małe litery.

Format zapisu programu

Język C++ jest językiem o wolnym formacie. Kilka instrukcji można wypisywać obok siebie lub jedną instrukcję można wypisać w kilku liniach. Koniec instrukcji rozpoznajemy po średniku na jej końcu.
Białe znaki wewnątrz instrukcji są prawie zawsze ignorowane. Program ich nie potrzebuje. Używamy ich, w celu ułatwienia czytania oraz szukania ewentualnych błędów przy pomocy *debugera*.

Format zapisu programu

Język C++ jest językiem o wolnym formacie. Kilka instrukcji można wypisywać obok siebie lub jedną instrukcję można wypisać w kilku liniach. Koniec instrukcji rozpoznajemy po średniku na jej końcu.
Białe znaki wewnątrz instrukcji są prawie zawsze ignorowane. Program ich nie potrzebuje. Używamy ich, w celu ułatwienia czytania oraz szukania ewentualnych błędów przy pomocy *debugera*.

Komentarze

Czas przeznaczony na pisanie komentarzy nie jest czasem straconym. Komentarze są ignorowane przez kompilator. Można je umieszczać na dwa sposoby:

Format zapisu programu

Język C++ jest językiem o wolnym formacie. Kilka instrukcji można wypisywać obok siebie lub jedną instrukcję można wypisać w kilku liniach. Koniec instrukcji rozpoznajemy po średniku na jej końcu.
Białe znaki wewnątrz instrukcji są prawie zawsze ignorowane. Program ich nie potrzebuje. Używamy ich, w celu ułatwienia czytania oraz szukania ewentualnych błędów przy pomocy *debugera*.

Komentarze

Czas przeznaczony na pisanie komentarzy nie jest czasem straconym. Komentarze są ignorowane przez kompilator. Można je umieszczać na dwa sposoby:

- pomiędzy sekwencjami znaków /* oraz */ – nie mogą one być zagnieżdżone

Format zapisu programu

Język C++ jest językiem o wolnym formacie. Kilka instrukcji można wypisywać obok siebie lub jedną instrukcję można wypisać w kilku liniach. Koniec instrukcji rozpoznajemy po średniku na jej końcu.
Białe znaki wewnątrz instrukcji są prawie zawsze ignorowane. Program ich nie potrzebuje. Używamy ich, w celu ułatwienia czytania oraz szukania ewentualnych błędów przy pomocy *debugera*.

Komentarze

Czas przeznaczony na pisanie komentarzy nie jest czasem straconym. Komentarze są ignorowane przez kompilator. Można je umieszczać na dwa sposoby:

- pomiędzy sekwencjami znaków /* oraz */ – nie mogą one być zagnieżdżone
- na prawo od // do końca linii

Środowisko programistyczne

Edytor

Tekst programu zapisany w edytowanym pliku, powinien zawierać tylko wpisane przez nas znaki, bez znaków sterujących formatem. Plik źródłowy programu napisanego w języku C++ powinien posiadać nazwę o rozszerzeniu cpp, a w języku C, nazwę o rozszerzeniu c.

Środowisko programistyczne

Edytor

Tekst programu zapisany w edytowanym pliku, powinien zawierać tylko wpisane przez nas znaki, bez znaków sterujących formatem. Plik źródłowy programu napisanego w języku C++ powinien posiadać nazwę o rozszerzeniu cpp, a w języku C, nazwę o rozszerzeniu c.

Kompilator

Kod źródłowy programu napisanego w języku C++, za pomocą **kompilatora C++** tłumaczony jest na język procesora. Inaczej mówiąc, poddając nasz plik kompilacji, otrzymujemy kod naszego programu przetłumaczony na język maszynowy.

Linker

Skompilowana wersja programu, przy pomocy **linkera** musi zostać połączona z bibliotekami, zawierającymi dodatkowe instrukcje. Dyrektywa *#include* zapoznaje kompilator jedynie z samym nagłówkiem biblioteki, zawierającym deklaracje. Taki nagłówek wykorzystany jest do weryfikowania poprawności korzystania z biblioteki. Po zlinkowaniu program nadaje się do uruchomienia.

Instrukcje sterujące

Instrukcje w języku C++ zakończone są średnikiem. Nawiasy klamrowe { i } są używane do grupowania deklaracji i instrukcji w blok, składniowo równoważny jednej instrukcji. Średnik nie występuje po nawiasie klamrowym zamykającym blok. Sam średnik oznacza tzw. pustą instrukcję.

Instrukcje sterujące

Instrukcje w języku C++ zakończone są średnikiem. Nawiasy klamrowe { i } są używane do grupowania deklaracji i instrukcji w blok, składniowo równoważny jednej instrukcji. Średnik nie występuje po nawiasie klamrowym zamykającym blok. Sam średnik oznacza tzw. pustą instrukcję.

Operator przecinek „ , ”

Kilka wyrażeń oddzielonych przecinkiem (w roli operatora) oblicza się od lewej strony do prawej. Typem i wartością wyniku jest typ i wartość prawego argumentu (po ostatnim przecinku). Przecinki oddzielające argumenty funkcji, zmienne w deklaracjach, itp. nie są operatorami i nie gwarantują obliczeń od lewej strony do prawej.

Instrukcja **if-else**

```
if (wyrażenie) instrukcja1
else instrukcja2
```

Instrukcja **if-else**

```
if (wyrażenie) instrukcja1
else instrukcja2
```

Część **else** (druga linia) jest opcjonalna. Najpierw oblicza się *wyrażenie*. Jeśli wartością jego jest prawda (**true** – wartość różna od 0), wykonuje się `instrukcja1`. Gdy jest fałszywe (**false** – wartość 0) i istnieje część **else**, wykonuje się `instrukcja2`.
W ciągu zagnieżdżonych instrukcji **if**, każda z części **else** jest przyporządkowana najbliższej z poprzednich instrukcji **if**, nie zawierającej części **else**. W celu innego przyporządkowania stosuje się nawiasy {}.

Często występującym zastosowaniem jest konstrukcja **else-if**, jako sposób zapisania decyzji wielowariantowych.

Często występującym zastosowaniem jest konstrukcja **else-if**, jako sposób zapisania decyzji wielowariantowych.

Konstrukcja else-if

```
if (wyrażenie1) instrukcja1
else if (wyrażenie2) instrukcja2
else if (wyrażenie3) instrukcja3
else instrukcja4
```

Instrukcja **switch**

Instrukcja **switch** służy do podejmowania decyzji wielowariantowych, w których sprawdza się czy wartość pewnego wyrażenia pasuje do jednej z kilku całkowitych stałych wyrażeń i wykonuje odpowiedni skok.

Instrukcja **switch**

Instrukcja **switch** służy do podejmowania decyzji wielowariantowych, w których sprawdza się czy wartość pewnego wyrażenia pasuje do jednej z kilku całkowitych stałych wyrażeń i wykonuje odpowiedni skok.

```
switch (wyrażenie){
   case wyrażenie-stałe: instrukcje
   case wyrażenie-stałe: instrukcje
   default: instrukcje
}
```

Z każdym wariantem instrukcji związana jest jedna lub kilka całkowitych wartości bądź wyrażeń stałych. Gdy dany przypadek (**case**) jest zgodny z wartością wyrażenia, to od niego rozpoczyna się dalsze wykonywanie programu. Przypadek **default** jest opcjonalny i może występować w dowolnej kolejności z innymi. Zostaje wykonany gdy żadne wyrażenie stałe nie jest zgodne z wartością wyrażenia będącego argumentem **switch**. Instrukcja **break** powoduje natychmiastowe wyjście z instrukcji **switch**, a także pętli **while**, **for** i **do**.

Z każdym wariantem instrukcji związana jest jedna lub kilka całkowitych wartości bądź wyrażeń stałych. Gdy dany przypadek (**case**) jest zgodny z wartością wyrażenia, to od niego rozpoczyna się dalsze wykonywanie programu. Przypadek **default** jest opcjonalny i może występować w dowolnej kolejności z innymi. Zostaje wykonany gdy żadne wyrażenie stałe nie jest zgodne z wartością wyrażenia będącego argumentem **switch**. Instrukcja **break** powoduje natychmiastowe wyjście z instrukcji **switch**, a także pętli **while**, **for** i **do**.

switch czy if-else?

Do wielowariantowych wyborów lepsza jest instrukcja **switch**, poza przypadkami gdy jest to niemożliwe:

Z każdym wariantem instrukcji związana jest jedna lub kilka całkowitych wartości bądź wyrażeń stałych. Gdy dany przypadek (**case**) jest zgodny z wartością wyrażenia, to od niego rozpoczyna się dalsze wykonywanie programu. Przypadek **default** jest opcjonalny i może występować w dowolnej kolejności z innymi. Zostaje wykonany gdy żadne wyrażenie stałe nie jest zgodne z wartością wyrażenia będącego argumentem **switch**. Instrukcja **break** powoduje natychmiastowe wyjście z instrukcji **switch**, a także pętli **while**, **for** i **do**.

switch czy if-else?

Do wielowariantowych wyborów lepsza jest instrukcja **switch**, poza przypadkami gdy jest to niemożliwe:

- obiektu porównywanego nie można zmienić na typ całkowity

Z każdym wariantem instrukcji związana jest jedna lub kilka całkowitych wartości bądź wyrażeń stałych. Gdy dany przypadek (**case**) jest zgodny z wartością wyrażenia, to od niego rozpoczyna się dalsze wykonywanie programu. Przypadek **default** jest opcjonalny i może występować w dowolnej kolejności z innymi. Zostaje wykonany gdy żadne wyrażenie stałe nie jest zgodne z wartością wyrażenia będącego argumentem **switch**. Instrukcja **break** powoduje natychmiastowe wyjście z instrukcji **switch**, a także pętli **while**, **for** i **do**.

switch czy if-else?

Do wielowariantowych wyborów lepsza jest instrukcja **switch**, poza przypadkami gdy jest to niemożliwe:

- obiektu porównywanego nie można zmienić na typ całkowity
- operacja porównania nie sprawdza równości dwóch elementów

Z każdym wariantem instrukcji związana jest jedna lub kilka całkowitych wartości bądź wyrażeń stałych. Gdy dany przypadek (**case**) jest zgodny z wartością wyrażenia, to od niego rozpoczyna się dalsze wykonywanie programu. Przypadek **default** jest opcjonalny i może występować w dowolnej kolejności z innymi. Zostaje wykonany gdy żadne wyrażenie stałe nie jest zgodne z wartością wyrażenia będącego argumentem **switch**. Instrukcja **break** powoduje natychmiastowe wyjście z instrukcji **switch**, a także pętli **while**, **for** i **do**.

switch czy if-else?

Do wielowariantowych wyborów lepsza jest instrukcja **switch**, poza przypadkami gdy jest to niemożliwe:

- obiektu porównywanego nie można zmienić na typ całkowity
- operacja porównania nie sprawdza równości dwóch elementów
- wyrażenia z którymi porównujemy, nie są stałe w momencie kompilacji

Pętle **while** i **do-while**

while

W pętli:

```
while (wyrażenie) instrukcja
```

na początku obliczane jest *wyrażenie*. Gdy jest ono różne od zera wykonywana jest instrukcja i cykl się powtarza. Jeśli jest równe zeru, sterowanie wychodzi z pętli.

Pętle **while** i **do-while**

while

W pętli:

```
while (wyrażenie) instrukcja
```

na początku obliczane jest *wyrażenie*. Gdy jest ono różne od zera wykonywana jest instrukcja i cykl się powtarza. Jeśli jest równe zeru, sterowanie wychodzi z pętli.

do-while

```
do
   instrukcja
while (wyrażenie);
```

Najpierw wykonywana jest instrukcja, następnie obliczane jest *wyrażenie*. Reszta jak w **while**.

Pętla for

```
for (wyr1; wyr2; wyr3) instrukcja
```

jest równoważna rozwinięciu:

```
wyr1
while (wyr2){
    instrukcja
    wyr3
}
```

Pętla for

```
for (wyr1; wyr2; wyr3) instrukcja
```

jest równoważna rozwinięciu:

```
wyr1
while (wyr2){
   instrukcja
   wyr3
}
```

Wszystkie wyrażenia pętli **for** są opcjonalne i dowolne – średniki muszą pozostać. *wyr1* i *wyr3* najczęściej stanowią przypisania lub wywołania funkcji, a *wyr2* wyrażenie warunkowe – jeśli go brak, przyjmuje się je za prawdziwe.

Instrukcja **goto** i etykiety

Instrukcję **goto** stosuje się najczęściej do jednoczesnego przerwania działania kilku pętli. Po niej należy wpisać nazwę *etykiety*, do której ma zostać przekazane sterowanie, następnie średnik. *Etykieta* ma postać nazwy zmiennej i zakończona jest dwukropkiem. Można nią opatrzyć każdą instrukcję w funkcji zawierającej **goto**.

Instrukcja **goto** i etykiety

Instrukcję **goto** stosuje się najczęściej do jednoczesnego przerwania działania kilku pętli. Po niej należy wpisać nazwę *etykiety*, do której ma zostać przekazane sterowanie, następnie średnik. *Etykieta* ma postać nazwy zmiennej i zakończona jest dwukropkiem. Można nią opatrzyć każdą instrukcję w funkcji zawierającej **goto**.

```
   for (;;){
      int i=0;
      while (true)
         if (++i>5) goto koniec;
         else cout << "i = " << i << '\n';
   }
koniec:
   system("pause");
```

Instrukcja **continue**

Instrukcja **continue** powoduje przerwanie bieżącego i wykonanie od początku następnego kroku, zawierającej ją pętli **for**, **while** lub **do-while**. W pętli **while** i **do-while** przechodzi natychmiast do sprawdzenia warunku zatrzymania, w **for** przechodzi do części przyrostowej.

Typy

Typ obiektu określa zbiór jego wartości i operacje, jakie można na nim wykonać.

Typy

Typ obiektu określa zbiór jego wartości i operacje, jakie można na nim wykonać.

Definicja i deklaracja

Definicja spełnia funkcję deklaracji i dodatkowo rezerwuje miejsce w pamięci, powołując obiekt do istnienia np.

```
int a;
```

Deklaracja powiadamia kompilator o typie obiektu reprezentowanego przez daną nazwę np.

```
extern int a;
```

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Podział ze względu na konstrukcję:

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Podział ze względu na konstrukcję:

- Fundamentalne (podstawowe)

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Podział ze względu na konstrukcję:

- Fundamentalne (podstawowe)
- Złożone (wykorzystują typy fundamentalne)

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Podział ze względu na konstrukcję:

- Fundamentalne (podstawowe)
- Złożone (wykorzystują typy fundamentalne)

Podział ze względu na pochodzenie:

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Podział ze względu na konstrukcję:

- Fundamentalne (podstawowe)
- Złożone (wykorzystują typy fundamentalne)

Podział ze względu na pochodzenie:

- Wbudowane

Zmienne w programie powinny zostać zadeklarowane przed użyciem. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych, oddzielonych przecinkami. W deklaracjach można nadawać zmiennym wartości początkowe.

Systematyka typów

Podział ze względu na konstrukcję:

- Fundamentalne (podstawowe)
- Złożone (wykorzystują typy fundamentalne)

Podział ze względu na pochodzenie:

- Wbudowane
- Zdefiniowane przez użytkownika

Typy fundamentalne

Typy związane z liczbami całkowitymi

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity
- **long int** lub **long** – co najmniej 4 bajty

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity
- **long int** lub **long** – co najmniej 4 bajty
- **long long int** lub **long long**

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity
- **long int** lub **long** – co najmniej 4 bajty
- **long long int** lub **long long**
- **enum** typ wyliczeniowy

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity
- **long int** lub **long** – co najmniej 4 bajty
- **long long int** lub **long long**
- **enum** typ wyliczeniowy

Specyfikatory łączone z typami dla liczb całkowitych int

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity
- **long int** lub **long** – co najmniej 4 bajty
- **long long int** lub **long long**
- **enum** typ wyliczeniowy

Specyfikatory łączone z typami dla liczb całkowitych int

- **signed** – domniemany

Typy fundamentalne

Typy związane z liczbami całkowitymi

- **short int** lub **short** – zajmuje co najmniej 2 bajty
- **int** całkowity
- **long int** lub **long** – co najmniej 4 bajty
- **long long int** lub **long long**
- **enum** typ wyliczeniowy

Specyfikatory łączone z typami dla liczb całkowitych int

- **signed** – domniemany
- **unsigned**

Typy związane ze znakami alfanumerycznymi

Typy związane ze znakami alfanumerycznymi

- **char** 1 bajt (8 bitów), mieszczący jeden znak z lokalnego, podstawowego zbioru znaków – w zależności od kompilatora odpowiada jednemu z kolejnych dwóch

Typy związane ze znakami alfanumerycznymi

- **char** 1 bajt (8 bitów), mieszczący jeden znak z lokalnego, podstawowego zbioru znaków – w zależności od kompilatora odpowiada jednemu z kolejnych dwóch
- **signed char** – od 128 do 127

Typy związane ze znakami alfanumerycznymi

- **char** 1 bajt (8 bitów), mieszczący jeden znak z lokalnego, podstawowego zbioru znaków – w zależności od kompilatora odpowiada jednemu z kolejnych dwóch
- **signed char** – od 128 do 127
- **unsigned char** – od 0 do 255

Typy związane ze znakami alfanumerycznymi

- **char** 1 bajt (8 bitów), mieszczący jeden znak z lokalnego, podstawowego zbioru znaków – w zależności od kompilatora odpowiada jednemu z kolejnych dwóch
- **signed char** – od 128 do 127
- **unsigned char** – od 0 do 255
- **wchar_t** rozszerzony zestaw znaków alfanumerycznych (można w nim umieszczać znaki z zestawu UNICODE)

Typy związane ze znakami alfanumerycznymi

- **char** 1 bajt (8 bitów), mieszczący jeden znak z lokalnego, podstawowego zbioru znaków – w zależności od kompilatora odpowiada jednemu z kolejnych dwóch
- **signed char** – od 128 do 127
- **unsigned char** – od 0 do 255
- **wchar_t** rozszerzony zestaw znaków alfanumerycznych (można w nim umieszczać znaki z zestawu UNICODE)
- **char16_t** do kodowania znaków Unicode na 16-tu bitach (UTF-16)

Typy związane ze znakami alfanumerycznymi

- **char** 1 bajt (8 bitów), mieszczący jeden znak z lokalnego, podstawowego zbioru znaków – w zależności od kompilatora odpowiada jednemu z kolejnych dwóch
- **signed char** – od 128 do 127
- **unsigned char** – od 0 do 255
- **wchar_t** rozszerzony zestaw znaków alfanumerycznych (można w nim umieszczać znaki z zestawu UNICODE)
- **char16_t** do kodowania znaków Unicode na 16-tu bitach (UTF-16)
- **char32_t** do kodowania znaków Unicode na 32-óch bitach (UTF-32)

Typy reprezentujące liczby zmiennoprzecinkowe

Typy reprezentujące liczby zmiennoprzecinkowe

- **float**

Typy reprezentujące liczby zmiennoprzecinkowe

- **float**
- **double** podwójna dokładność

Typy reprezentujące liczby zmiennoprzecinkowe

- **float**
- **double** podwójna dokładność
- **long double**

Typy reprezentujące liczby zmiennoprzecinkowe

- **float**
- **double** podwójna dokładność
- **long double**

Typ reprezentujący obiekty logiczne

bool przyjmuje wartości: *true* (1), *false* (0)

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

- **int8_t**, **int16_t**, **int32_t**, **int64_t** (dla komputerów przynajmniej 64-bitowych) – całkowite ze znakiem o szerokościach: 8, 16, 32, 64 bitów.

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

- **int8_t**, **int16_t**, **int32_t**, **int64_t** (dla komputerów przynajmniej 64-bitowych) – całkowite ze znakiem o szerokościach: 8, 16, 32, 64 bitów.
- **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t** – całkowite bez znaku. Gwarantują szerokość tylu bitów, ile jest w nazwie.

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

- **int8_t**, **int16_t**, **int32_t**, **int64_t** (dla komputerów przynajmniej 64-bitowych) – całkowite ze znakiem o szerokościach: 8, 16, 32, 64 bitów.
- **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t** – całkowite bez znaku. Gwarantują szerokość tylu bitów, ile jest w nazwie.
- **int_least8_t**, **uint_least8_t**, **int_least16_t**, **uint_least16_t**, **int_least32_t**, **uint_least32_t**, **int_least64_t**, **uint_least64_t** – najmniejszy z typów całkowitych, zdolny pomieścić daną o podanej liczbie bitów.

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

- **int8_t**, **int16_t**, **int32_t**, **int64_t** (dla komputerów przynajmniej 64-bitowych) – całkowite ze znakiem o szerokościach: 8, 16, 32, 64 bitów.
- **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t** – całkowite bez znaku. Gwarantują szerokość tylu bitów, ile jest w nazwie.
- **int_least8_t**, **uint_least8_t**, **int_least16_t**, **uint_least16_t**, **int_least32_t**, **uint_least32_t**, **int_least64_t**, **uint_least64_t** – najmniejszy z typów całkowitych, zdolny pomieścić daną o podanej liczbie bitów.
- **int_fast8_t**, **uint_fast8_t**, **int_fast16_t**, **uint_fast16_t**, **int_fast32_t**, **uint_fast32_t**, **int_fast64_t**, **uint_fast64_t** – najbardziej wydajny z typów, o nie mniejszej liczbie bitów niż podana.

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

- **int8_t**, **int16_t**, **int32_t**, **int64_t** (dla komputerów przynajmniej 64-bitowych) – całkowite ze znakiem o szerokościach: 8, 16, 32, 64 bitów.
- **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t** – całkowite bez znaku. Gwarantują szerokość tylu bitów, ile jest w nazwie.
- **int_least8_t**, **uint_least8_t**, **int_least16_t**, **uint_least16_t**, **int_least32_t**, **uint_least32_t**, **int_least64_t**, **uint_least64_t** – najmniejszy z typów całkowitych, zdolny pomieścić daną o podanej liczbie bitów.
- **int_fast8_t**, **uint_fast8_t**, **int_fast16_t**, **uint_fast16_t**, **int_fast32_t**, **uint_fast32_t**, **int_fast64_t**, **uint_fast64_t** – najbardziej wydajny z typów, o nie mniejszej liczbie bitów niż podana.
- **intmax_t**, **uintmax_t** – największa ilość bitów na komputerze.

Typy całkowite z oznaczoną ilością bitów

umieszczone w pliku nagłówkowym <cstdint> i w przestrzeni nazw *std*.

- **int8_t**, **int16_t**, **int32_t**, **int64_t** (dla komputerów przynajmniej 64-bitowych) – całkowite ze znakiem o szerokościach: 8, 16, 32, 64 bitów.
- **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t** – całkowite bez znaku. Gwarantują szerokość tylu bitów, ile jest w nazwie.
- **int_least8_t**, **uint_least8_t**, **int_least16_t**, **uint_least16_t**, **int_least32_t**, **uint_least32_t**, **int_least64_t**, **uint_least64_t** – najmniejszy z typów całkowitych, zdolny pomieścić daną o podanej liczbie bitów.
- **int_fast8_t**, **uint_fast8_t**, **int_fast16_t**, **uint_fast16_t**, **int_fast32_t**, **uint_fast32_t**, **int_fast64_t**, **uint_fast64_t** – najbardziej wydajny z typów, o nie mniejszej liczbie bitów niż podana.
- **intmax_t**, **uintmax_t** – największa ilość bitów na komputerze.
- **intptr_t**, **uintptr_t** – pomieści dowolny adres pamięci komputera.

Przedział liczb możliwych do przechowywania w obiektach różnych typów fundamentalnych zależy od typu komputera i kompilatora.
Różne typy dają możliwość wyboru pomiędzy dokładnością a szybkością i użyciem pamięci.

Przedział liczb możliwych do przechowywania w obiektach różnych typów fundamentalnych zależy od typu komputera i kompilatora.
Różne typy dają możliwość wyboru pomiędzy dokładnością a szybkością i użyciem pamięci.

Miejsca definiowania zmiennych

W języku C++ zmienne można definiować w biegu, tzn. między kolejnymi instrukcjami.
Można również definiować w części inicjalizacyjnej pętli **for**, w części warunkowej pętli **while** oraz warunku instrukcji **if** – zmienne te mają zasięg ograniczony do zakresu tych instrukcji.

Inicjalizacja i przypisanie

Inicjalizacja to nadanie obiektowi wartości początkowej w chwili jego utworzenia.
Przypisanie to nadanie obiektowi wartości w późniejszej instrukcji, po jego utworzeniu.

Inicjalizacja i przypisanie

Inicjalizacja to nadanie obiektowi wartości początkowej w chwili jego utworzenia.
Przypisanie to nadanie obiektowi wartości w późniejszej instrukcji, po jego utworzeniu.

Sposoby inicjalizacji obiektów (ostatnie 2 sygnalizują o błędzie w przypadku niezgodności typów):

Inicjalizacja i przypisanie

Inicjalizacja to nadanie obiektowi wartości początkowej w chwili jego utworzenia.
Przypisanie to nadanie obiektowi wartości w późniejszej instrukcji, po jego utworzeniu.

Sposoby inicjalizacji obiektów (ostatnie 2 sygnalizują o błędzie w przypadku niezgodności typów):

- *typ* **zmienna** = *wartość*; – np. int a = 4;

Inicjalizacja i przypisanie

Inicjalizacja to nadanie obiektowi wartości początkowej w chwili jego utworzenia.
Przypisanie to nadanie obiektowi wartości w późniejszej instrukcji, po jego utworzeniu.

Sposoby inicjalizacji obiektów (ostatnie 2 sygnalizują o błędzie w przypadku niezgodności typów):

- *typ* **zmienna** = *wartość*; – np. int a = 4;
- *typ* **zmienna** (*wartość*); – np. int a (4);

Inicjalizacja i przypisanie

Inicjalizacja to nadanie obiektowi wartości początkowej w chwili jego utworzenia.
Przypisanie to nadanie obiektowi wartości w późniejszej instrukcji, po jego utworzeniu.

Sposoby inicjalizacji obiektów (ostatnie 2 sygnalizują o błędzie w przypadku niezgodności typów):

- *typ* **zmienna** = *wartość*; – np. int a = 4;
- *typ* **zmienna** (*wartość*); – np. int a (4);
- *typ* **zmienna** {*wartość*}; – np. int a {4};

Inicjalizacja i przypisanie

Inicjalizacja to nadanie obiektowi wartości początkowej w chwili jego utworzenia.
Przypisanie to nadanie obiektowi wartości w późniejszej instrukcji, po jego utworzeniu.

Sposoby inicjalizacji obiektów (ostatnie 2 sygnalizują o błędzie w przypadku niezgodności typów):

- *typ* **zmienna** = *wartość*; – np. int a = 4;
- *typ* **zmienna** (*wartość*); – np. int a (4);
- *typ* **zmienna** {*wartość*}; – np. int a {4};
- *typ* **zmienna** = {*wartość*}; – np. int a = {4};

Stałe dosłowne

Stałe będące liczbami całkowitymi

Stałe dosłowne

Stałe będące liczbami całkowitymi

- Stała całkowita taka jak 1234, jest obiektem typu **int**.

Stałe dosłowne

Stałe będące liczbami całkowitymi

- Stała całkowita taka jak 1234, jest obiektem typu **int**.
- W stałej typu **long** na jej końcu występuje mała lub wielka litera L, np. 12345678L. Jeśli stała całkowita nie mieści się w zakresie typu **int**, jest traktowana jak stała typu **long**.

Stałe dosłowne

Stałe będące liczbami całkowitymi

- Stała całkowita taka jak 1234, jest obiektem typu **int**.
- W stałej typu **long** na jej końcu występuje mała lub wielka litera L, np. 12345678L. Jeśli stała całkowita nie mieści się w zakresie typu **int**, jest traktowana jak stała typu **long**.
- Na końcu stałej typu **long long** występują dwie małe lub wielkie litery L, np. 12LL.

Stałe dosłowne

Stałe będące liczbami całkowitymi

- Stała całkowita taka jak 1234, jest obiektem typu **int**.
- W stałej typu **long** na jej końcu występuje mała lub wielka litera L, np. 12345678L. Jeśli stała całkowita nie mieści się w zakresie typu **int**, jest traktowana jak stała typu **long**.
- Na końcu stałej typu **long long** występują dwie małe lub wielkie litery L, np. 12LL.
- W stałych typu **unsigned** na końcu występuje mała lub wielka litera U.

Stałe dosłowne

Stałe będące liczbami całkowitymi

- Stała całkowita taka jak 1234, jest obiektem typu **int**.
- W stałej typu **long** na jej końcu występuje mała lub wielka litera L, np. 12345678L. Jeśli stała całkowita nie mieści się w zakresie typu **int**, jest traktowana jak stała typu **long**.
- Na końcu stałej typu **long long** występują dwie małe lub wielkie litery L, np. 12LL.
- W stałych typu **unsigned** na końcu występuje mała lub wielka litera U.
- końcówka u lub U może łączyć się w dowolnej kolejności z końcówkami typów **long** oraz **long long**, np. 54uL czy 43LLU.

Stałe dosłowne

Stałe będące liczbami całkowitymi

- Stała całkowita taka jak 1234, jest obiektem typu **int**.
- W stałej typu **long** na jej końcu występuje mała lub wielka litera L, np. 12345678L. Jeśli stała całkowita nie mieści się w zakresie typu **int**, jest traktowana jak stała typu **long**.
- Na końcu stałej typu **long long** występują dwie małe lub wielkie litery L, np. 12LL.
- W stałych typu **unsigned** na końcu występuje mała lub wielka litera U.
- końcówka u lub U może łączyć się w dowolnej kolejności z końcówkami typów **long** oraz **long long**, np. 54uL czy 43LLU.
- Początkowe zero w stałej całkowitej oznacza liczbę ósemkową, początkowe znaki 0x lub 0X liczbę szesnastkową, a początkowe 0b liczbę w zapisie dwójkowym.

Stałe reprezentujące liczby zmiennoprzecinkowe

zawierają kropkę dziesiętną lub zapisywane są w notacji wykładniczej, czyli po liczbie wypisywana jest litera e lub E, po której następuje wykładnik potęgi o podstawie 10 np. liczba 12345 to: 12345. lub 1.2345E4.

Stałe reprezentujące liczby zmiennoprzecinkowe

zawierają kropkę dziesiętną lub zapisywane są w notacji wykładniczej, czyli po liczbie wypisywana jest litera e lub E, po której następuje wykładnik potęgi o podstawie 10 np. liczba 12345 to: 12345. lub 1.2345E4.

- Takie stałe bez końcowej litery są typu **double**

Stałe reprezentujące liczby zmiennoprzecinkowe

zawierają kropkę dziesiętną lub zapisywane są w notacji wykładniczej, czyli po liczbie wypisywana jest litera e lub E, po której następuje wykładnik potęgi o podstawie 10 np. liczba 12345 to: 12345. lub 1.2345E4.

- Takie stałe bez końcowej litery są typu **double**
- Występująca na końcu litera f lub F oznacza stałą typu **float**.

Stałe reprezentujące liczby zmiennoprzecinkowe

zawierają kropkę dziesiętną lub zapisywane są w notacji wykładniczej, czyli po liczbie wypisywana jest litera e lub E, po której następuje wykładnik potęgi o podstawie 10 np. liczba 12345 to: 12345. lub 1.2345E4.

- Takie stałe bez końcowej litery są typu **double**
- Występująca na końcu litera f lub F oznacza stałą typu **float**.
- Końcowa litera L oznacza stałą typu **long double**

Stałe reprezentujące liczby zmiennoprzecinkowe

zawierają kropkę dziesiętną lub zapisywane są w notacji wykładniczej, czyli po liczbie wypisywana jest litera e lub E, po której następuje wykładnik potęgi o podstawie 10 np. liczba 12345 to: 12345. lub 1.2345E4.

- Takie stałe bez końcowej litery są typu **double**
- Występująca na końcu litera f lub F oznacza stałą typu **float**.
- Końcowa litera L oznacza stałą typu **long double**

Stała dosłowna dla wskaźników **nullptr**

jest to zerowy adres w pamięci. Jest specjalnie wymyślonego typu o nazwie **std::nullptr_t**, aby unikąć dwuznaczności poprzez mylenie z wartością 0 typu int.

Stała znakowa

to znak alfanumeryczny ujęty w apostrofy. Jednym z najbardziej popularnych sposobów kodowania znaków jest kod ASCII – jest to najbardziej podstawowy zestaw, z którego tylko początkowe 127 znaków jest objętych standardem. Można je również zapisywać bezpośrednio liczbowym kodem znaku, ujętym w apostrofy, w postaci ósemkowej lub szesnastkowej, np. *'a'* w kodzie ASCII reprezentowany przez 97, zapisujemy bez 0 przed liczbą: *'\141'* lub *'\x61'*.
Stałe takie są typu **char**. Poprzedzone literą **L** mają typ **wchar_t**, np. *L'a'*. Typy przechowujące znaki są typami całkowitymi, czyli arytmetycznymi – można na nich wykonywać operacje matematyczne.

Znaki specjalne

Znaki specjalne

- \b – backspace

Znaki specjalne

- \b – backspace
- \f – nowa strona

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy
- \a – sygnał dźwiękowy

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy
- \a – sygnał dźwiękowy
- \\ – odwrotny ukośnik

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy
- \a – sygnał dźwiękowy
- \\ – odwrotny ukośnik
- \' – apostrof

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy
- \a – sygnał dźwiękowy
- \\ – odwrotny ukośnik
- \' – apostrof
- \" – cudzysłów

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy
- \a – sygnał dźwiękowy
- \\ – odwrotny ukośnik
- \' – apostrof
- \" – cudzysłów
- \? – pytajnik

Znaki specjalne

- \b – backspace
- \f – nowa strona
- \n – nowa linia
- \r – powrót karetki
- \t – tabulator poziomy
- \v – tabulator pionowy
- \a – sygnał dźwiękowy
- \\ – odwrotny ukośnik
- \' – apostrof
- \" – cudzysłów
- \? – pytajnik
- \0 – *null*, czyli znak o kodzie 0

Stałe tekstowe

Stała napisowa lub **napis** lub **C-string** jest ciągiem złożonym ze znaków zawartych między znakami cudzysłowu. *Stała napisowa* jest tablicą znaków – ostatnim elementem tej tablicy jest dodany znak '\0' – tzw. znak *null*. C-string musi w jednej linii otworzyć się i zamknąć cudzysłowiem. Jednak sąsiadujące (oddzielone ewentualnie białymi znakami) *C-stringi* są sklejane podczas kompilacji programu.
C-string składający się z *n* znaków jest typu: *const char[n+1]*, aby zabezpieczyć przed zniszczeniem sąsiednich komórek pamięci.

Stałe tekstowe

Stała napisowa lub **napis** lub **C-string** jest ciągiem złożonym ze znaków zawartych między znakami cudzysłowu. *Stała napisowa* jest tablicą znaków – ostatnim elementem tej tablicy jest dodany znak '\0' – tzw. znak *null*. C-string musi w jednej linii otworzyć się i zamknąć cudzysłowiem. Jednak sąsiadujące (oddzielone ewentualnie białymi znakami) *C-stringi* są sklejane podczas kompilacji programu.
C-string składający się z *n* znaków jest typu: *const char[n+1]*, aby zabezpieczyć przed zniszczeniem sąsiednich komórek pamięci.

C-string złożony z tzw. szerokich znaków typu wchar_t, poprzedza się wielką literą **L**, np. wcout « L"Szerokie znaki";

Surowe stałe tekstowe

to tekst, w którym znaki specjalne wypisywane są bez dodatkowych znaków i tekst jest interpretowany tak, jak jest widoczny na ekranie.
R"(*tekst***)"** – opatrujemy je wielką literą **R**, następnie *tekst* obejmujemy sekwencją znaków: "(i)".

Surowe stałe tekstowe

to tekst, w którym znaki specjalne wypisywane są bez dodatkowych znaków i tekst jest interpretowany tak, jak jest widoczny na ekranie.
R"(*tekst***)"** – opatrujemy je wielką literą **R**, następnie *tekst* obejmujemy sekwencją znaków: **"(** i **)"**.

Jeśli w tekście miałaby pojawić się zamykająca sekwencja znaków *)"*, stosujemy umowny zapis: **R"***ogranicznik***(***tekst***)***ogranicznik***"**, w którym *ogranicznik* to dowolny ciąg znaków o długości < 32.

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu
- * – wskaźnik do pokazywania na obiekty danego typu

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu
- * – wskaźnik do pokazywania na obiekty danego typu
- () – funkcja zwracająca wartość danego typu

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu
- * – wskaźnik do pokazywania na obiekty danego typu
- () – funkcja zwracająca wartość danego typu
- & – referencja obiektu danego typu

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu
- * – wskaźnik do pokazywania na obiekty danego typu
- () – funkcja zwracająca wartość danego typu
- & – referencja obiektu danego typu

Typ **void**

Słowo *void* występuje jako typ prosty w deklaracji typu złożonego np.:

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu
- * – wskaźnik do pokazywania na obiekty danego typu
- () – funkcja zwracająca wartość danego typu
- & – referencja obiektu danego typu

Typ **void**

Słowo *void* występuje jako typ prosty w deklaracji typu złożonego np.:

- *void *p;* – *p* jest wskaźnikiem pokazującym na obiekt nieznanego typu

Typy złożone

Nazwa typu złożonego składa się z nazwy typu prostszego i operatora:

- [] – tablica obiektów danego typu
- * – wskaźnik do pokazywania na obiekty danego typu
- () – funkcja zwracająca wartość danego typu
- & – referencja obiektu danego typu

Typ **void**

Słowo *void* występuje jako typ prosty w deklaracji typu złożonego np.:

- *void *p;* – *p* jest wskaźnikiem pokazującym na obiekt nieznanego typu
- *void funkcja();* – deklaracja funkcji nie zwracającej żadnej wartości

Zakres ważności nazwy obiektu

Czas życia obiektu to okres od momentu jego definicji (przydzielenia miejsca w pamięci), do momentu gdy przestaje on istnieć (zwolnienie miejsca w pamięci). Obiekt może istnieć i nie być dostępny np. gdy znajdujemy się poza zakresem ważności jego nazwy.
Zakres ważności nazwy obiektu to część programu, w której on jest dostępny, czyli jego nazwa jest znana.
Obiekt może istnieć, lecz w danym miejscu programu być niedostępny.

Zakres lokalny

Nawiasy klamrowe { i } są używane do grupowania deklaracji i instrukcji w blok, składniowo różwnoważny jednej instrukcji.
Po nawiasie klamrowym zamykającym blok, nie występuje średnik.
Zdefiniowane w tym bloku nazwy mają zakres ważności od miejsca ich zdefiniowania, do klamry } kończącej lokalny blok.

Zakres lokalny

Nawiasy klamrowe { i } są używane do grupowania deklaracji i instrukcji w blok, składniowo różwnoważny jednej instrukcji.
Po nawiasie klamrowym zamykającym blok, nie występuje średnik.
Zdefiniowane w tym bloku nazwy mają zakres ważności od miejsca ich zdefiniowania, do klamry } kończącej lokalny blok.

Zakres instrukcji

Zakres ważności obejmujący jedną instrukcję może wystąpić gdy definiujemy zmienną np. w części inicjalizacyjnej pętli *for*, czy w części warunkowej *if* lub *while*.

Zakres bloku funkcji

Zakres ważności obiektów jak w bloku lokalnym, za wyjątkiem etykiety – jest ona znana w całej funkcji. Poza funkcją jest ona nieznana.

Zakres bloku funkcji

Zakres ważności obiektów jak w bloku lokalnym, za wyjątkiem etykiety – jest ona znana w całej funkcji. Poza funkcją jest ona nieznana.

W funkcji zmienne wewnętrzne zaczynają istnieć w chwili jej wywołania i znikają po jej zakończeniu.
Zmienne zewnętrzne istnieją stale od momentu ich zadeklarowania.

Zakres bloku funkcji

Zakres ważności obiektów jak w bloku lokalnym, za wyjątkiem etykiety – jest ona znana w całej funkcji. Poza funkcją jest ona nieznana.

W funkcji zmienne wewnętrzne zaczynają istnieć w chwili jej wywołania i znikają po jej zakończeniu.
Zmienne zewnętrzne istnieją stale od momentu ich zadeklarowania.

Zakres obszaru pliku

Nazwa zadeklarowana na zewnątrz bloku objętego nawiasami klamrowymi (w tym bloku funkcji) oraz poza przestrzenią nazw, nazywana jest **nazwą globalną**. Zakres ważności jej nazwy zaczyna się od miejsca deklaracji do końca pliku.
Zmienne te są dostępne dla wielu funkcji. Dla funkcji wymagających dostępu do dużej liczby wspólnych danych, zmienne zewnętrzne są bardziej efektywne niż długie listy argumentów.

Przestrzenie nazw

Aby uniknąć konfliktu powtórzenia nazw m.in. z różnych bibliotek, nazwy można grupować w przestrzenie nazw – namespace.

Przestrzenie nazw

Aby uniknąć konfliktu powtórzenia nazw m.in. z różnych bibliotek, nazwy można grupować w przestrzenie nazw – namespace.

Do bloku przestrzeni nazw dołącza się deklaracje nazw ujęte pomiędzy nawiasami klamrowymi { i }, poprzedzonymi słowem kluczowym *namespace* i nazwą przestrzeni nazw np.

```
namespace nazwa {int a, b; double c;}
```

Przestrzenie nazw

Aby uniknąć konfliktu powtórzenia nazw m.in. z różnych bibliotek, nazwy można grupować w przestrzenie nazw – namespace.

Do bloku przestrzeni nazw dołącza się deklaracje nazw ujęte pomiędzy nawiasami klamrowymi { i }, poprzedzonymi słowem kluczowym *namespace* i nazwą przestrzeni nazw np.

```
namespace nazwa {int a, b; double c;}
```

Można wielokrotnie dodawać deklaracje nazw obiektów globalnych, deklaracje czy definicje funkcji, definicje klasy. Można też dodać dyrektywę *#include*, wstawiającą plik nagłówkowy, zawierający wiele deklaracji.

Dostęp do nazw danej przestrzeni realizowany jest na 3 sposoby:

Dostęp do nazw danej przestrzeni realizowany jest na 3 sposoby:

- poprzedzenie nazwy, nazwą przestrzeni i operatorem zakresu *::*

Dostęp do nazw danej przestrzeni realizowany jest na 3 sposoby:

- poprzedzenie nazwy, nazwą przestrzeni i operatorem zakresu *::*
- wypisanie **dyrektywy** *using*, słowa *namespace* i nazwy przestrzeni – nazwy wymienionej przestrzeni są rozpoznawane do końca pliku

Dostęp do nazw danej przestrzeni realizowany jest na 3 sposoby:

- poprzedzenie nazwy, nazwą przestrzeni i operatorem zakresu *::*
- wypisanie **dyrektywy** *using*, słowa *namespace* i nazwy przestrzeni – nazwy wymienionej przestrzeni są rozpoznawane do końca pliku
- wypisanie **deklaracji** *using*, nazwy przestrzeni, operatora zakresu *::* i danej nazwy

Zasłanianie nazw

Zdefiniowany obiekt lokalny zasłania w jego lokalnym zakresie, obiekt globalny o tej samej nazwie. Dostęp do obiektu globalnego w tym lokalnym zakresie możliwy jest poprzez operator zakresu poprzedzający tę nazwę.

Zasłanianie nazw

Zdefiniowany obiekt lokalny zasłania w jego lokalnym zakresie, obiekt globalny o tej samej nazwie. Dostęp do obiektu globalnego w tym lokalnym zakresie możliwy jest poprzez operator zakresu poprzedzający tę nazwę.

Obiekt lokalny może zasłaniać też inny obiekt lokalny o tej samej nazwie. Operator zakresu nie umożliwia jednak dostępu do poprzedniego obiektu lokalnego.

Specyfikator **const**

Obiekty **const** można inicjalizować (również w trakcie pracy programu), lecz nie można im nic przypisać.
Słowo *const* stawia się w definicji przed nazwą typu.

Specyfikator **const**

Obiekty **const** można inicjalizować (również w trakcie pracy programu), lecz nie można im nic przypisać.
Słowo *const* stawia się w definicji przed nazwą typu.

Specyfikator **constexpr**

stosujemy dla stałych, których wartość jest znana już w momencie kompilacji. Obiekty *constexpr* można inicjalizować stałą dosłowną lub wyrażeniem składającym się z innych stałych typu *constexpr*, np.
constexpr double c_speed {299792458};
Kompilator może wykonać część operacji na wyrażeniach *constexpr* w czasie kompilacji, co przyspieszy działanie programu.
Można te obiekty wykorzystać np. w etykietach instrukcji *switch* czy w rozmiarze definicji tablic.

Specyfikator **volatile**

Słowo **volatile** poprzedzające nazwę typu w definicji obiektu, ostrzega kompilator o możliwej zmianie wartości obiektu bez wiedzy kompilatora, np. gdy jest on umieszczony w obszarze pamięci komputera, która reprezentuje układ sprzęgający z pewnym miernikiem.
Z każdym użyciem obiektu kompilator musi dotrzeć do jego miejsca w pamięci, a nie w rejestrze.

Specyfikator **volatile**

Słowo **volatile** poprzedzające nazwę typu w definicji obiektu, ostrzega kompilator o możliwej zmianie wartości obiektu bez wiedzy kompilatora, np. gdy jest on umieszczony w obszarze pamięci komputera, która reprezentuje układ sprzęgający z pewnym miernikiem.
Z każdym użyciem obiektu kompilator musi dotrzeć do jego miejsca w pamięci, a nie w rejestrze.

Obiekty **register**

Specyfikator **register** można dodać do definicji zmiennej automatycznej typu całkowitego przed nazwą typu. Kompilator zwykle uwzględnia to polecenie i zapisuje tę zmienną w rejestrze, przez co dostęp do niej jest szybszy. Jeśli zarządamy jej adresu, kompilator umieści ten obiekt w zwykłej pamięci.

Instrukcje **typedef** oraz **using**

nadają dodatkową nazwę już istniejącemu typowi fundamentalnemu, pochodnemu czy dla funkcji.

```
typedef int calk, *wskCalk;
using dlugosc = int;
```

Tutaj *calk* i *dlugosc* są synonimem typu *int*, a *wskCalk* synonimem wskaźnika do typu *int*. Z synonimów tych można korzystać w deklaracjach, rzutowaniach itp., tak samo jak z typu *int*.
Można łączyć te synonimy ze specyfikatorami opisującymi sposób postępowania, np. *const*, *volatile*.
Dobrze jest przyjąć pewną konwencję dla tworzonych nazw typów, aby odróżnić je od nazw zmiennych, np. dlugosc_t

Uproszczenie zapisu skomplikowanych typów

Poniższa deklaracja tworzy typ o nazwie pfi, jako wskaźnik do funkcji zwracającej wartość typu int o dwóch argumentach typu *char **.

```
typedef int (*pfi)(char *, char *);
```

Typ *pfi* można stosować jako deklarację prototpów funkcji: *pfi strcmp;*

Uproszczenie zapisu skomplikowanych typów

Poniższa deklaracja tworzy typ o nazwie pfi, jako wskaźnik do funkcji zwracającej wartość typu int o dwóch argumentach typu *char **.

```
typedef int (*pfi)(char *, char *);
```

Typ *pfi* można stosować jako deklarację prototpów funkcji: *pfi strcmp;*

Łatwość zmiany typów zdefiniowanych zmiennych

Jeżeli deklaracje *typedef* są stosowane dla tych typów danych, które mogą zależeć od maszyny (kompilatora), to tylko te deklaracje będą wymagać zmiany przy przenoszeniu programu.

Typy wyliczeniowe **enum**

Z typu wyliczeniowego korzystamy, gdy chcemy w obiekcie typu całkowitego przechować pewien rodzaj informacji zamiast liczby.

```
enum class dzienTyg {Pn = 1, Wt, Sr, Cz, Pt, So, N};
dzienTyg dzien = dzienTyg::Wt;
```

Zamiast słowa **class** może występować **struct**.

Typy wyliczeniowe **enum**

Z typu wyliczeniowego korzystamy, gdy chcemy w obiekcie typu całkowitego przechować pewien rodzaj informacji zamiast liczby.

```
enum class dzienTyg {Pn = 1, Wt, Sr, Cz, Pt, So, N};
dzienTyg dzien = dzienTyg::Wt;
```

Zamiast słowa **class** może występować **struct**.

Kolejne nazwy przyjmują kolejne wartości całkowite poczynając od zera, chyba że wystąpi jawnie podana wartość. Reprezentacje liczbowe nie muszą się różnić – nadajemy wtedy tej samej akcji różne nazwy. Do zmiennej danego typu wyliczeniowego mogą być przypisywane tylko elementy listy wyliczeniowej – nie mogą to być liczby.

Przez domniemanie podwaliną typu wyliczeniowego jest typ *int*. Można wybrać inny całkowity typ podwaliny (np. w celu zaoszczędzenia pamięci), dodając po nazwie typu znak : i typ całkowity, np.
enum class oceany : char {Atlantycki, Spokojny, Indyjski};

Przez domniemanie podwaliną typu wyliczeniowego jest typ *int*. Można wybrać inny całkowity typ podwaliny (np. w celu zaoszczędzenia pamięci), dodając po nazwie typu znak : i typ całkowity, np.
enum class oceany : char {Atlantycki, Spokojny, Indyjski};

Zwykłe enum

ze względów historycznych jest nadal dostępne, choć jest powodem wielu konfliktów nazw:

```
enum dzienTyg {Pn = 1, Wt, Sr, Cz, Pt, So, N};
dzienTyg dzien = Wt;
```

Przez domniemanie podwaliną typu wyliczeniowego jest typ *int*. Można wybrać inny całkowity typ podwaliny (np. w celu zaoszczędzenia pamięci), dodając po nazwie typu znak : i typ całkowity, np.
enum class oceany : char {Atlantycki, Spokojny, Indyjski};

Zwykłe enum

ze względów historycznych jest nadal dostępne, choć jest powodem wielu konfliktów nazw:

```
enum dzienTyg {Pn = 1, Wt, Sr, Cz, Pt, So, N};
dzienTyg dzien = Wt;
```

enum jest typem arytmetycznym.
Zwykły typ wyliczeniowy nie musi mieć nazwy. Wtedy można w wygodny sposób zdefiniować kilka stałych całkowitych.

auto – automatyczne rozpoznawanie typu

Podczas definiowania obiektu można użyć słowa **auto** zamiast nazwy typu. Wtedy typ definiowanego obiektu określony jest przez typ wyliczonej wartości wyrażenia inicjalizującego, np.

```
auto n = 25-14; // n jest typu int
```

Specyfikatory takie jak *const*, *volatile* trzeba dołączać dodatkowo, np.

```
const auto d = 2.5;
```

auto – automatyczne rozpoznawanie typu

Podczas definiowania obiektu można użyć słowa **auto** zamiast nazwy typu. Wtedy typ definiowanego obiektu określony jest przez typ wyliczonej wartości wyrażenia inicjalizującego, np.

```
auto n = 25-14; // n jest typu int
```

Specyfikatory takie jak *const*, *volatile* trzeba dołączać dodatkowo, np.

```
const auto d = 2.5;
```

Operator **decltype**

występuje w roli typu wyrażenia podanego jako jego argument, np.

```
decltype(zmiennaTypu_float) nowaZmienna; // deklaracja zmiennej
using nowy_typ = decltype(zmiennaTypu_int); // zapamiętanie typu
```

Operator ten ma zastosowanie w programowaniu uogólnionym.

Domniemane wartości inicjalizacji

Zmienne zdefiniowane w *zakresie globalnym* inicjalizowane są automatycznie odpowiednią dla danego typu wartością zero.
Zmienne definiowane w *zakresie lokalnym* nie są automatycznie inicjalizowane.
Aby zainicjalizować zmienną, odpowiednią dla danego typu wartością domniemaną, używamy pustej klamry inicjalizacyjnej, np.
long a {};
Wartością domniemaną dla typów wbudowanych jest odpowiedni rodzaj zera.

Specyfikator alignas

stawiamy w definicji obiektu życząc sobie, aby kompilator zbudował go pod adresem pamięci, o wartości podzielnej przez kolejne potęgi liczby 2 (począwszy od 0), np. alignas(16) long a;
Zależnie od zdolności i preferencji kompilatora ta sugestia może być spełniona lub nie.

Specyfikator **alignas**

stawiamy w definicji obiektu życząc sobie, aby kompilator zbudował go pod adresem pamięci, o wartości podzielnej przez kolejne potęgi liczby 2 (począwszy od 0), np. alignas(16) long a;
Zależnie od zdolności i preferencji kompilatora ta sugestia może być spełniona lub nie.

Elementarne komórki pamięci są ponumerowane (adresowane) kolejnymi liczbami i mieszczą w sobie zwykle 1 bajt.
Każde dwie sąsiednie komórki pamięci zgrupowane są w **słowa pamięci**.

Specyfikator **alignas**

stawiamy w definicji obiektu życząc sobie, aby kompilator zbudował go pod adresem pamięci, o wartości podzielnej przez kolejne potęgi liczby 2 (począwszy od 0), np. alignas(16) long a;
Zależnie od zdolności i preferencji kompilatora ta sugestia może być spełniona lub nie.

Elementarne komórki pamięci są ponumerowane (adresowane) kolejnymi liczbami i mieszczą w sobie zwykle 1 bajt.
Każde dwie sąsiednie komórki pamięci zgrupowane są w **słowa pamięci**.

Adresy kolejnych dwubajtowych słów pamięci będą maiły adresy równe kolejnym liczbom parzystym.
Podstawą pracy komputerów są obliczenia na słowach, a nie na bajtach – są więc tak zaprojektowane aby dostęp do słów był najszybszy. Kompilator wie, że zmienną 4-bajtową trzeba umieścić w pamięci pod adresem o wartości podzielnej przez 4, aby program wykonywał się najszybciej.

Operatory

Operatory arytmetyczne

Występują dwuargumentowe operatory arytmetyczne: „+”, „–”, „*”, „/” oraz „%” dla argumentów typu *int*.

Dzielenie liczb całkowitych „/” obcina część ułamkową, a $x\%y$ daje resztę z dzielenia x przez y.

Operatory arytmetyczne są lewostronnie łączne.

Operatory mnożenia i dzielenia mają wyższy priorytet niż dodawania i odejmowania.

Jednoargumentowy operator „+” (plus) nie wykonuje żadnego działania, jednoargumentowy „–” (minus) zamienia wartość wyrażenia na liczbę przeciwną.

Operatory zwiększania i zmniejszania

Operator zwiększania „++” dodaje *1* do zmiennej, a zmniejszania „−−” odejmuje *1*. Jeśli operatory „++” lub „−−” użyte są jako przedrostkowe (przed obiektem), zostaje on zmieniony przed jego użyciem, jeśli jako przyrostkowe, obiekt zostaje zmieniony po jego użyciu w wyrażeniu.

Operatory przypisania

W przypisaniach wartość prawej strony jest przekształcana do typu wyniku z lewej strony. Zamiana typów *double* lub *float* na *int* powoduje obcięcie części ułamkowej. Przypisanie samo w sobie ma wartość równą przypisywanej wartości. Operatory przypisania mają najniższy priorytet, nie wliczając operatora przecinkowego i operatora *throw*.

Operatory przypisania

W przypisaniach wartość prawej strony jest przekształcana do typu wyniku z lewej strony. Zamiana typów *double* lub *float* na *int* powoduje obcięcie części ułamkowej. Przypisanie samo w sobie ma wartość równą przypisywanej wartości. Operatory przypisania mają najniższy priorytet, nie wliczając operatora przecinkowego i operatora *throw*.

Wyrażenia podobne do *x = x*(a+b);* w których zmienna występująca po lewej stronie operatora przypisania „=" powtarza się natychmiast po prawej stronie, można zapisać w bardziej zwartej postaci *x *= a+b;*

Operatory przypisania

W przypisaniach wartość prawej strony jest przekształcana do typu wyniku z lewej strony. Zamiana typów *double* lub *float* na *int* powoduje obcięcie części ułamkowej. Przypisanie samo w sobie ma wartość równą przypisywanej wartości. Operatory przypisania mają najniższy priorytet, nie wliczając operatora przecinkowego i operatora *throw*.

Wyrażenia podobne do *x = x*(a+b);* w których zmienna występująca po lewej stronie operatora przypisania „=" powtarza się natychmiast po prawej stronie, można zapisać w bardziej zwartej postaci *x *= a+b;*

Dla większości dwuargumentowych operatorów istnieje analogiczny operator przypisania: „+=", „-=", „*=", „/=", „%=", „<<=", „>>=", „&=" (lub *and_eq*), „|=" (lub *or_eq*), „^=" (lub *xor_eq*).

Wyrażenie warunkowe

W wyrażeniu *wyr1 ? wyr2 : wyr3* obliczana jest wartość *wyr1*, następnie jeśli jest ona różna od zera (prawdziwa), obliczana jest wartość *wyr2* i będzie ona wartością całego wyrażenia warunkowego.
W przeciwnym przypadku wynikiem będzie *wyr3*.

```
if(a > b) z = a;
else z = b;
```

jest równoważna

```
z = a > b ? a : b;
```

Operatory logiczne

Operatory relacji

Operatory relacji: „>”, „>=”, „<”, „<=” mają wyższy priorytet niż operatory przyrównania „==”, „!=” (lub *not_eq*), a niższy niż priorytet operatorów arytmetycznych.

Operatory logiczne

Operatory relacji

Operatory relacji: „>”, „>=”, „<”, „<=” mają wyższy priorytet niż operatory przyrównania „==”, „!=” (lub *not_eq*), a niższy niż priorytet operatorów arytmetycznych.

Logiczna suma, iloczyn i negacja

Wyrażenia połączone operatorami logicznymi iloczynu && (lub *and*), koniunkcji || (lub *or*) oblicza się od lewej do prawej, do momentu określenia wyniku jako „prawda” (numeryczna wartość 1) lub „fałsz” (wartość 0). Jednoargumentowy operator negacji „!” (lub *not*) zamienia argument różny od zera na 0, a 0 na 1.

Operatory logiczne

Operatory relacji

Operatory relacji: „>”, „>=”, „<”, „<=” mają wyższy priorytet niż operatory przyrównania „==”, „!=” (lub *not_eq*), a niższy niż priorytet operatorów arytmetycznych.

Logiczna suma, iloczyn i negacja

Wyrażenia połączone operatorami logicznymi iloczynu && (lub *and*), koniunkcji || (lub *or*) oblicza się od lewej do prawej, do momentu określenia wyniku jako „prawda” (numeryczna wartość 1) lub „fałsz” (wartość 0). Jednoargumentowy operator negacji „!” (lub *not*) zamienia argument różny od zera na 0, a 0 na 1.

Np. rok jest przestępny, jeśli jest podzielny przez 4 i nie dzieli się przez 100 – z wyjątkiem lat podzielnych przez 400:

```
cout << "rok " << r << (r%4==0 && r%100 || r%400==0
? "" : " nie") << " jest przestepny";
```

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

- & (lub *bitand*) – bitowa koniunkcja

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

- & (lub *bitand*) – bitowa koniunkcja
- | (lub *bitor*) – bitowa alternatywa

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

- & (lub *bitand*) – bitowa koniunkcja
- | (lub *bitor*) – bitowa alternatywa
- ^ (lub *xor*) – bitowa różnica symetryczna

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

- & (lub *bitand*) – bitowa koniunkcja
- | (lub *bitor*) – bitowa alternatywa
- ^ (lub *xor*) – bitowa różnica symetryczna
- ~ (lub *compl*) – dopełnienie jedynkowe (operator jednoargumentowy)

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

- & (lub *bitand*) – bitowa koniunkcja
- | (lub *bitor*) – bitowa alternatywa
- ^ (lub *xor*) – bitowa różnica symetryczna
- ~ (lub *compl*) – dopełnienie jedynkowe (operator jednoargumentowy)
- << – przesunięcie w lewo

Operatory bitowe

Operatory bitowe można stosować do argumentów całkowitych, to znaczy *char*, *short*, *int* oraz *long* – ze znakiem lub bez:

- & (lub *bitand*) – bitowa koniunkcja
- | (lub *bitor*) – bitowa alternatywa
- ^ (lub *xor*) – bitowa różnica symetryczna
- ~ (lub *compl*) – dopełnienie jedynkowe (operator jednoargumentowy)
- << – przesunięcie w lewo
- >> – przesunięcie w prawo

Bitowy operator koniunkcji &

można stosować do „zasłaniania” pewnego zbioru bitów, np. *n &= 017;* zeruje wszystkie oprócz czterech najmniej znaczących bitów zmiennej *n*.

Bitowy operator koniunkcji &

można stosować do „zasłaniania” pewnego zbioru bitów, np. $n \;\&= 017;$ zeruje wszystkie oprócz czterech najmniej znaczących bitów zmiennej n.

Bitowy operator alternatywy |

używany jest do „ustawiania” bitów, np. $x \mid= SET\_ON$ ustawia jedynki w x na tych bitach, które są jedynkami w SET_ON.

Bitowy operator koniunkcji &

można stosować do „zasłaniania” pewnego zbioru bitów, np. $n \;\&= 017;$ zeruje wszystkie oprócz czterech najmniej znaczących bitów zmiennej n.

Bitowy operator alternatywy |

używany jest do „ustawiania” bitów, np. $x \mid= SET\_ON$ ustawia jedynki w x na tych bitach, które są jedynkami w SET_ON.

Operator bitowej różnicy symetrycznej ^

ustawia na każdej pozycji bitowej jedynkę tam, gdzie bity w obu argumentach są różne, a zera tam gdzie bity są jednakowe.

Bitowy operator koniunkcji &

można stosować do „zasłaniania” pewnego zbioru bitów, np. *n &= 017;* zeruje wszystkie oprócz czterech najmniej znaczących bitów zmiennej *n*.

Bitowy operator alternatywy |

używany jest do „ustawiania” bitów, np. *x |= SET_ON* ustawia jedynki w *x* na tych bitach, które są jedynkami w *SET_ON*.

Operator bitowej różnicy symetrycznej ^

ustawia na każdej pozycji bitowej jedynkę tam, gdzie bity w obu argumentach są różne, a zera tam gdzie bity są jednakowe.

Jednoargumentowy operator ~

zamienia każdy bit 1 na 0 i odwrotnie.
Np. *x &= ~07;* wyzeruje w *x* ostatnie 3 bity.

Operatory << i >>

przesuwają bity kopii argumentu stojącego po lewej stronie operatora odpowiednio w lewo lub w prawo, o liczbę pozycji określoną przez dodatni argument po prawej stronie.
Zwolnione bity wypełnia zerami np. $x >> 3;$ odpowiada podzieleniu x przez 8.
Przesuwanie bitów stosuje się zwykle dla zmiennych typu *unsigned*.
W przypadku ujemnej danej typu *signed* w niektórych komputerach operator >> powoduje uzupełnienie brakujących z lewej strony bitów jedynkami, w przypadku innych zerami.

Operator uzyskiwania adresu &

Operator & działając na dany obiekt daje jego adres w pamięci.
W adresie zawarta jest informacja o numerze komórki w pamięci, zawierającej początek obiektu oraz o typie obiektu.

Operator uzyskiwania adresu &

Operator & działając na dany obiekt daje jego adres w pamięci.
W adresie zawarta jest informacja o numerze komórki w pamięci, zawierającej początek obiektu oraz o typie obiektu.

Operator *size_t* sizeof

Jednoargumentowy operator **sizeof** daje w wyniku wartość całkowitą (w bajtach), równą rozmiarowi wskazanego obiektu lub typu.
Obiektem może być zmienna, tablica lub struktura. Nazwą typu może być nazwa jednego z typów podstawowych, albo nazwa typu pochodnego jak struktura czy wskaźnik np.:

```
cout << "Typ int ma rozmiar " << sizeof(int) << " bajtow";
```

Deklaracja *static_assert(bool _constexpr, const char *tekst)*

jeszcze w czasie kompilacji sprawdza warunek, podany jako pierwszy argument – jego wartość musi dać się obliczyć podczas kompilacji. Gdy warunek nie jest spełniony, to pojawia się komunikat o błędzie kompilacji, zawierający *tekst* podany jako drugi argument.

Deklaracja *static_assert(bool _constexpr, const char *tekst)*

jeszcze w czasie kompilacji sprawdza warunek, podany jako pierwszy argument – jego wartość musi dać się obliczyć podczas kompilacji. Gdy warunek nie jest spełniony, to pojawia się komunikat o błędzie kompilacji, zawierający *tekst* podany jako drugi argument.

Operator **alignof**

informuje o najkorzystniejszym wyrównaniu adresu dla danego typu. Zwracaną wartością jest pewna potęga liczby 2, która może być wykorzystana przy definiowaniu innych obiektów, np.
alignas(alignof(long)) char z;
Wyrównywać adres możemy tylko w górę, tzn. dla typu char – numery adersów pamięci po wyrównaniu muszą być podzielne przez co najmniej 2^1.

Przekształcenia typów

Automatycznie wykonywane są tylko takie przekształcenia, w których bez utraty informacji, argument o mniejszej precyzji zamieniany jest na argument o większej precyzji.

Przekształcenia typów

Automatycznie wykonywane są tylko takie przekształcenia, w których bez utraty informacji, argument o mniejszej precyzji zamieniany jest na argument o większej precyzji.

Działania operatorów dwuargumentowwych

Jeśli argumenty operatora dwuargumentowego (np. „+”, „–”) są różnego typu, to przed wykonaniem operacji typ „mniejszy” jest przekształcany do „większego”.

Przekształcenia typów

Automatycznie wykonywane są tylko takie przekształcenia, w których bez utraty informacji, argument o mniejszej precyzji zamieniany jest na argument o większej precyzji.

Działania operatorów dwuargumentowwych

Jeśli argumenty operatora dwuargumentowego (np. „+”, „–”) są różnego typu, to przed wykonaniem operacji typ „mniejszy” jest przekształcany do „większego”.

Przypisania

W przypisaniach wartość prawej strony jest przekształcana to typu wyniku z lewej strony. Zamiana *float* na *int* powoduje obcięcie części ułamkowej.

Operatory rzutowania

Stare niezalecane sposoby rzutowania

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie
- *nazwa_typu* (wyrażenie)

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie
- *nazwa_typu* (wyrażenie)

Nowe operatory rzutowania

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie
- *nazwa_typu* (wyrażenie)

Nowe operatory rzutowania

- *static_cast<nazwa_typu>*(wyrażenie) – stosuje się w sytuacjach bezpiecznych jeśli chodzi o utratę danych lub bezpiecznych na naszą odpowiedzialność

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie
- *nazwa_typu* (wyrażenie)

Nowe operatory rzutowania

- *static_cast<nazwa_typu>*(wyrażenie) – stosuje się w sytuacjach bezpiecznych jeśli chodzi o utratę danych lub bezpiecznych na naszą odpowiedzialność
- *const_cast<nazwa_typu>*(wyrażenie) – może służyć do pozbycia się lub nadania specyfikatora *const* lub *volatile*

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie
- *nazwa_typu* (wyrażenie)

Nowe operatory rzutowania

- *static_cast<nazwa_typu>*(wyrażenie) – stosuje się w sytuacjach bezpiecznych jeśli chodzi o utratę danych lub bezpiecznych na naszą odpowiedzialność
- *const_cast<nazwa_typu>*(wyrażenie) – może służyć do pozbycia się lub nadania specyfikatora *const* lub *volatile*
- *dynamic_cast<nazwa_typu>*(wyrażenie) – jeśli w trakcie wykonywania programu rzutowanie ma sens, to je wykonuje, w przeciwnym przypadku rzuca objekt wyjątku

Operatory rzutowania

Stare niezalecane sposoby rzutowania

- (*nazwa_typu*) wyrażenie
- *nazwa_typu* (wyrażenie)

Nowe operatory rzutowania

- *static_cast<nazwa_typu>*(wyrażenie) – stosuje się w sytuacjach bezpiecznych jeśli chodzi o utratę danych lub bezpiecznych na naszą odpowiedzialność
- *const_cast<nazwa_typu>*(wyrażenie) – może służyć do pozbycia się lub nadania specyfikatora *const* lub *volatile*
- *dynamic_cast<nazwa_typu>*(wyrażenie) – jeśli w trakcie wykonywania programu rzutowanie ma sens, to je wykonuje, w przeciwnym przypadku rzuca objekt wyjątku
- *reinterpret_cast<nazwa_typu>*(wyrażenie) – wykonuje rzutowanie wskaźników na naszą odpowiedzialność

Priorytety i łączność operatorów

Jednoargumentowe operatory „+”, „−”, „*” oraz „&” mają proiorytet wyższy niż ich odpowiedniki dwuargumentowe.

Priorytety i łączność operatorów

Jednoargumentowe operatory „+”, „–”, „*” oraz „&” mają proiorytet wyższy niż ich odpowiedniki dwuargumentowe.

Operatory jednoargumentowe są prawostronnie łączne (działają na argument stojący po ich prawej stronie) – za wyjątkiem operatorów postinkrementacji i postdekrementacji.
Operatory dwuargumentowe są lewostronnie łączne, za wyjątkiem operatorów przypisania (prawostronnie łącznych).

Priorytety od najwyższego oraz łączność

nawiasy w wyrażeniu	(wyrażenie)
wyrażenie lambda	[lista przechw.] lambda deklarator [lista instr.]

Priorytety od najwyższego oraz łączność

nawiasy w wyrażeniu	(wyrażenie)
wyrażenie lambda	[lista przechw.] lambda deklarator [lista instr.]

Lewostronna

określenie zakresu	nazwaKlasy::składnik
określenie zakresu	nazwaPrzestrzeniNazw::składnik
nazwa globalna	::nazwaGlobalna

Priorytety od najwyższego oraz łączność

nawiasy w wyrażeniu	(wyrażenie)
wyrażenie lambda	[lista przechw.] lambda deklarator [lista instr.]

Lewostronna

określenie zakresu	nazwaKlasy::składnik
określenie zakresu	nazwaPrzestrzeniNazw::składnik
nazwa globalna	::nazwaGlobalna

L

wybranie składnika	obiekt . składnik
wybranie składnika	wskaźnik –> składnik

Prawostronna

element tablicy (indeksowanie)	wskaźnik[wyrażenie]
wywołanie funkcji	funkcja(listaArgumentów)
konstrukcja wartości	typ{listaWyrażeń}
rzutowanie (w stylu wyw. funkcji)	typ(wyrażenie)
postinkrementacja	lwartość ++
postdekrementacja	lwartość – –
identyfikacja typu wg nazwy	typeid(typ)
identyfikacja typu wyrażenia	typeid(wyrażenie)
rzutowanie (spr. w trakcie wykonania)	dynamic_cast<typ>(wyrażenie)
rzutowanie (spr. w trakcie kompilacji)	static_cast<typ>(wyrażenie)
rzutowanie niesprawdzane	reinterpret_cast<typ>(wyrażenie)
konwersja const (lub volatile)	const_cast<typ>(wyrażenie)

P część 1

rozmiar obiektu	sizeof(wyrażenie)
rozmiar typu	sizeof(typ)
rozmiar paczki parametrów	sizeof... nazwa
wyrównanie dla typu	alignof(typ)
preinkrementacja	++ lwartość
predekrementacja	– – lwartość
negacja każdego bitu	~ wyrażenie
negacja	!wyrażenie
jednoargumentowy minus	– wyrażenie
jednoargumentowy plus	+ wyrażenie
adres czegoś	& lwartość
odniesienie się wskaźnikiem	*wyrażenie
twórz (rezerwój nowy obiekt)	new typ
twórz obiekt i inicjalizuj go ()	new typ (listaWyrażeń)
twórz obiekt i inicjalizuj go {}	new typ {listaWyrażeń}

P część 2

twórz w określonym miejscu	new (adresMiejsca) typ
twórz w okr. miejscu i inicjalizuj ()	new (adresMiejsca) typ (listaWyrażeń)
twórz w okr. miejscu i inicjalizuj {}	new (adresMiejsca) typ {listaWyrażeń}
zlikwiduj (anuluj rezerwację)	delete wskaźnik
zlikwiduj tablicę (macierz)	delete [ ] wskaźnik
czy wyrażenie rzuca wyjątek	noexcept(wyrażenie)
rzutowanie (konwersja typu)	typ(wyrażenie)

P część 2

twórz w określonym miejscu	new (adresMiejsca) typ
twórz w okr. miejscu i inicjalizuj ()	new (adresMiejsca) typ (listaWyrażeń)
twórz w okr. miejscu i inicjalizuj {}	new (adresMiejsca) typ {listaWyrażeń}
zlikwiduj (anuluj rezerwację)	delete wskaźnik
zlikwiduj tablicę (macierz)	delete [ ] wskaźnik
czy wyrażenie rzuca wyjątek	noexcept(wyrażenie)
rzutowanie (konwersja typu)	typ(wyrażenie)

L

wybór składnika wskaźnikiem i nazwą obiektu	obiekt . *wskDoSkładnika
wybór składnika wsk. i wskaźnikiem do obiektu	wsk –> *wskDoSkładnika

P część 2

twórz w określonym miejscu	new (adresMiejsca) typ
twórz w okr. miejscu i inicjalizuj ()	new (adresMiejsca) typ (listaWyrażeń)
twórz w okr. miejscu i inicjalizuj {}	new (adresMiejsca) typ {listaWyrażeń}
zlikwiduj (anuluj rezerwację)	delete wskaźnik
zlikwiduj tablicę (macierz)	delete [ ] wskaźnik
czy wyrażenie rzuca wyjątek	noexcept(wyrażenie)
rzutowanie (konwersja typu)	typ(wyrażenie)

L

wybór składnika wskaźnikiem i nazwą obiektu	obiekt . *wskDoSkładnika
wybór składnika wsk. i wskaźnikiem do obiektu	wsk –> *wskDoSkładnika

L

mnożenie	wyrażenie * wyrażenie
dzielenie	wyrażenie / wyrażenie
reszta z dzielenia	wyrażenie % wyrażenie

L

dodawanie	wyrażenie + wyrażenie
odejmowanie	wyrażenie − wyrażenie

L

dodawanie	wyrażenie + wyrażenie
odejmowanie	wyrażenie − wyrażenie

L

przesunięcie w lewo	wyrażenie << wyrażenie
przesunięcie w prawo	wyrażenie >> wyrażenie

L	
dodawanie	wyrażenie + wyrażenie
odejmowanie	wyrażenie − wyrażenie

L	
przesunięcie w lewo	wyrażenie << wyrażenie
przesunięcie w prawo	wyrażenie >> wyrażenie

L	
mniejsze niż	wyrażenie < wyrażenie
mniejsze lub równe	wyrażenie <= wyrażenie
większe niż	wyrażenie > wyrażenie
większe lub równe	wyrażenie >= wyrażenie

L

dodawanie	wyrażenie + wyrażenie
odejmowanie	wyrażenie − wyrażenie

L

przesunięcie w lewo	wyrażenie << wyrażenie
przesunięcie w prawo	wyrażenie >> wyrażenie

L

mniejsze niż	wyrażenie < wyrażenie
mniejsze lub równe	wyrażenie <= wyrażenie
większe niż	wyrażenie > wyrażenie
większe lub równe	wyrażenie >= wyrażenie

L

równe	wyrażenie == wyrażenie
różne od	wyrażenie != wyrażenie

bitowy iloczyn wyrażenie & wyrażenie

L	
bitowy iloczyn	wyrażenie & wyrażenie

L	
bitowa różnica symetryczna	wyrażenie ^ wyrażenie

L	
bitowy iloczyn	wyrażenie & wyrażenie

L	
bitowa różnica symetryczna	wyrażenie ^ wyrażenie

L	
bitowa alternatywa	wyrażenie | wyrażenie

L

bitowy iloczyn wyrażenie & wyrażenie

L

bitowa różnica symetryczna wyrażenie ^ wyrażenie

L

bitowa alternatywa wyrażenie | wyrażenie

L

logiczna koniunkcja wyrażenie && wyrażenie

L	
bitowy iloczyn	wyrażenie & wyrażenie

L	
bitowa różnica symetryczna	wyrażenie ^ wyrażenie

L	
bitowa alternatywa	wyrażenie | wyrażenie

L	
logiczna koniunkcja	wyrażenie && wyrażenie

L	
logiczna alternatywa	wyrażenie || wyrażenie

L	
bitowy iloczyn	wyrażenie & wyrażenie

L	
bitowa różnica symetryczna	wyrażenie ^ wyrażenie

L	
bitowa alternatywa	wyrażenie | wyrażenie

L	
logiczna koniunkcja	wyrażenie && wyrażenie

L	
logiczna alternatywa	wyrażenie || wyrażenie

P	
wyrażenie warunkowe	wyrażenie ? wyrażenie : wyrażenie

P

lista	{lista wyrażeń}
rzucenie wyjątku	throw wyrażenie
zwykłe przypisanie	lwartość = wyrażenie
mnóż i przypisz	lwartość *= wyrażenie
dziel i przypisz	lwartość /= wyrażenie
modulo i przypisz	lwartość %= wyrażenie
dodaj i przypisz	lwartość += wyrażenie
odejmij i przypisz	lwartość −= wyrażenie
przesuń w lewo i przypisz	lwartość <<= wyrażenie
przesuń w prawo i przypisz	lwartość >>= wyrażenie
koniunkcja bitowa i przypisanie	lwartość &= wyrażenie
alternatywa bitowa i przypisanie	lwartość |= wyrażenie
różnica symetryczna i przypisanie	lwartość ^= wyrażenie

P

lista	{lista wyrażeń}
rzucenie wyjątku	throw wyrażenie
zwykłe przypisanie	lwartość = wyrażenie
mnóż i przypisz	lwartość *= wyrażenie
dziel i przypisz	lwartość /= wyrażenie
modulo i przypisz	lwartość %= wyrażenie
dodaj i przypisz	lwartość += wyrażenie
odejmij i przypisz	lwartość −= wyrażenie
przesuń w lewo i przypisz	lwartość <<= wyrażenie
przesuń w prawo i przypisz	lwartość >>= wyrażenie
koniunkcja bitowa i przypisanie	lwartość &= wyrażenie
alternatywa bitowa i przypisanie	lwartość |= wyrażenie
różnica symetryczna i przypisanie	lwartość ^= wyrażenie

L

przecinek	wyrażenie , wyrażenie

L	::
L	-> .
L	() [] przyrostkowe ++ −− typ static_cast
P	! ~ ++ −− + − * & new delete (typ) sizeof
L	. -> składnik wskaźnikiem
L	* / %
L	+ −
L	<< >>
L	< <= > >=
L	== !=
L	&
L	^
L	|
L	&&
L	||
P	?:
P	= += −= *= /= %= ^= |= <<= >>= throw
L	,

Klasa std::vector

Aby korzystać z bibliotecznej klasy *vector* musimy dołączyć plik nagłówkowy <vector>, zawierający deklarację tej klasy.
Nazwa klasy *vector* wchodzi do przestrzeni nazw *std*.

Klasa std::vector

Aby korzystać z bibliotecznej klasy *vector* musimy dołączyć plik nagłówkowy <vector>, zawierający deklarację tej klasy.
Nazwa klasy *vector* wchodzi do przestrzeni nazw *std*.

Sposoby definiowania obiektów klasy *vector*:

```
#include <vector>
using namespace std;

vector<double> a; // tworzy pusty wektor - rozmiaru zero
vector<int> liczbyPierwsze {2, 3, 5, 7};
vector<string> nazwiska(12); // tworzy wektor 12 elementowy
// każdy element zawiera odpowiednie dla danego typu zero

vector<char> znaki(20, '*'); // tworzy 20 elementów '*'
```

Podstawowe funkcje klasy **vector**

Indeksowanie elementów wektora zaczyna się od zera.

```
// 3 - realizacja dostępu do drugiego elementu obiektu
cout << liczbyPierwsze[1];

// 4 - wartością jest ilość elementów obiektu
liczbyPierwsze.size();

// dodaje do wektora nowy element
liczbyPierwsze.push_back(11);
```

Zakresowe for

przebiega przez wszystkie elementy tzw. pojemnika, którym może być wektor, tablica, string czy lista inicjalizatorów (ujęta w nawiasy klamrowe sekwencja stałych dosłownych – nie można ich modyfikować).
for(auto & element : pojemnik) cout << element;

- *auto* – można je użyć zamiast nazwy typu elementów pojemnika, jeśli poprzedzimy je *const*, zagwarantuje to brak modyfikacji elementów pojemnika
- *&* – stawiamy aby pracować na oryginale
- *element* – kopia elementu pojemnika

Zakresowe for

przebiega przez wszystkie elementy tzw. pojemnika, którym może być wektor, tablica, string czy lista inicjalizatorów (ujęta w nawiasy klamrowe sekwencja stałych dosłownych – nie można ich modyfikować).
for(auto & element : pojemnik) cout << element;

- *auto* – można je użyć zamiast nazwy typu elementów pojemnika, jeśli poprzedzimy je *const*, zagwarantuje to brak modyfikacji elementów pojemnika
- *&* – stawiamy aby pracować na oryginale
- *element* – kopia elementu pojemnika

```
for(auto & c : liczbyPierwsze) c = -c;

for(auto w : {'A', 'B', 'C', 'D'}) cout << '\t' << w;
```

Wektory wielowymiarowe

Wektor dwuwymiarowy (*wektor 2D*) podobnie jak w przypadku tablic, jest wektorem wektorów jednowymiarowych: vector<vector<double>> a;

Wektory wielowymiarowe

Wektor dwuwymiarowy (*wektor 2D*) podobnie jak w przypadku tablic, jest wektorem wektorów jednowymiarowych: vector<vector<double>> a;

```
void Wypisz(vector<vector<double>> a){ // a jest kopią
   for(int w=0; w<a.size(); ++w) // indeks wiersza
      for(int k=0; k<a[w].size(); ++k) // indeks kolumny
         cout << a[w][k] << (k<a[w].size()-1 ? '\t' : '\n');
}
   // lub dla zakresowego for (vector wierszy)
void Wypisz(vector<vector<double>> a){
   for(auto wiersz : a){ // wiersze
      for(auto element : wiersz) // elementy wektora 2D
         cout << '\t' << element;
      cout << '\n';
   }
}
```

Definicje wektorów 2D

Za pomocą *listy inicjalizatorów* wektor sam określa swoje wymiary:

```
vector<vector<int>> potegi{
    {1, 2, 3, 4},
    {1, 4, 9, 16},
    {1, 8, 27, 64}
};
```

Definicje wektorów 2D

Za pomocą *listy inicjalizatorów* wektor sam określa swoje wymiary:

```
vector<vector<int>> potegi{
    {1, 2, 3, 4},
    {1, 4, 9, 16},
    {1, 8, 27, 64}
};
```

Rozmiary można podać za pomocą zwykłych zmiennych:

```
int wiersze=50, kolumny=400;
vector<vector<double>> dane(
    wiersze,
    vector<double>(kolumny)
);
```

Ten sposób powoduje zerowanie nowo utworzonych elementów.

Funkcja **resize**

zmienia rozmiary wektora. Wiersze mogą być różnej długości. Powstałe elementy inicjalizuje zerami.

```
vector<vector<int>> a; // definicja pustego wektora
a.resize(wiersze); // tworzy puste wiersze
for(int w=0; w<wiersze; ++w)
   a[w].resize(kolumny); // wydłuża wiersze i zeruje
```

Jeśli zmniejszymy liczbę wierszy, to skasowane zostaną te ostatnie. Jeśli zmniejszymy długość wiersza, to skasowane zostaną jego ostatnie elementy.

Funkcja push_back

dodaje do wektora jeden element o zadanej wartości.

```
vector<vector<int>> a;
for(int w=0; w<wiersze; ++w){
   vector<int> nowyWiersz;
   for(int k=0; k<kolumny; ++k)
      nowyWiersz.push_back(w+k); // powiększa nowy wiersz
   a.push_back(nowyWiersz); // dodaje nowy wiersz do a
}
a[2].push_back(0); // powiększa trzeci wiersz o wartość 0
a.push_back(vector<int>{4, 3, 2, 1}); // dołącza dany wiersz
```

Funkcja push_back

dodaje do wektora jeden element o zadanej wartości.

```
vector<vector<int>> a;
for(int w=0; w<wiersze; ++w){
   vector<int> nowyWiersz;
   for(int k=0; k<kolumny; ++k)
      nowyWiersz.push_back(w+k); // powiększa nowy wiersz
   a.push_back(nowyWiersz); // dodaje nowy wiersz do a
}
a[2].push_back(0); // powiększa trzeci wiersz o wartość 0
a.push_back(vector<int>{4, 3, 2, 1}); // dołącza dany wiersz
```

Funkcja pop_back

w pełni analogiczna do *push_back*, usuwa ostatni element wektora, np. a[2].pop_back(); a.pop_back();

Wektory trójwymiarowe

Wektor trójwymiarowy jest wektorem wektorów dwuwymiarowych:

```
void Podstaw(vector<vector<vector<char>>> & c, char z){
   for(int i=0; i<c.size(); ++i) // indeks
      for(auto & cj : c[i])
         for(auto & ck : cj) // elementy wektora 3D
            ck = z;
}
```

Odbieranie wektora przez referencję daje pełen do niego dostęp – nie tylko do adresu.

Definicje wektora 3D o zadanych wymiarach

```
int x=50, y=100, z=30;
vector<vector<vector<char>>> a(
   x, vector<vector<char>>(
      y, vector<char>(z, '+'))
);
```

Definicje wektora 3D o zadanych wymiarach

```
int x=50, y=100, z=30;
vector<vector<vector<char>>> a(
   x, vector<vector<char>>(
      y, vector<char>(z, '+'))
);
```

```
using Tz = vector<char>;
using Ty = vector<Tz>;
using Tx = vector<Ty>;
Tx c(x, Ty(y, Tz(z, '+')));
```

Definicje wektora 3D o zadanych wymiarach

```
int x=50, y=100, z=30;
vector<vector<vector<char>>> a(
   x, vector<vector<char>>(
      y, vector<char>(z, '+'))
);
```

```
using Tz = vector<char>;
using Ty = vector<Tz>;
using Tx = vector<Ty>;
Tx c(x, Ty(y, Tz(z, '+')));
```

Zmiana rozmiarów i zapełnianie wektorów 3D jest w pełni analogiczne do wektorów 2D.

Funkcje

Każda definicja funkcji ma następujący format:

```
typ_powrotu nazwa_funkcji (deklaracje parametrów)
{
   deklaracje i instrukcje
}
```

Jeśli funkcja nie zwraca żadnych wartości, jej typ powrotu oznaczany jest przez **void**.

Instrukcja **return**

Wywołana funkcja przekazuje do miejsca wywołania wartość wyrażenia następującego po instrukcji *return*.
W przypadku funkcji zwracającej typ *void*, instrukcję *return* można opuścić lub w celu zakończenia działania funkcji dodać po niej sam średnik: *return;*

Instrukcja **return**

Wywołana funkcja przekazuje do miejsca wywołania wartość wyrażenia następującego po instrukcji *return*.
W przypadku funkcji zwracającej typ *void*, instrukcję *return* można opuścić lub w celu zakończenia działania funkcji dodać po niej sam średnik: *return;*

W funkcji *main* instrukcję *return 0;* można opuścić – kompilator sam doda ją przed zamykającym ją nawiasem *}*. Zwrot wartości 0 oznacza poprawne zakończenie działania całego programu.

Deklaracja funkcji

Każda nazwa przed jej użyciem musi być zadeklarowana.
Każda definicja funkcji jest jednocześnie jej deklaracją.
Jeśli definicja funkcji nie znajduje się w pliku powyżej jej wywołania, konieczna jest jej wcześniejsza deklaracja. Takie deklaracje funkcji najlepiej umieszczać na początku pliku:

typ_powrotu nazwa_funkcji(deklaracje parametrów);
auto nazwa_funkcji(deklaracje parametrów) -> typ_powrotu;

```
int zapisz(int c, char z);
auto zapisz(int c, char z) -> decltype(c);
```

Deklaracja funkcji

Każda nazwa przed jej użyciem musi być zadeklarowana.
Każda definicja funkcji jest jednocześnie jej deklaracją.
Jeśli definicja funkcji nie znajduje się w pliku powyżej jej wywołania, konieczna jest jej wcześniejsza deklaracja. Takie deklaracje funkcji najlepiej umieszczać na początku pliku:

typ_powrotu nazwa_funkcji(deklaracje parametrów);
auto nazwa_funkcji(deklaracje parametrów) -> typ_powrotu;

```
int zapisz(int c, char z);
auto zapisz(int c, char z) -> decltype(c);
```

Parametry w deklaracji funkcji

Jeśli brak deklaracji parametrów lub w ich miejscu jest słowo *void*, mamy brak jakichkolwiek argumentów.
Jeśli w ich miejscu są ..., mamy dowolną liczbę argumentów.
Nazwy argumntów (nie typy) w nawiasach można pominąć.

Argumenty funkcji

zapisane w jej definicji nazywane są **formalnymi**, **argumenty aktualne** podajemy przy wywołaniu funkcji.
Argumenty funkcji prekazywane są przez wartość – są to kopie zmiennych, do których funkcja nie ma dostępu.
Zdefiniowane w obrębie funkcji zmienne przechowywane są zwykle na **stosie**, czyli w podręcznej pamięci.

Argumenty funkcji

zapisane w jej definicji nazywane są **formalnymi**, **argumenty aktualne** podajemy przy wywołaniu funkcji.
Argumenty funkcji prekazywane są przez wartość – są to kopie zmiennych, do których funkcja nie ma dostępu.
Zdefiniowane w obrębie funkcji zmienne przechowywane są zwykle na **stosie**, czyli w podręcznej pamięci.

Nienazwany argument

Gdy w definicji funkcji wpiszemy na liście sam typ argumentu bez jego nazwy, kompilator zinterpretuje funkcję jako wywoływaną z argumentem danego typu, lecz nie wykorzystywanym w tej funkcji.

Przesyłanie argumentów przez referencję

Do funkcji argumenty można przesyłać również przez referencję – przesyłany jest wówczas adres danej zmiennej. W deklaracji funkcji, w liście argumentów nazwy tych zmiennych poprzedza się operatorem adresu &. W ciele tej funkcji i podczas jej wywołania operator ten nie występuje.

```
double f(int a, double &b){ return a+b; }
f(a, b);
```

Przesyłanie argumentów przez referencję

Do funkcji argumenty można przesyłać również przez referencję – przesyłany jest wówczas adres danej zmiennej. W deklaracji funkcji, w liście argumentów nazwy tych zmiennych poprzedza się operatorem adresu &. W ciele tej funkcji i podczas jej wywołania operator ten nie występuje.

```
double f(int a, double &b){ return a+b; }
f(a, b);
```

Działania na zmiennych podanych przez referencję jako argumenty funkcji, skutkują odpowiednimi zmianami wartości tych zmiennych.
Sposób przekazywania zmiennych przez referencję znacznie utrudnia śledzenie zmian wartości tych zmiennych. Nie wymaga jednak tworzenia kopii obiektów, co przyspiesza wykonywanie funkcji.

Lwartość i rwartość

Lwartość to wyrażenie, które może występować po lewej stronie operatora przypisania. **Rwartość** zaś nie może występować po jego lewej stronie. Lwartością może być obiekt albo wskaźnik czy referencja do obiektu. Rwartość to obiekt chwilowy. Zostaje obliczony, użyty i może być zniszczony.

Lwartość i rwartość

Lwartość to wyrażenie, które może występować po lewej stronie operatora przypisania. **Rwartość** zaś nie może występować po jego lewej stronie. Lwartością może być obiekt albo wskaźnik czy referencja do obiektu. Rwartość to obiekt chwilowy. Zostaje obliczony, użyty i może być zniszczony.

Referencję do rwartości w definicji poprzedzamy dwoma znakami &&, np. int && a = 5;

Lwartość i rwartość

Lwartość to wyrażenie, które może występować po lewej stronie operatora przypisania. **Rwartość** zaś nie może występować po jego lewej stronie. Lwartością może być obiekt albo wskaźnik czy referencja do obiektu. Rwartość to obiekt chwilowy. Zostaje obliczony, użyty i może być zniszczony.

Referencję do rwartości w definicji poprzedzamy dwoma znakami &&, np. int && a = 5;

Sposoby przesyłania i deklaracji argumentu do funkcji:

- int f(int a); – przesyłana kopia lwartości i rwartości
- int f(const int & a); – przesyłana jest referencja (adres) do stałej lwartości i rwartości, dla dużych obiektów bez możliwości modyfikacji
- int f(int & a); – przesyłana jest referencja do lwartości, możliwość modyfikacji
- int f(int && a); – przesyłana referencja do rwartości

Argumenty domniemane

Każdy z argumentów domniemanych w danym zakresie ważności piszemy tylko raz – w deklaracji funkcji. W jej defnicji tylko wtedy, gdy spełnia ona jednocześnie rolę deklaracji, występując w pliku powyżej jej wywołania.

Argumenty domniemane

Każdy z argumentów domniemanych w danym zakresie ważności piszemy tylko raz – w deklaracji funkcji. W jej defnicji tylko wtedy, gdy spełnia ona jednocześnie rolę deklaracji, występując w pliku powyżej jej wywołania.

W deklaracji argumentowi przypisujemy pewną wartość. Argumenty domniemane muszą występować na końcu listy argumentów, np.

```
int f(int a, char b='z', int c=5, float d=4.5f);
int f(int, char ='z', int =5, float =4.5f);
```

Argumenty domniemane

Każdy z argumentów domniemanych w danym zakresie ważności piszemy tylko raz – w deklaracji funkcji. W jej defnicji tylko wtedy, gdy spełnia ona jednocześnie rolę deklaracji, występując w pliku powyżej jej wywołania.

W deklaracji argumentowi przypisujemy pewną wartość. Argumenty domniemane muszą występować na końcu listy argumentów, np.

```
int f(int a, char b='z', int c=5, float d=4.5f);
int f(int, char ='z', int =5, float =4.5f);
```

Argumenty domniemane można rozpisać w kilku deklaracjach, przesuwając się z domniemanymi wartościami w stronę początku listy, nie pomijając żadnego np.

```
int f(int, char, int, float);
int f(int, char, int, float =4.5f);
int f(int, char ='z', int =5, float);
```

Przykłady wywołania funkcji f: f(3); f(3, 'C'); f(3, 'C', 1); f(3, 'C', 1, 2.7);

Przykłady wywołania funkcji f: f(3); f(3, 'C'); f(3, 'C', 1); f(3, 'C', 1, 2.7);

Deklaracja bez argumentów domniemanych może wystąpić też po deklaracji z domniemaniem, np.

```
int g(char a, int b, char c='z');
int g(char a, int b, char c);
```

Przykłady wywołania funkcji f: f(3); f(3, 'C'); f(3, 'C', 1); f(3, 'C', 1, 2.7);

Deklaracja bez argumentów domniemanych może wystąpić też po deklaracji z domniemaniem, np.

```
int g(char a, int b, char c='z');
int g(char a, int b, char c);
```

Wartościami domniemanymi mogą być również wyrażenia zawierające obiekty globalne (nie lokalne!) i wywołania funkcji.

Przykłady wywołania funkcji f: f(3); f(3, 'C'); f(3, 'C', 1); f(3, 'C', 1, 2.7);

Deklaracja bez argumentów domniemanych może wystąpić też po deklaracji z domniemaniem, np.

```
int g(char a, int b, char c='z');
int g(char a, int b, char c);
```

Wartościami domniemanymi mogą być również wyrażenia zawierające obiekty globalne (nie lokalne!) i wywołania funkcji.

W nowym lokalnym zakresie ważności, np. po otwarciu bloku nawiasem klamrowym, możemy umieścić ponownie deklarację funkcji z nowymi wartościami domniemanych argumentów. Wtedy wszystkie poprzednie domniemane wartości tracą ważność do końca bieżącego zakresu ważności.

Funkcje **inline**

Dodatkowe słowo **inline** poprzedzające definicję funkcji, daje możliwość kompilatorowi wypisania jej treści w miejscu jej wywołania. Wtedy kod programu wykonywany będzie szybciej, pomijając kilka instrukcji na poziomie języka maszynowego (asemblera), związanych z wywołaniem i zakończeniem funkcji.

```
inline int zaokr(double a){ return a+.5; }
```

Funkcje **inline**

Dodatkowe słowo **inline** poprzedzające definicję funkcji, daje możliwość kompilatorowi wypisania jej treści w miejscu jej wywołania. Wtedy kod programu wykonywany będzie szybciej, pomijając kilka instrukcji na poziomie języka maszynowego (asemblera), związanych z wywołaniem i zakończeniem funkcji.

```
inline int zaokr(double a){ return a+.5; }
```

Funkcje typu *inline* przewidziane zostały dla krótkich funkcji. Mogą jednak zawierać definicje obiektów lokalnych czy statycznych.
W przypadku bardziej rozbudowanych, kompilator utworzy zwykłą funkcję.
Polecenie *inline* nie zostanie uwzględnione, gdy funkcję kompilujemy dla pracy z programem diagnostycznym (debuggerem).
Kompilator napotykając wywołanie funkcji *inline*, musi wstawić właściwe instrukcje. Sama poprzedzająca deklaracja więc nie wystarczy.
Definicje tych funkcji powinny znajdować się na samej górze pliku programu lub w dołączonym pliku nagłówkowym.

Obiekty lokalne automatyczne

są definowane w bloku. Przy wyjściu z tego bloku obiekty te automatycznie przestają istnieć.
Zmienne automatyczne nie są zerowane w chwili ich definicji. Obiekty te komputer przechowuje na stosie i przydziela wymagany dla nich obszar.

Obiekty lokalne automatyczne

są definowane w bloku. Przy wyjściu z tego bloku obiekty te automatycznie przestają istnieć.
Zmienne automatyczne nie są zerowane w chwili ich definicji. Obiekty te komputer przechowuje na stosie i przydziela wymagany dla nich obszar.

Obiekty lokalne statyczne

Obiekt statyczny definiowany w obrębie funkcji, po zakończeniu jej pracy nie jest niszczony i zachowuje swoją wartość. Nie jest jednak dostępny spoza zakresu funkcji. W definicji używamy słowa **static**, np.
static float predkosc; // wartość początkowa wynosi 0
static float predkosc = 12; // wartość początkowa wynosi 12 – jednorazowo

Obiekty lokalne automatyczne

są definowane w bloku. Przy wyjściu z tego bloku obiekty te automatycznie przestają istnieć.
Zmienne automatyczne nie są zerowane w chwili ich definicji. Obiekty te komputer przechowuje na stosie i przydziela wymagany dla nich obszar.

Obiekty lokalne statyczne

Obiekt statyczny definiowany w obrębie funkcji, po zakończeniu jej pracy nie jest niszczony i zachowuje swoją wartość. Nie jest jednak dostępny spoza zakresu funkcji. W definicji używamy słowa **static**, np.
static float predkosc; // wartość początkowa wynosi 0
static float predkosc = 12; // wartość początkowa wynosi 12 – jednorazowo

Obiekty lokalne statyczne, globalne i z przestrzeni nazw tworzone są w normalnym obszarze pamięci, który jest zerowany przed uruchomieniem programu. Obiekty te od samego początku mają więc wartość zero, odpowiednią dla danego typu.

Funkcje rekurencyjne

Funkcja może wywołać samą siebie bezpośrednio lub pośrednio np. w parze z inną wywołując się wzajemnie.
Funkcja przeznaczona do rekurencyjnego wywołania powinna posiadać warunek, w którym decyduje czy wywołać samą siebie czy zatrzymać rekurencję.

Funkcje rekurencyjne

Funkcja może wywołać samą siebie bezpośrednio lub pośrednio np. w parze z inną wywołując się wzajemnie.
Funkcja przeznaczona do rekurencyjnego wywołania powinna posiadać warunek, w którym decyduje czy wywołać samą siebie czy zatrzymać rekurencję.

Algorytm Euklidesa

Funkcja do obliczeń Największego Wspólnego Dzielnika:

```
int NWD(int m, int n){ return n>0 ? NWD(n, m%n) : m; }
```

Działanie funkcji rekurencyjnej

Rekurencja pozwala na proste wykonanie skomplikowanego algorytmu. Rekurencja coraz bardziej się zagnieżdża, a po wystąpieniu warunku zatrzymania następują powroty z tych zagnieżdżeń. Dany poziom zagnieżdżenia ma swój własny zestaw zmiennych, którym może nadać pewne wartości obliczone przy zagnieżdżaniu, a skorzystać z nich w trakcie powrotów.

Działanie funkcji rekurencyjnej

Rekurencja pozwala na proste wykonanie skomplikowanego algorytmu. Rekurencja coraz bardziej się zagnieżdża, a po wystąpieniu warunku zatrzymania następują powroty z tych zagnieżdżeń. Dany poziom zagnieżdżenia ma swój własny zestaw zmiennych, którym może nadać pewne wartości obliczone przy zagnieżdżaniu, a skorzystać z nich w trakcie powrotów.

Kolejne wywołanie rekurencyjne funkcji sprawia, że na stosie pojawiają się kolejne zmienne automatyczne, związane z kolejnym wywołaniem. Zmienna statyczna w tej sytuacji jest wspólna dla wszystkich rekurencyjnych wywołań funkcji.

Funkcje w programie złożonym z kilku plików

Program można podzielić na kilka plików. Można wtedy kompilować każdy plik osobno. Definicji funkcji nie można dzielić na kilka plików.

Funkcje w programie złożonym z kilku plików

Program można podzielić na kilka plików. Można wtedy kompilować każdy plik osobno. Definicji funkcji nie można dzielić na kilka plików.

Deklaracje zmiennych globalnych i funkcji

Aby funkcje i zmienne globalne z innych plików były dostępne, należy umieścić w danym pliku deklaracje odpowiednich funkcji i zmiennych globalnych. Wszystkie te deklaracje wygodnie jest umieścić w osobnym pliku nagłówkowym i włączać do innych plików dyrektywą *#include*.
Deklaracje zmiennych globalnych poprzedza się słowem **extern**.
Deklarację *extern* stosuje się również, gdy odwołanie do niej występuje przed jej definicją w tym samym pliku.
Instrukcja: *extern int a = 5;* jest równoważna definicji: *int a = 5;*

Ograniczenie ważności nazwy do bieżącego pliku

Aby dana nazwa obiektu globalnego lub funkcji była znana tylko w bieżącym pliku, można ją zadeklarować lub zdefiniować w tzw. **anonimowej przestrzeni nazw** (bez nazwy), np.

```
namespace{
    int t, s;
    double f1(int a);
    void f2(int d){ s += d; }
}
```

Przestarzałym sposobem na ograniczenie dostępu do danej nazwy jest poprzedzenie jej deklaracji czy definicji słowem *static*.

Funkcje zwracające referencje do lwartości

stosujemy gdy wywołanie takiej funkcji ma zastąpić nazwę obiektu, do którego coś przypiszemy, np. $f(2) = 5;$

Funkcje zwracające referencje do lwartości

stosujemy gdy wywołanie takiej funkcji ma zastąpić nazwę obiektu, do którego coś przypiszemy, np. $f(2) = 5;$

Funkcja taka może np. wybrać obiekt, którego referencję zwróci:

```
int a, b;
int & f(int wybor){
   return wybor>0 ? a : b;
}
```

Funkcja powinna zwracać referencję do obiektu, który będzie istniał bezpośrednio po zakończeniu jej pracy. Błąd polegający na zwrocie referencji do obiektu lokalnego, nazywa się *martwą referencją*.

Funkcje constexpr

można wykorzystać do obliczenia wartości (inicjalizacji) wielu obiektów *constexpr* jeszcze podczas kompilacji.
Funkcje te (użyte podczas kompilacji) muszą spełniać poniższe kryteria:

- *argumentami aktualnymi* muszą być wyrażenia zawierające stałe dosłowne, obiekty, funkcje czy referencje do obiektów typu *constexpr*
- ciało funkcji musi zawierać tylko jedną instrukcję *return* i ta instrukcja musi zwracać wyrażenie *constexpr*
- nie może zawierać instrukcji warunkowych, pętli, zmiennych pomocniczych

Funkcje constexpr

można wykorzystać do obliczenia wartości (inicjalizacji) wielu obiektów *constexpr* jeszcze podczas kompilacji.
Funkcje te (użyte podczas kompilacji) muszą spełniać poniższe kryteria:

- *argumentami aktualnymi* muszą być wyrażenia zawierające stałe dosłowne, obiekty, funkcje czy referencje do obiektów typu *constexpr*
- ciało funkcji musi zawierać tylko jedną instrukcję *return* i ta instrukcja musi zwracać wyrażenie *constexpr*
- nie może zawierać instrukcji warunkowych, pętli, zmiennych pomocniczych

Funkcja może zawierać:

- wyrażenia warunkowe (*? :*)
- zmienne globalne, statyczne (nie może zmieniać ich wartości)
- może wywołać samą siebie

Jeśli argumentami funkcji *constexpr* nie będą same wyrażenia typu *constexpr* lub gdy zwróci wartość innego typu, to kompilator uzna ją za zwykłą funkcję, której zwracana wartość nie ma cechy *constexpr*. W tym przypadku nie można użyć funkcji do inicjalizacji obiektu *constexpr*.

Jeśli argumentami funkcji *constexpr* nie będą same wyrażenia typu *constexpr* lub gdy zwróci wartość innego typu, to kompilator uzna ją za zwykłą funkcję, której zwracana wartość nie ma cechy *constexpr*. W tym przypadku nie można użyć funkcji do inicjalizacji obiektu *constexpr*.

Gdy funkcja *constexpr* nie jest używana jako zwykła funkcja (używana w czasie kompilacji, a nie w trakcie wykonywania programu), to jej definicja musi być umieszczona powyżej jej wywołania – tak jak dla funkcji *inline*.

Jeśli argumentami funkcji *constexpr* nie będą same wyrażenia typu *constexpr* lub gdy zwróci wartość innego typu, to kompilator uzna ją za zwykłą funkcję, której zwracana wartość nie ma cechy *constexpr*. W tym przypadku nie można użyć funkcji do inicjalizacji obiektu *constexpr*.

Gdy funkcja *constexpr* nie jest używana jako zwykła funkcja (używana w czasie kompilacji, a nie w trakcie wykonywania programu), to jej definicja musi być umieszczona powyżej jej wywołania – tak jak dla funkcji *inline*.

```
constexpr double c = 299792458.; // prędkość światła w próżni
constexpr double kwadrat(double a){ return a*a; }
constexpr double c2 = kwadrat(c);
constexpr int modul(const int & a){ return a<0 ? -a : a; }
```

Jeśli argumentami funkcji *constexpr* nie będą same wyrażenia typu *constexpr* lub gdy zwróci wartość innego typu, to kompilator uzna ją za zwykłą funkcję, której zwracana wartość nie ma cechy *constexpr*. W tym przypadku nie można użyć funkcji do inicjalizacji obiektu *constexpr*.

Gdy funkcja *constexpr* nie jest używana jako zwykła funkcja (używana w czasie kompilacji, a nie w trakcie wykonywania programu), to jej definicja musi być umieszczona powyżej jej wywołania – tak jak dla funkcji *inline*.

```
constexpr double c = 299792458.; // prędkość światła w próżni
constexpr double kwadrat(double a){ return a*a; }
constexpr double c2 = kwadrat(c);
constexpr int modul(const int & a){ return a<0 ? -a : a; }
```

Obiekty klasy *std::string* nie mają cechy *constexpr* – mają ją *C-stringi*.

Definicja referencji przy pomocy słowa *auto*

Referencja jest synonimem nazwy danego obiektu.
Słowo *auto* definiuje typ przy pomocy typu przypisywanej wartości.
W definicji słowem *auto* referencje są pomijane, podobnie jak specyfikatory *const* i *volatile*. Referencję zdefiniujemy stawiając za słowem *auto* znak &.
Jeśli definiując w ten sposób referencję, użyjemy stałego obiektu lub stałej referencji, to definiowana referencja otrzyma specyfikator *const*, aby nie zniszczyć zawartości pokazywanego referencją obiektu.
Stawiając na początku *const*, samodzielnie tworzymy stałą referencję.

Definicja referencji przy pomocy słowa *auto*

Referencja jest synonimem nazwy danego obiektu.
Słowo *auto* definiuje typ przy pomocy typu przypisywanej wartości.
W definicji słowem *auto* referencje są pomijane, podobnie jak specyfikatory *const* i *volatile*. Referencję zdefiniujemy stawiając za słowem *auto* znak &.
Jeśli definiując w ten sposób referencję, użyjemy stałego obiektu lub stałej referencji, to definiowana referencja otrzyma specyfikator *const*, aby nie zniszczyć zawartości pokazywanego referencją obiektu.
Stawiając na początku *const*, samodzielnie tworzymy stałą referencję.

```
int b = 3; // definicja obiektu
int & rb = b; // referencja do obiektu
const int & crb = b; // referencja do obiektu stałego
auto ak = b; auto akr = rb; auto akcr = crb; // kopie obiektu
auto & ar = b; auto & arr = rb; // referencje
auto & arcr = crb; // specyfikator const jest przenoszony
const auto & car = b; // tworzenie stałych referencji
const auto & carr = rb; const auto & carcr = crb;
```

Te same zasady obowiązują przy inicjalizacji za pomocą funkcji zwracającej referencję.

Te same zasady obowiązują przy inicjalizacji za pomocą funkcji zwracającej referencję.

```
double polozenie = 2., predkosc = .5; // zmienne globalne
double & parStanu(bool k){
   return k ? predkosc : polozenie;
}
const double & parStanu_c(bool k){
   return k ? predkosc : polozenie;
}
auto & ref_Stan = parStanu(true); // referencja do predkosc
auto & c_ref_Stan = parStanu_c(true); // referencja stała
```

Funkcje biblioteczne

Funkcje biblioteczne nie są częścią języka C++, lecz ze względu na ich przydatność dołączono je do biblioteki standardowej.
Wszystkie dostępne standardowe funkcje biblioteczne można znaleźć w opisie funkcji bibliotecznych danego kompilatora (*Reference Manual*).
Z daną funkcją biblioteczną zwykle związany jest plik nagłówkowy zawierający jej deklarację, który trzeba dołączyć do naszego pliku.

Funkcje biblioteczne

Funkcje biblioteczne nie są częścią języka C++, lecz ze względu na ich przydatność dołączono je do biblioteki standardowej.
Wszystkie dostępne standardowe funkcje biblioteczne można znaleźć w opisie funkcji bibliotecznych danego kompilatora (*Reference Manual*).
Z daną funkcją biblioteczną zwykle związany jest plik nagłówkowy zawierający jej deklarację, który trzeba dołączyć do naszego pliku.

Opisy funkcji bibliotecznych zawierają również takie strony jak `www.cplusplus.com` lub `en.cppreference.com`

Funkcje matematyczne: nagłówek <cmath>

Argumenty x, y oraz zwracane przez funkcje wartości są typu *double*.

sin(x)	
cos(x)	
tan(x)	tangens x
asin(x)	arcus sinus x w przedziale $[-\pi/2, \pi/2]$, x w $[-1, 1]$
acos(x)	arcus cosinus x w przedziale $[0, \pi]$, x w $[-1, 1]$
atan(x)	arcus tangens x w przedziale $[-\pi/2, \pi/2]$
atan2(y, x)	arcus tangens y/x w przedziale $[-\pi, \pi]$
sinh(x)	sinus hiperboliczny x
cosh(x)	cosinus hiperboliczny x
tanh(x)	tangens hiperboliczny x

Funkcje matematyczne cd..

exp(x)	funkcja wykładnicza o podstawie e
log(x)	logarytm naturalny $ln\ x$
log10(x)	logarytm o podstawie 10
pow(x, y)	x do potęgi y
sqrt(x)	pierwiastek z x
ceil(x)	najmniejsza liczba całkowita, nie mniejsza niż x
floor(x)	największa liczba całkowita, nie większa niż x
fabs(x)	wartość bezwzględna z x
fmod(x, y)	zmiennoprzecinkowa reszta z dzielenia x/y, ze znakiem x

Niektóre funkcje operujące na tekstach <cstring>

Tutaj *s*, *t* są wskaźnikami do znaków, a *c*, *n* są typu *int*.

strcat(s, t)	dopisuje *t* na koniec *s*
strncat(s, t, n)	dopisuje *n* znaków z *t* na koniec *s*
strcmp(s, t)	zwraca wartość ujemną gdy $s<t$, 0 dla $s=t$, dodatnią dla $s>t$
strncmp(s, t, n)	to samo co *strcmp*, tylko dla początkowych *n* znaków *t*
strcpy(s, t)	kopiuje *t* do *s*
strncpy(s, t, n)	kopiuje co najwyżej *n* znaków *t* do *s*
strlen(s)	zwraca długość *s*
strchr(s, c)	zwraca wskaźnik do pierwszego wystąpienia *c* w *s* lub *NULL* gdy *c* nie występuje w *s*
strrchr(s, c)	zwraca wskaźnik do ostatniego wystąpienia *c* w *s* lub *NULL* gdy *c* nie występuje w *s*

Niektóre funkcje przekształcające i badające znaki <cctype>

W poniższych opisach *c* jest typu *int* lub *unsigned char* łącznie z *EOF*. Wszystkie funkcje zwracają wartość typu *int*.

isalpha(c)	*true*, jeśli *c* jest literą, *false* jeśli nie
isupper(c)	*true*, jeśli *c* jest wielką literą, *false* jeśli nie
islower(c)	*true*, jeśli *c* jest małą literą, *false* jeśli nie
isdigit(c)	*true*, jeśli *c* jest cyfrą, *false* jeśli nie
isalnum(c)	*true*, jeśli *c* jest literą lub cyfrą, *false* jeśli nie
isspace(c)	*true*, jeśli *c* jest odstępem, '\n', '\t', *false* jeśli nie
toupper(c)	małe litery zmienia na wielkie
tolower(c)	wielkie litery zmienia na małe

Niektóre funkcje przekształcające i badające znaki <cctype>

W poniższych opisach *c* jest typu *int* lub *unsigned char* łącznie z *EOF*. Wszystkie funkcje zwracają wartość typu *int*.

isalpha(c)	*true*, jeśli *c* jest literą, *false* jeśli nie
isupper(c)	*true*, jeśli *c* jest wielką literą, *false* jeśli nie
islower(c)	*true*, jeśli *c* jest małą literą, *false* jeśli nie
isdigit(c)	*true*, jeśli *c* jest cyfrą, *false* jeśli nie
isalnum(c)	*true*, jeśli *c* jest literą lub cyfrą, *false* jeśli nie
isspace(c)	*true*, jeśli *c* jest odstępem, '\n', '\t', *false* jeśli nie
toupper(c)	małe litery zmienia na wielkie
tolower(c)	wielkie litery zmienia na małe

Nagłówek <chrono>

zawiera deklaracje klas i funkcji bibliotecznych do pomiaru czasu.

Przeładowanie nazwy funkcji

Przeładowanie nazwy funkcji występuje, gdy mamy więcej niż jedną funkcję o tej samej nazwie, w danym zakresie ważności.
Rozróżnienie opiera się na zestawie typów argumentów i ich kolejności.
Typ zwracany przez funkcję nie jest brany pod uwagę.

Przeładowanie nazwy funkcji

Przeładowanie nazwy funkcji występuje, gdy mamy więcej niż jedną funkcję o tej samej nazwie, w danym zakresie ważności.
Rozróżnienie opiera się na zestawie typów argumentów i ich kolejności.
Typ zwracany przez funkcję nie jest brany pod uwagę.

Przeładowanie stosujemy, aby wykonać to samo zadanie z różnymi zestawami argumentów.

Przeładowanie nazwy funkcji

Przeładowanie nazwy funkcji występuje, gdy mamy więcej niż jedną funkcję o tej samej nazwie, w danym zakresie ważności.
Rozróżnienie opiera się na zestawie typów argumentów i ich kolejności.
Typ zwracany przez funkcję nie jest brany pod uwagę.

Przeładowanie stosujemy, aby wykonać to samo zadanie z różnymi zestawami argumentów.

Przeładowanie a domniemanie

Deklaracja funkcji z n domniemanymi argumentami rezerwuje $n+1$ różnych zestawów argumentów. Przeładowanie nie może powtórzyć żadnego z tych zestawów.

Przeładowanie nazwy funkcji

Przeładowanie nazwy funkcji występuje, gdy mamy więcej niż jedną funkcję o tej samej nazwie, w danym zakresie ważności.
Rozróżnienie opiera się na zestawie typów argumentów i ich kolejności.
Typ zwracany przez funkcję nie jest brany pod uwagę.

Przeładowanie stosujemy, aby wykonać to samo zadanie z różnymi zestawami argumentów.

Przeładowanie a domniemanie

Deklaracja funkcji z n domniemanymi argumentami rezerwuje $n+1$ różnych zestawów argumentów. Przeładowanie nie może powtórzyć żadnego z tych zestawów.

Kompilator języka C++ rozszerza nazwy funkcji o znaki związane z listą argumentów tej funkcji. W ten sposób rozróżniane są przeładowane nazwy.

W innych językach nazwy funkcji nie muszą być modyfikowane.
Gdy dołączamy moduł napisany w innym języku i zamierzamy korzystać z funkcji zawartych w tym module, deklaracja takiej funkcji powinna zawierać dodatkowo symbol *"C"*. Oznacza to konwencję jak w klasycznym języku C, gdzie nazwy funkcji nie są modyfikowane.

W innych językach nazwy funkcji nie muszą być modyfikowane.
Gdy dołączamy moduł napisany w innym języku i zamierzamy korzystać z funkcji zawartych w tym module, deklaracja takiej funkcji powinna zawierać dodatkowo symbol *"C"*. Oznacza to konwencję jak w klasycznym języku C, gdzie nazwy funkcji nie są modyfikowane.

Deklaracje takich funkcji można zapisywać pojedynczo lub zgrupowane w blok. Do takiego bloku można dołączyć cały plik zawierający deklaracje.

```
extern "C" int dodaj(int, int);
extern "C" {
   void swap(int *a, int *b);
   double wyznacznik(double **tab, int rozmiar);
}
extern "C" { #include "macierze.h" }
```

W innych językach nazwy funkcji nie muszą być modyfikowane.
Gdy dołączamy moduł napisany w innym języku i zamierzamy korzystać z funkcji zawartych w tym module, deklaracja takiej funkcji powinna zawierać dodatkowo symbol *"C"*. Oznacza to konwencję jak w klasycznym języku C, gdzie nazwy funkcji nie są modyfikowane.

Deklaracje takich funkcji można zapisywać pojedynczo lub zgrupowane w blok. Do takiego bloku można dołączyć cały plik zawierający deklaracje.

```
extern "C" int dodaj(int, int);
extern "C" {
   void swap(int *a, int *b);
   double wyznacznik(double **tab, int rozmiar);
}
extern "C" { #include "macierze.h" }
```

Przeładowanie nazw funkcji może wystąpić, gdy zakres ważności tych nazw jest ten sam. Identyczna nazwa zakresu lokalnego zasłoni nazwę globalną.

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,
- typy wyliczeniowe są odrębnymi typami, niezależnie od typu ich podwaliny,

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,
- typy wyliczeniowe są odrębnymi typami, niezależnie od typu ich podwaliny,
- za identyczne przy przeładowaniu uznawane są dla danego typu obiektów tablica i wskaźnik do niego,

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,
- typy wyliczeniowe są odrębnymi typami, niezależnie od typu ich podwaliny,
- za identyczne przy przeładowaniu uznawane są dla danego typu obiektów tablica i wskaźnik do niego,
- argumenty tablicowe są rozróżnialne, gdy występuje różnica w ich rozmiarach (za wyjątkiem pierwszego od lewej),

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,
- typy wyliczeniowe są odrębnymi typami, niezależnie od typu ich podwaliny,
- za identyczne przy przeładowaniu uznawane są dla danego typu obiektów tablica i wskaźnik do niego,
- argumenty tablicowe są rozróżnialne, gdy występuje różnica w ich rozmiarach (za wyjątkiem pierwszego od lewej),
- argumenty przesyłane przez wartość i referencję nie są rozróżniane,

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,
- typy wyliczeniowe są odrębnymi typami, niezależnie od typu ich podwaliny,
- za identyczne przy przeładowaniu uznawane są dla danego typu obiektów tablica i wskaźnik do niego,
- argumenty tablicowe są rozróżnialne, gdy występuje różnica w ich rozmiarach (za wyjątkiem pierwszego od lewej),
- argumenty przesyłane przez wartość i referencję nie są rozróżniane,
- dodanie kwalifikatorów *const* i *volatile* nie ma znaczenia ze względu na rozróżnialność – inicjalizatory są identyczne,

Przeładowania nazw a rozróżnianie typów

Kompilator musi odróżnić dwie nazwy przeładowanych funkcji po ich argumentach aktualnych (wywołania) – inicjalizatorze:

- synonimy typów tworzone deklaracjami *typedef* czy *using*, nie są odróżniane jako oddzielne typy,
- typy wyliczeniowe są odrębnymi typami, niezależnie od typu ich podwaliny,
- za identyczne przy przeładowaniu uznawane są dla danego typu obiektów tablica i wskaźnik do niego,
- argumenty tablicowe są rozróżnialne, gdy występuje różnica w ich rozmiarach (za wyjątkiem pierwszego od lewej),
- argumenty przesyłane przez wartość i referencję nie są rozróżniane,
- dodanie kwalifikatorów *const* i *volatile* nie ma znaczenia ze względu na rozróżnialność – inicjalizatory są identyczne,
- kwalifikatory *const* i *volatile* w przypadku wskaźników i referencji, które pracują na oryginale, wymagają każda odmiennego inicjalizatora.

Adres funkcji przeładowanej

jest dobierany na podstawie dopasowania do typu docelowego.

Adres funkcji przeładowanej

jest dobierany na podstawie dopasowania do typu docelowego.

```
int f(int);
int f(double);
int (*wsk_f)(double); // deklaracja wskaźnika do funkcji
wsk_f = f; // lub wsk_f = &f;
```

Adres funkcji przeładowanej

jest dobierany na podstawie dopasowania do typu docelowego.

```
int f(int);
int f(double);
int (*wsk_f)(double); // deklaracja wskaźnika do funkcji
wsk_f = f; // lub wsk_f = &f;
```

```
int (*zwrotWsk(void))(int){
   return f; // zwraca wskaźnik do: int f(int);
}
```

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne
2. dokładne z trywialną konwersją np. z dodatkowym *const*

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne
2. dokładne z trywialną konwersją np. z dodatkowym *const*
3. z awansem np. z *short int* na *int*

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne
2. dokładne z trywialną konwersją np. z dodatkowym *const*
3. z awansem np. z *short int* na *int*
4. z konwersją standardową np. z *int* na *double* lub odwrotnie

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne
2. dokładne z trywialną konwersją np. z dodatkowym *const*
3. z awansem np. z *short int* na *int*
4. z konwersją standardową np. z *int* na *double* lub odwrotnie
5. z konwersją własną z innej klasy

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne
2. dokładne z trywialną konwersją np. z dodatkowym *const*
3. z awansem np. z *short int* na *int*
4. z konwersją standardową np. z *int* na *double* lub odwrotnie
5. z konwersją własną z innej klasy
6. do funkcji z wielokropkiem (dowolność liczby argumentów i typów)

Dopasowanie argumentów do funkcji przeładowanych

jest udane gdy kompilator znajdzie dokładnie jedną z wersji przeładowań, pasujących do wywołania bardziej niż inne. Gdy więcej niż jedna wersja przeładowania jest jednakowo zbliżona do argumentu wywołania, sygnalizowany jest błąd.

Etapy dopasowania wywołanej przeładowanej funkcji:

1. dopasowanie dokładne
2. dokładne z trywialną konwersją np. z dodatkowym *const*
3. z awansem np. z *short int* na *int*
4. z konwersją standardową np. z *int* na *double* lub odwrotnie
5. z konwersją własną z innej klasy
6. do funkcji z wielokropkiem (dowolność liczby argumentów i typów)

Wskaźniki dopasowywane są tylko dosłownie. W wywoływanej funkcji z kilkoma argumentami, na każdym dokonuje się procedura dopasowania.

Preprocesor

Preprocesor jest pierwszym krokiem tłumaczenia programu, zanim kompilator zacznie pracę. Pierwszym czarnym znakiem w linii zawierającej dyrektywę preprocesora jest znak „#”.

Preprocesor

Preprocesor jest pierwszym krokiem tłumaczenia programu, zanim kompilator zacznie pracę. Pierwszym czarnym znakiem w linii zawierającej dyrektywę preprocesora jest znak „#”.

Najczęściej stosowane polecenia to:

- *#include* – wstawia zawartość pewnego pliku podczas kompilacji,
- *#define* – umożliwia zastępowanie pewnego zwrotu dowolnym ciągiem znaków.

Preprocesor

Preprocesor jest pierwszym krokiem tłumaczenia programu, zanim kompilator zacznie pracę. Pierwszym czarnym znakiem w linii zawierającej dyrektywę preprocesora jest znak „#".

Najczęściej stosowane polecenia to:

- *#include* – wstawia zawartość pewnego pliku podczas kompilacji,
- *#define* – umożliwia zastępowanie pewnego zwrotu dowolnym ciągiem znaków.

#include

Wyrażenie *#include "nazwa_pliku"* lub *#include <nazwa_pliku>* zastępowane jest zawartością pliku o podanej nazwie.
W pierwszym przypadku poszukiwanie pliku zaczyna się tam, gdzie znaleziono właściwy program źródłowy. Jeśli go tam nie ma lub gdy *nazwa_pliku* zawarta jest między znakami < i >, pliku szuka się zgodnie z zasadami obowiązującymi w danej implementacji.

#define

Dyrektywa ta powoduje, że dalsze wystąpienia *NAZWA* (za wyjątkiem stałych napisowych) będą zastępowane przez ciąg znaków tworzących *zastępujący tekst*, który może zawierać białe znaki.

```
#define NAZWA zastępujący tekst
```

#define

Dyrektywa ta powoduje, że dalsze wystąpienia *NAZWA* (za wyjątkiem stałych napisowych) będą zastępowane przez ciąg znaków tworzących *zastępujący tekst*, który może zawierać białe znaki.

```
#define NAZWA zastępujący tekst
```

Nazwa w *#define* ma taką samą postać, jak nazwa zmiennej. Zwyczajowo pisana jest wielkimi literami. Zastępujący ją tekst jest dowolny. Długie definicje można kontynuoawć w następnych wierszach po umieszczeniu na końcu przedłużanego wiersza znaku „\". Zasięg nazwy wprowadzonej przez *#define* rozciąga się od miejsca definicji do końca pliku źródłowego. Definicja może korzystać z poprzednich definicji.

#define

Dyrektywa ta powoduje, że dalsze wystąpienia *NAZWA* (za wyjątkiem stałych napisowych) będą zastępowane przez ciąg znaków tworzących *zastępujący tekst*, który może zawierać białe znaki.

```
#define NAZWA zastępujący tekst
```

Nazwa w *#define* ma taką samą postać, jak nazwa zmiennej. Zwyczajowo pisana jest wielkimi literami. Zastępujący ją tekst jest dowolny. Długie definicje można kontynuoawć w następnych wierszach po umieszczeniu na końcu przedłużanego wiersza znaku „\". Zasięg nazwy wprowadzonej przez *#define* rozciąga się od miejsca definicji do końca pliku źródłowego. Definicja może korzystać z poprzednich definicji.

Definicję nazwy można skasować za pomocą polecenia:

```
#undef NAZWA
```

Makrodefinicje

Można definiować *makrodefinicje* z argumentami np.

```
#define max(A,B) ((A)>(B) ? (A) : (B))
```

Wywołanie *max* powoduje wstawienie rozwiniętego tekstu makra bezpośrednio do tekstu programu. *Makrodefinicja* nie może zawierać białych znaków – po takim znaku następuje *rozwinięcie makrodefinicji*.

Makrodefinicje

Można definiować *makrodefinicje* z argumentami np.

```
#define max(A,B) ((A)>(B) ? (A) : (B))
```

Wywołanie *max* powoduje wstawienie rozwiniętego tekstu makra bezpośrednio do tekstu programu. *Makrodefinicja* nie może zawierać białych znaków – po takim znaku następuje *rozwinięcie makrodefinicji*.

Operator

Jeśli w zastępującym tekście parametr sąsiaduje z operatorem ##, to ten parametr zastępuje się aktualnym argumentem, następnie usuwany jest operator ## wraz z otaczającymi go białymi znakami np.

```
#define sklej(nazwa,licznik) int nazwa ## licznik;
```

sklej(kajak,1) daje *int kajak1;*

Operator

Jeśli nazwę parametru w zastępującym tekście poprzedza znak „#”, to ten znak i nazwa parametru zostanie rozwinięta w ciąg znaków ograniczony cudzysłowami. Sam parametr będzie zastąpiony argumentem aktualnym np.

```
#define pokaz(x) cout << #x " = " << (x) << '\n';
```

jest makrem pożytecznym w fazie testowania programu.
pokaz(x/y) daje w wyniku: *cout << "x/y = " << (x/y) << '\n';*

Kompilacja warunkowa

Kompilacja warunkowa pozwala na kompilację pewnego fragmentu kodu pod określonym **warunkiem**. *Warunkiem* jest wyrażenie, w którym każdy element jest stały już w momencie pracy preprocesora.

```
#if warunek
   // linie kompilowane warunkowo
#endif
```

Kompilacja warunkowa

Kompilacja warunkowa pozwala na kompilację pewnego fragmentu kodu pod określonym **warunkiem**. *Warunkiem* jest wyrażenie, w którym każdy element jest stały już w momencie pracy preprocesora.

```
#if warunek
   // linie kompilowane warunkowo
#endif
```

Operator **defined**

używany jest w wyrażeniach warunkowych preprocesora np.
defined NAZWA lub *defined(NAZWA)*
Zwraca wartość *true* w obszarze ważności *NAZWA*, zdefiniowanej operatorem *define*.

```
#if TEST==1 && defined(NAZWA)
   instrukcje
#endif
```

Warunki kompilacji warunkowej można zagnieżdżać.
Można też stosować inne dyrektywy:

```
#if warunek
   instrukcje1
#else
   instrukcje2
#endif
```

```
#if warunek1
   instrukcje1
#elif warunek2
   instrukcje2
#elif warunek3
   instrukcje3
#else
   instrukcje4
#endif
```

Warunki kompilacji warunkowej można zagnieżdżać.
Można też stosować inne dyrektywy:

```
#if warunek
   instrukcje1
#else
   instrukcje2
#endif
```

```
#if warunek1
   instrukcje1
#elif warunek2
   instrukcje2
#elif warunek3
   instrukcje3
#else
   instrukcje4
#endif
```

Dyrektywa *#ifdef NAZWA* oznacza to samo co *#if defined(NAZWA)*, natomiast *#ifndef NAZWA* stosuje się zamiast *#if !defined(NAZWA)*.

Strażnik nagłówka

Aby zagwarantować jednokrotne włączanie danego pliku dyrektywą *#include*, stosuje się w tym pliku konstrukcję zwaną *strażnikiem nagłówka*:

```
#ifndef NAZWAPLIKU
   #define NAZWAPLIKU
   // treść pliku
#endif
```

Nazwę pliku włączanego dyrektywą *#include* można tworzyć przy pomocy makrodefinicji zawierających dyrektywy *#define* i operatora *##*.

Strażnik nagłówka

Aby zagwarantować jednokrotne włączanie danego pliku dyrektywą *#include*, stosuje się w tym pliku konstrukcję zwaną *strażnikiem nagłówka*:

```
#ifndef NAZWAPLIKU
   #define NAZWAPLIKU
   // treść pliku
#endif
```

Nazwę pliku włączanego dyrektywą *#include* można tworzyć przy pomocy makrodefinicji zawierających dyrektywy *#define* i operatora *##*.

#error tekst

przerywa kompilację i wypisuje komunikat o błędzie zawierający stałą napisową *tekst*.

#line stała "nazwaPliku"

Dyrektywa ta powoduje w kompilatorze przypisanie numerowi odczytywanej linii numeru: *stała* – kolejne numery linii będą kolejnymi po tej *stałej*. Jeśli występuje opcjonalna *"nazwaPliku"*, podobnie przypisze nazwie pliku w kompilatorze wartość: *nazwaPliku*.

#line stała "nazwaPliku"

Dyrektywa ta powoduje w kompilatorze przypisanie numerowi odczytywanej linii numeru: *stała* – kolejne numery linii będą kolejnymi po tej *stałej*. Jeśli występuje opcjonalna *"nazwaPliku"*, podobnie przypisze nazwie pliku w kompilatorze wartość: *nazwaPliku*.

Pusta dyrektywa

składająca się ze znaku „#" jako jedynego w linii, jest przez preprocesor ignorowana.

#line stała "nazwaPliku"

Dyrektywa ta powoduje w kompilatorze przypisanie numerowi odczytywanej linii numeru: *stała* – kolejne numery linii będą kolejnymi po tej *stałej*.
Jeśli występuje opcjonalna *"nazwaPliku"*, podobnie przypisze nazwie pliku w kompilatorze wartość: *nazwaPliku*.

Pusta dyrektywa

składająca się ze znaku „#" jako jedynego w linii, jest przez preprocesor ignorowana.

#pragma komenda

jest dyrektywą z niestandardową komendą stosowaną przez dany kompilator. Objaśnień należy szukać w opisie kompilatora.
Jeśli *komenda* jest nieznana danemu kompilatorowi – jest ignorowana.
Można zamiennie stosować operator: *_Pragma(komenda)*
Dyrektywa **#pragma once** na górze pliku pełni rolę strażnika nagłówka.

Predefiniowane nazwy preprocesora

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku
- __DATE__ – data kompilacji pliku

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku
- __DATE__ – data kompilacji pliku
- __TIME__ – czas kompilacji pliku

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku
- __DATE__ – data kompilacji pliku
- __TIME__ – czas kompilacji pliku
- __cplusplus__ – wartość typu long określająca standard C++ pracy kompilatora

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku
- __DATE__ – data kompilacji pliku
- __TIME__ – czas kompilacji pliku
- __cplusplus__ – wartość typu long określająca standard C++ pracy kompilatora
- __STDC_HOSTED__ – wartość 1 gdy kompilujemy w systemie operacyjnym zapewniającym całą bibliotekę standardową, w przeciwnym przypadku wartość 0

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku
- __DATE__ – data kompilacji pliku
- __TIME__ – czas kompilacji pliku
- __cplusplus__ – wartość typu long określająca standard C++ pracy kompilatora
- __STDC_HOSTED__ – wartość 1 gdy kompilujemy w systemie operacyjnym zapewniającym całą bibliotekę standardową, w przeciwnym przypadku wartość 0
- __STDC__ – nakazuje kompilację według klasycznego języka C

Predefiniowane nazwy preprocesora

- __FILE__ – nazwa kompilowanego pliku
- __LINE__ – numer bieżącej linii pliku
- __DATE__ – data kompilacji pliku
- __TIME__ – czas kompilacji pliku
- __cplusplus__ – wartość typu long określająca standard C++ pracy kompilatora
- __STDC_HOSTED__ – wartość 1 gdy kompilujemy w systemie operacyjnym zapewniającym całą bibliotekę standardową, w przeciwnym przypadku wartość 0
- __STDC__ – nakazuje kompilację według klasycznego języka C
- __func__ – zawiera nazwę bieżącej funkcji

Tablice

Tablica jest ciągiem obiektów danego typu, zajmujących ciągły obszar w pamięci.
Tablice są typem złożonym. Rozmiar tablicy musi być stałą całkowitą > 0, znaną w momencie kompilacji czyli typu *constexpr*, np. w definicji: *int a[3];*

Tablice

Tablica jest ciągiem obiektów danego typu, zajmujących ciągły obszar w pamięci.
Tablice są typem złożonym. Rozmiar tablicy musi być stałą całkowitą > 0, znaną w momencie kompilacji czyli typu *constexpr*, np. w definicji: *int a[3];*

Tablica *a* jest blokiem trzech kolejnych obiektów: *a[0]*, *a[1]* i *a[2]*.
Tablice indeksuje się zaczynając od liczby 0.
Próba zapisu do elementu *a[3]* nie jest sygnalizowana jako błąd, jednak niszczy w pamięci coś, co następuje bezpośrednio za tablicą.

Tablice można tworzyć z następujących elementów:

Tablice można tworzyć z następujących elementów:

- typów fundamentalnych (bez void)

Tablice można tworzyć z następujących elementów:

- typów fundamentalnych (bez void)
- typów wyliczeniowych

Tablice można tworzyć z następujących elementów:

- typów fundamentalnych (bez void)
- typów wyliczeniowych
- wskaźników

Tablice można tworzyć z następujących elementów:

- typów fundamentalnych (bez void)
- typów wyliczeniowych
- wskaźników
- innych tablic

Tablice można tworzyć z następujących elementów:

- typów fundamentalnych (bez void)
- typów wyliczeniowych
- wskaźników
- innych tablic
- klas

Tablice można tworzyć z następujących elementów:

- typów fundamentalnych (bez void)
- typów wyliczeniowych
- wskaźników
- innych tablic
- klas
- wskaźników wskazujących składniki klasy

Inicjalizacja tablicy

Tablicę można inicjalizować za pomocą zwykłej operacji przypisania lub za pomocą tzw. *inicjalizacji zbiorczej* np.

```
int b[4] {}; // wszystkie inicjalizowane zerami
int b[4] {1, 2, 3, 4};
int b[4] = {1, 2, 3, 4};
int b[4] = {1, 2}; // pozostałe końcowe inicjalizowane zerami
int b[] = {1, 2, 3, 4}; // tu kompilator sam oblicza rozmiar
```

Gdy lista elementów jest zbyt krótka, pozostałe są inicjalizowane zerami.
Jedynie w przypadku inicjalizacji kompilator sprawdza, czy rozmiar tablicy jest przekroczony, sygnalizując błąd.
Inicjalizacja zbiorcza jest jedynym sposobem inicjalizacji tablicy obiektów stałych *const* lub *constexpr*.
Kompilator sam inicjalizuje zerami tablice zdefiniowane jako obiekty globalne lub statyczne. Tablice lokalne należy zainicjalizować samodzielnie.

Nazwa tablicy jest adresem jej początku (zerowego elementu).
Operator & zwraca adres obiektu.
Adres i-tego elementu tablicy &b[i] można też zapisać jako: b+i.
Tablice przekazywane są poprzez ich adresy jako argumenty funkcji, np.

```
int Suma(int b[], int rozmiar); // deklaracja
Suma(b, 4); // wywołanie
```

Tablice znakowe

C-string

Tekst w tablicy znakowej przechowuje się w postaci ciągu kodów liczbowych liter, po których następuje znak o kodzie 0 – zwany *null*. Taki ciąg liter zakończony znakiem *null* ma nazwę *string* lub *C-string*.

Tablice znakowe

C-string

Tekst w tablicy znakowej przechowuje się w postaci ciągu kodów liczbowych liter, po których następuje znak o kodzie 0 – zwany *null*.
Taki ciąg liter zakończony znakiem *null* ma nazwę *string* lub *C-string*.

Tablicę znakową można zainicjalizować za pomocą *inicjalizacji zbiorczej* np. *char napis[100] = {"dwa"};*
Inicjalizujący tekst ujmuje się w cudzysłów – wtedy w poszczególnych komórkach tablicy znajdują się odpowiednie znaki z dopisanym na końcu ciągu znakiem *null*. Reszta nie wymieniona w cudzysłowach – według zasad inicjalizacji zbiorczej inicjalizowana jest zerami.

Definicja tablicy *char napis[] = {"dwa"};* rezerwuje 4 znaki,
natomiast *char napis[] = { 'd', 'w', 'a'};* rezerwuje 3 znaki.
Cudzysłów sprawia, że kompilator traktuje ujęte w nim znaki jako *C-string*, dopisując na końcu znak o kodzie 0.

Definicja tablicy *char napis[] = {"dwa"};* rezerwuje 4 znaki,
natomiast *char napis[] = { 'd', 'w', 'a'};* rezerwuje 3 znaki.
Cudzysłów sprawia, że kompilator traktuje ujęte w nim znaki jako *C-string*, dopisując na końcu znak o kodzie 0.

Długość *C-stringu* jest to ilość znaków należących do niego.
Jednak rozmiar jest większy o 1 – wliczając znak *null*.

Definicja tablicy *char napis[] = {"dwa"};* rezerwuje 4 znaki,
natomiast *char napis[] = { 'd', 'w', 'a'};* rezerwuje 3 znaki.
Cudzysłów sprawia, że kompilator traktuje ujęte w nim znaki jako *C-string*, dopisując na końcu znak o kodzie 0.

Długość *C-stringu* jest to ilość znaków należących do niego.
Jednak rozmiar jest większy o 1 – wliczając znak *null*.

Do istniejących tablic tekst wpisujemy do każdej komórki oddzielnie.
Można też wykorzystać do tego funkcje zadeklarowane w *<cstring>*.

Definicja tablicy *char napis[] = {"dwa"};* rezerwuje 4 znaki,
natomiast *char napis[] = { 'd', 'w', 'a'};* rezerwuje 3 znaki.
Cudzysłów sprawia, że kompilator traktuje ujęte w nim znaki jako *C-string*, dopisując na końcu znak o kodzie 0.

Długość *C-stringu* jest to ilość znaków należących do niego.
Jednak rozmiar jest większy o 1 – wliczając znak *null*.

Do istniejących tablic tekst wpisujemy do każdej komórki oddzielnie.
Można też wykorzystać do tego funkcje zadeklarowane w *<cstring>*.

```
// kopiuje tablicę znakową t do tablicy s
void Kopiuj(char s[], char t[]){
   int i = 0;
   while(s[i]=t[i]) i++;
}
```

Tablice wielowymiarowe

Tablice wielowymiarowe są tablicami składającymi się z innych tablic.

```
char tabDni[2][13] = {
   {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
   {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

W obiektach typu *char* można przechowywać niewielkie liczby całkowite.

Tablice wielowymiarowe

Tablice wielowymiarowe są tablicami składającymi się z innych tablic.

```
char tabDni[2][13] = {
   {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
   {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

W obiektach typu *char* można przechowywać niewielkie liczby całkowite.

Elementy są umieszczane w pamięci wierszami, a więc skrajnie prawy indeks (nr kolumny) zmienia się najszybciej, wraz z położeniem w pamięci. *tabDni[1]* – oznacza ostatni wiersz powyższej tablicy, czyli adres w pamięci pierwszego elementu tego wiersza.

Inicjalizacja zbiorcza tablic wielowymiarowych

wykonywana jest z grupowaniem wierszy nawiasami klamrowymi lub bez.

Inicjalizacja zbiorcza tablic wielowymiarowych

wykonywana jest z grupowaniem wierszy nawiasami klamrowymi lub bez.

```
int nat[2][3] = {1, 2, 3, 4, 5, 6};

// pozostałe 2 w drugim wierszu są zerowane
int nat[2][3] = {1, 2, 3, 4};

// pozostałe 2 w pierwszym i ostatni w drugim są zerowane
int nat[2][3] = {{1}, {4, 5}};
```

Inicjalizacja zbiorcza tablic wielowymiarowych

wykonywana jest z grupowaniem wierszy nawiasami klamrowymi lub bez.

```
int nat[2][3] = {1, 2, 3, 4, 5, 6};

// pozostałe 2 w drugim wierszu są zerowane
int nat[2][3] = {1, 2, 3, 4};

// pozostałe 2 w pierwszym i ostatni w drugim są zerowane
int nat[2][3] = {{1}, {4, 5}};
```

Dla tablicy obiektów danej klasy w przypadku braku inicjalizatora, zamiast wstawiania zera, zostaje uruchomiony domniemany konstruktor.

Przesyłanie tablic wielowymiarowych do funkcji

polega na przekazywaniu adresu pierwszego elementu danej tablicy reprezentowanego przez jej nazwę, np. w wywołaniu: f(tab);
Gdy funkcji przekazujemy tablicę dwuwymiarową, w jej deklaracji podajemy odpowiednią liczbę kolumn.

Przesyłanie tablic wielowymiarowych do funkcji

polega na przekazywaniu adresu pierwszego elementu danej tablicy reprezentowanego przez jej nazwę, np. w wywołaniu: f(tab);
Gdy funkcji przekazujemy tablicę dwuwymiarową, w jej deklaracji podajemy odpowiednią liczbę kolumn.

Kompilator może obliczyć tylko jeden wymiar tablicy wielowymiarowej: pierwszy od lewej – w deklaracji musimy podać resztę wymiarów.
int f(char tab[2][5][13]); – z pierwszego od lewej wymiaru funkcja nie korzysta w celu obliczenia położenia w pamięci danego elementu w tablicy, ani go nie zna. Więc częściej zapisujemy: *int f(char tab[][5][13]);*

Wskaźniki i adresy

Jednoargumentowy operator & podaje adres obiektu, więc instrukcja

```
p = &c;
```

przypisuje zmiennej *p* adres zmiennej *c*.
p jest wskaźnikiem, który wskazuje na *c*.
Wskaźnik daje adres miejsca w pamięci oraz informację o typie pokazywanego obiektu.

Wskaźniki i adresy

Jednoargumentowy operator & podaje adres obiektu, więc instrukcja

```
p = &c;
```

przypisuje zmiennej *p* adres zmiennej *c*.
p jest wskaźnikiem, który wskazuje na *c*.
Wskaźnik daje adres miejsca w pamięci oraz informację o typie pokazywanego obiektu.

Jednoargumentowy operator * zastosowany do wskaźnika, daje dostęp do obiektu wskazywanego (**p* można stosować wymiennie z *c*).

Wskaźniki i adresy

Jednoargumentowy operator & podaje adres obiektu, więc instrukcja

```
p = &c;
```

przypisuje zmiennej *p* adres zmiennej *c*.
p jest wskaźnikiem, który wskazuje na *c*.
Wskaźnik daje adres miejsca w pamięci oraz informację o typie pokazywanego obiektu.

Jednoargumentowy operator * zastosowany do wskaźnika, daje dostęp do obiektu wskazywanego (**p* można stosować wymiennie z *c*).

Operator adresu & może być stosowany tylko do obiektów zajmujących pamięć: zmiennych i elementów tablic. Nie można go stosować do wyrażeń, stałych, zmiennych *register*, referencji czy pól bitowych.

```
int x = 1, y = 2;
        // p jest wskaźnikiem do obiektu typu int
int *p; // *p ma wartość z int
p = &x; // p wskazuje na x
y = *p; // y ma wartość 1
*p = 0; // x ma wartość 0
auto *py = &y; // py to wskaźnik do obiektu typu int
auto p1 = &x; // p1 to również wskaźnik do int
```

```
int x = 1, y = 2;
        // p jest wskaźnikiem do obiektu typu int
int *p; // *p ma wartość z int
p = &x; // p wskazuje na x
y = *p; // y ma wartość 1
*p = 0; // x ma wartość 0
auto *py = &y; // py to wskaźnik do obiektu typu int
auto p1 = &x; // p1 to również wskaźnik do int
```

```
double *s, *t;
...
s = t; // wskaźniki używane bez adresowania pośredniego
```

```
int x = 1, y = 2;
        // p jest wskaźnikiem do obiektu typu int
int *p; // *p ma wartość z int
p = &x; // p wskazuje na x
y = *p; // y ma wartość 1
*p = 0; // x ma wartość 0
auto *py = &y; // py to wskaźnik do obiektu typu int
auto p1 = &x; // p1 to również wskaźnik do int
```

```
double *s, *t;
...
s = t; // wskaźniki używane bez adresowania pośredniego
```

*(*p)++* nawiasy są niezbędne, aby zwiększyć wskazywany element o 1.
Operacje określone przez jednoargumentowe operatory * i *++*
są wykonywane od prawej strony do lewej.

Operator **reinterpret_cast**

wymusza konwersję między różnymi typami wskaźników, np.

```
double *pa;
int *pn;
pn = reinterpret_cast<int *>(pa);
pn = (int *)pa; // przestarzała metoda
```

Operator **reinterpret_cast**

wymusza konwersję między różnymi typami wskaźników, np.

```
double *pa;
int *pn;
pn = reinterpret_cast<int *>(pa);
pn = (int *)pa; // przestarzała metoda
```

reinterpret_cast pozwala na konwersję wskaźnika na typ całkowity i konwersję odwrotną, np.

```
int adres = 0x05a681c;
double *pa = reinterpret_cast<double *>(adres);
int adres1 = reinterpret_cast<int>(pa);
```

Wskaźnik typu **void ***

pozbawiony jest informacji o typie wskazywanego obiektu. Więc nie można tym wskaźnikiem odczytać wskazanego miejsca, ani przemieszczać po sąsiednich miejscach np. w tablicy.
Stosuje się go głównie w funkcjach, dla których ważny jest tylko adres w pamięci.

Wskaźnik typu **void ***

pozbawiony jest informacji o typie wskazywanego obiektu. Więc nie można tym wskaźnikiem odczytać wskazanego miejsca, ani przemieszczać po sąsiednich miejscach np. w tablicy.
Stosuje się go głównie w funkcjach, dla których ważny jest tylko adres w pamięci.

Wskaźnikowi typu *void** można bez rzutowania przypisać wskaźnik innego typu (za wyjątkiem wskaźnika do obiektu stałego – aby nie stracić informacji o stałości, wskaźnika do funkcji i wskaźnika do składnika klasy). Odwrotne przypisanie wymaga rzutowania.

```
int *pn;
void *pv = pn;
pn = reinterpret_cast<int *>(pv);
```

Wskaźnik zdefiniowany jako *obiekt statyczny* (globalny, lokalny ze specyfikatorem *static*), ma początkową wartość *nullptr* – wskazuje na adres zerowy.
Wskaźniki *lokalne niestatyczne* (automatyczne) są tworzone na stosie, więc nie są inicjalizowane.
Aby poprzez zapomnienie przypadkiem nie zniszczyć czegoś w pamięci, dobrze jest od razu w definicji takiego wskaźnika przypisać odpowiednią wartość.
Jeśli nie wiemy na co ustawić wskaźnik, ustawiamy go na *nullptr* – instrukcje przypisania pod taki zerowy adres są ignorowane.

Wskaźnik zdefiniowany jako *obiekt statyczny* (globalny, lokalny ze specyfikatorem *static*), ma początkową wartość *nullptr* – wskazuje na adres zerowy.
Wskaźniki *lokalne niestatyczne* (automatyczne) są tworzone na stosie, więc nie są inicjalizowane.
Aby poprzez zapomnienie przypadkiem nie zniszczyć czegoś w pamięci, dobrze jest od razu w definicji takiego wskaźnika przypisać odpowiednią wartość.
Jeśli nie wiemy na co ustawić wskaźnik, ustawiamy go na *nullptr* – instrukcje przypisania pod taki zerowy adres są ignorowane.

```
char c;
char *pc = &c;
char *pc = nullptr;
char *pc {}; // również inicjalizacja przez nullptr
if(!pc) cout << "Wskaznik wskazuje na nullptr";
```

Wskaźniki i tablice

Nazwa tablicy adresem

Nazwa tablicy reprezentuje położenie jej elementu początkowego, więc przypisanie *pa = &a[0];* jest równoważne *pa = a;* gdzie *int *pa;*
Jeśli *p* wskazuje na pewien element tablicy, to *p+1* wskazuje na kolejny, *pa+i* wskazuje na *a[i]*, a **(pa+i)* jest zawartością *a[i]*.

Wskaźniki i tablice

Nazwa tablicy adresem

Nazwa tablicy reprezentuje położenie jej elementu początkowego, więc przypisanie *pa = &a[0];* jest równoważne *pa = a;* gdzie *int *pa;*
Jeśli *p* wskazuje na pewien element tablicy, to *p+1* wskazuje na kolejny, *pa+i* wskazuje na *a[i]*, a **(pa+i)* jest zawartością *a[i]*.

Powiększenie wartości wskaźnika o 1 przesuwa go o odpowiednią ilość bajtów, związaną z typem obiektu na który pokazuje.

Wskaźniki i tablice

Nazwa tablicy adresem

Nazwa tablicy reprezentuje położenie jej elementu początkowego, więc przypisanie *pa = &a[0];* jest równoważne *pa = a;* gdzie *int *pa;*
Jeśli *p* wskazuje na pewien element tablicy, to *p+1* wskazuje na kolejny, *pa+i* wskazuje na *a[i]*, a **(pa+i)* jest zawartością *a[i]*.

Powiększenie wartości wskaźnika o 1 przesuwa go o odpowiednią ilość bajtów, związaną z typem obiektu na który pokazuje.

Z drugiej strony, jeśli *p* jest wskaźnikiem, to w wyrażeniach może wystąpić z indeksem, np. *p[i]* jest wtedy równoważne z **(p+i)*
Równoważne są definicje parametrów funkcji *char s[]* i *char *s*
Do funkcji tablicę przesyła się poprzez jej adres czyli nazwę np. *f(s);*

```
// jest <0 dla s<t; 0 dla s=t; >0 dla s>t
int Porownaj(const char *s, const char *t){
   for( ; *s==*t; s++, t++) if(!*s) return 0;
   return *s - *t;
}
```

```
// jest <0 dla s<t; 0 dla s=t; >0 dla s>t
int Porownaj(const char *s, const char *t){
   for( ; *s==*t; s++, t++) if(!*s) return 0;
   return *s - *t;
}
```

```
// zwraca długość tekstu
int DlugoscNapisu(const char *s){
   char *p = s; // podstawienie pod p, nie pod *p
   while(*p) p++;
   return p-s;
}
```

Argumenty wskaźnikowe funkcji

Argumenty funkcji przekazywane są przez wartość, więc funkcja nie ma dostępu do argumentów z którymi została wywołana.
Argumenty wskaźnikowe pozwalają funkcji mieć pośredni dostęp do argumentów operatora adresu.

Argumenty wskaźnikowe funkcji

Argumenty funkcji przekazywane są przez wartość, więc funkcja nie ma dostępu do argumentów z którymi została wywołana.
Argumenty wskaźnikowe pozwalają funkcji mieć pośredni dostęp do argumentów operatora adresu.

Wywołanie i definicja funkcji zamieniającej miejscami 2 elementy:

```
swap(&a, &b);

void swap(int *px, int *py){
    int tmp;
    tmp = *px, *px = *py, *py = tmp;
}
```

Aby zabezpieczyć się przed przypadkową zmianą wartości pokazywanej przez wskaźnik, w definicji funkcji deklarujemy wskaźnik ze specyfikatorem *const*, np. void f(const char *t){ ... }

Aby zabezpieczyć się przed przypadkową zmianą wartości pokazywanej przez wskaźnik, w definicji funkcji deklarujemy wskaźnik ze specyfikatorem *const*, np. void f(const char *t){ ... }

Przesyłanie tablic wielowymiarowych do funkcji

Gdy funkcji przekazujemy tablicę dwuwymiarową, w jej deklaracji podajemy odpowiednią liczbę kolumn.
Funkcji zostaje przekazany wskaźnik do tablicy wierszy.
f(char tabDni[2][13]){ ... } – z liczby wierszy funkcja nie korzysta,
więc częściej zapisujemy: *f(char tabDni[][13]){ ... }*
lub jako wskaźnik do tablicy 13-tu liczb: *f(char (*tabDni)[13]{ ... }*

Między nazwą tablicy a wskaźnikiem różnica polega na tym, że wskaźnik jest zmienną, a nazwa tablicy nie.
Stąd dla nazwy tablicy *a* konstrukcje *a = pa;* oraz *a++;* są *niedozwolone*.

Między nazwą tablicy a wskaźnikiem różnica polega na tym, że wskaźnik jest zmienną, a nazwa tablicy nie.
Stąd dla nazwy tablicy *a* konstrukcje *a = pa;* oraz *a++;* są *niedozwolone*.

Wskaźniki i liczby całkowite nie są wymienne. Zero jest jedynym wyjątkiem. Można je porównać lub przypisać wskaźnikowi, jednak zero nigdy nie jest poprawnym adresem danych.
Wskaźnik i liczba całkowita mogą być dodawane i odejmowane.
Wskaźniki tej samej tablicy mogą być odejmowane od siebie.

Między nazwą tablicy a wskaźnikiem różnica polega na tym, że wskaźnik jest zmienną, a nazwa tablicy nie.
Stąd dla nazwy tablicy *a* konstrukcje *a = pa;* oraz *a++;* są *niedozwolone*.

Wskaźniki i liczby całkowite nie są wymienne. Zero jest jedynym wyjątkiem. Można je porównać lub przypisać wskaźnikowi, jednak zero nigdy nie jest poprawnym adresem danych.
Wskaźnik i liczba całkowita mogą być dodawane i odejmowane.
Wskaźniki tej samej tablicy mogą być odejmowane od siebie.

Jeśli *p* i *q* wskazują na elementy tej samej tablicy wraz z pierwszym elementem po niej, to relacje ==, *!=*, <, >=, itp. działają poprawnie.

Między nazwą tablicy a wskaźnikiem różnica polega na tym, że wskaźnik jest zmienną, a nazwa tablicy nie.
Stąd dla nazwy tablicy *a* konstrukcje *a = pa;* oraz *a++;* są *niedozwolone*.

Wskaźniki i liczby całkowite nie są wymienne. Zero jest jedynym wyjątkiem. Można je porównać lub przypisać wskaźnikowi, jednak zero nigdy nie jest poprawnym adresem danych.
Wskaźnik i liczba całkowita mogą być dodawane i odejmowane.
Wskaźniki tej samej tablicy mogą być odejmowane od siebie.

Jeśli *p* i *q* wskazują na elementy tej samej tablicy wraz z pierwszym elementem po niej, to relacje ==, *!=*, <, >=, itp. działają poprawnie.

Ze wskaźników jako zmiennych można budować tablice.
Deklaracja tablicy wskaźników do znaków: *char *linePtr[ILOSC_LINII];*

Wywołanie i definicja funkcji zamieniającej dwie linie tekstu:

```
SwapLine(linePtr, i, j);

void SwapLine(char *line[], int i, int j){
    char *tmp;
    tmp = line[i], line[i] = line[j], line[j] = tmp;
}
```

Wywołanie i definicja funkcji zamieniającej dwie linie tekstu:

```
SwapLine(linePtr, i, j);

void SwapLine(char *line[], int i, int j){
   char *tmp;
   tmp = line[i], line[i] = line[j], line[j] = tmp;
}
```

Funkcja kopiująca C-string:

```
char *Kopiuj(char *zapis, const char *odczyt){
   char *start = zapis;
   while( (*(zapis++) = *(odczyt++)) ) ;
   return start;
}
...
cout << (Kopiuj(zapisz, "tekst")) << '\n';
```

Rezerwacja pamięci, operatory *new* i *delete*

Operator **new**

Aby utworzyć nowy obiekt, rezerwując pamięć można wykorzystać operator **new**. Np. dla obiektu typu *char* (nazwę posiada tylko wskaźnik do niego):

```
char *wskaznik;
wskaznik = new char;
```

lub krócej

```
char *wskaznik = new char;
auto *wskaznik = new char;
char *wskaznik = {new char};
char *wskaznik {new char};
```

Można za pomocą operatora *new* zainicjalizować obiekt np.

```
char *wskaznik = new char('a');
char *wskaznik = new char{'a'};
char *wskaznik = {new char{'a'}};
char *wskaznik {new char{'a'}};
char *wskaznik {new char{}}; // inicjalizacja zerem '\0'
```

Można za pomocą operatora *new* zainicjalizować obiekt np.

```
char *wskaznik = new char('a');
char *wskaznik = new char{'a'};
char *wskaznik = {new char{'a'}};
char *wskaznik {new char{'a'}};
char *wskaznik {new char{}}; // inicjalizacja zerem '\0'
```

Zmienną jest obiekt posiadający nazwę.
W ten sposób utworzony za pomocą operatora *new* obiekt nie ma nazwy. Stąd nie obowiązują go zwykłe zasady zakresu ważności nazw.
Dostęp do niego mamy tylko przy pomocy wskaźników. Przypisanie takiemu wskaźnikowi innej wartości jest błędem, powodującym utratę dostępu do obiektu.
Obiekty te są *dynamiczne*, więc po utworzeniu nie są inicjalizowane zerami. Są one tworzone w obszarze pamięci, przyznawanym programowi do swobodnego używania (*free store, heap*).

Operator **delete**

Likwidujemy obiekt (zwalniając pamięć) operatorem **delete**, poprzedzającym wskaźnik wskazujący na ten obiekt np.

```
delete wskaznik;
```

Operator **delete**

Likwidujemy obiekt (zwalniając pamięć) operatorem **delete**, poprzedzającym wskaźnik wskazujący na ten obiekt np.

```
delete wskaznik;
```

Tworzenie obiektu stałego

```
const int *pn = new const int{7};
delete pn;
```

Tworzenie i kasowanie tablic

Tworzenie tablic

Tablice tworzymy podając za typem obiektu ich rozmiar ujęty w nawiasy prostokątne *[]* np.

```
int *wskTab = new int[rozmiar];
```

Tworzenie i kasowanie tablic

Tworzenie tablic

Tablice tworzymy podając za typem obiektu ich rozmiar ujęty w nawiasy prostokątne *[]* np.

```
int *wskTab = new int[rozmiar];
```

Inicjalizacja tablic

typów definiowanych przez użytkownika może nastąpić przez konstruktor domniemany, a dla typów wbudowanych np.

```
int *a = new int[10] {3, 2, 1}; // reszta jest wyzerowana
float *g = new float[500] {}; // inicjalizacja zerami
float *g {new float[500] {}};
```

Kasowanie tablic

Likwidujemy tablicę podając za operatorem *delete* dodatkowo *[]* np.

```
delete [] wskTab;
```

Operator *delete* nie zwraca żadnej wartości (jest typu *void*).

Tablice wielowymiarowe

tworzymy jako tablice tablic o z góry podanych stałych wymiarach.
Czyli tylko wymiar pierwszy od lewej może być podany przez zmienną np.

```
double (*tablica)[12][30] = new double[wymiar][12][30];
auto *tablica = new double[wymiar][12][30];
```

Tablice wielowymiarowe

tworzymy jako tablice tablic o z góry podanych stałych wymiarach.
Czyli tylko wymiar pierwszy od lewej może być podany przez zmienną np.

```
double (*tablica)[12][30] = new double[wymiar][12][30];
auto *tablica = new double[wymiar][12][30];
```

Tablice wielowymiarowe likwidujemy jak jednowymiarowe np.

```
delete [] tablica;
```

Tablice wielowymiarowe

tworzymy jako tablice tablic o z góry podanych stałych wymiarach.
Czyli tylko wymiar pierwszy od lewej może być podany przez zmienną np.

```
double (*tablica)[12][30] = new double[wymiar][12][30];
auto *tablica = new double[wymiar][12][30];
```

Tablice wielowymiarowe likwidujemy jak jednowymiarowe np.

```
delete [] tablica;
```

Poważnym błędem jest kasowanie wcześniej skasowanego obiektu
lub przypisanie mu wartości.
Aby tego uniknąć, po kasowaniu można wskaźnik ustawić na *nullptr*
(zabezpiecza to przed omyłkowym skasowaniem):

```
tablica = nullptr;
```

Rezerwacja i kasowanie tablicy 2D

```
int w = 5, k = 20; // ilość wierszy i kolumn
int **tab = new int* [w]; // rezerwacja miejsc na wiersze
for(int i=0; i<w; ++i)
   tab[i] = new int[k]; // rezerwacja wierszy

   ...

for(int i=0; i<w; ++i) delete [] tab[i];
delete [] tab;
tab = nullptr;
```

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

1. zwrot adresu *nullptr* – stary sposób

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

1. zwrot adresu *nullptr* – stary sposób
2. rzucenie wyjątku *bad_alloc* – sposób domniemany

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

1. zwrot adresu *nullptr* – stary sposób
2. rzucenie wyjątku *bad_alloc* – sposób domniemany
3. wywołanie danej przez nas funkcji

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

1. zwrot adresu *nullptr* – stary sposób
2. rzucenie wyjątku *bad_alloc* – sposób domniemany
3. wywołanie danej przez nas funkcji

```
int *wsk = new (std::nothrow) int[wym];
if(!wsk) std::cout << "Rezerwacja nieudana";
```

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

1. zwrot adresu *nullptr* – stary sposób
2. rzucenie wyjątku *bad_alloc* – sposób domniemany
3. wywołanie danej przez nas funkcji

```
int *wsk = new (std::nothrow) int[wym];
if(!wsk) std::cout << "Rezerwacja nieudana";
```

```
try{ int *wsk = new int[wym]; }
catch(std::bad_alloc){ std::cout << "Rezerwacja nieudana"; }
```

W przypadku nieudanej próby rezerwacji pamięci, informację o tym możemy uzyskać na 3 sposoby:

1. zwrot adresu *nullptr* – stary sposób
2. rzucenie wyjątku *bad_alloc* – sposób domniemany
3. wywołanie danej przez nas funkcji

```
int *wsk = new (std::nothrow) int[wym];
if(!wsk) std::cout << "Rezerwacja nieudana";
```

```
try{ int *wsk = new int[wym]; }
catch(std::bad_alloc){ std::cout << "Rezerwacja nieudana"; }
```

```
#include <new>
   ...
set_new_handler(naszaFunkcjaNieudanejRezerwacji);
int *wsk = new int[wym];
```

Umiejscawiający operator *new*

zwykle służy do budowy obiektów zdefiniowanych przez nas klas, które bardzo często powstają i giną. Oszczędzony zostaje wtedy czas na ciągłe rezerwowanie i zwalnianie pamięci.
Może wymagać dołączenia pliku *<new>*.
Zwykle rezerwuje się pamięć w postaci tablicy 1-bajtowego typu *char*.
Do umiejscawiania służy konstrukcja składająca się ze słowa **new**, (wskaźnika) w nawiasach – oznaczającego miejsce utworzenia obiektu w zarezerwowanej wcześniej pamięci oraz typ budowanego obiektu.

Umiejscawiający operator *new*

zwykle służy do budowy obiektów zdefiniowanych przez nas klas, które bardzo często powstają i giną. Oszczędzony zostaje wtedy czas na ciągłe rezerwowanie i zwalnianie pamięci.
Może wymagać dołączenia pliku *<new>*.
Zwykle rezerwuje się pamięć w postaci tablicy 1-bajtowego typu *char*.
Do umiejscawiania służy konstrukcja składająca się ze słowa **new**, (wskaźnika) w nawiasach – oznaczającego miejsce utworzenia obiektu w zarezerwowanej wcześniej pamięci oraz typ budowanego obiektu.

```
#include <new>
   ...
char *miejsce = new char[500]; // zajęcie obszaru pamięci
void *poczatek = &miejsce[0];
naszObiekt *wsk = new (poczatek) naszObiekt[10];
   ...
delete [] miejsce;
```

Wskaźnik stały

inicjalizuje się tylko podczas jego definiowania. Nie można zmienić zapisanego w nim adresu, np.

```
int a;
int *const pa = &a;
```

Wskaźnik stały

inicjalizuje się tylko podczas jego definiowania. Nie można zmienić zapisanego w nim adresu, np.

```
int a;
int *const pa = &a;
```

Wskaźnik do stałego obiektu

wskazywany obiekt uznaje za stały. Nie można modyfikować obiektu, np.

```
const int *pa;
```

Wskaźnik stały

inicjalizuje się tylko podczas jego definiowania. Nie można zmienić zapisanego w nim adresu, np.

```
int a;
int *const pa = &a;
```

Wskaźnik do stałego obiektu

wskazywany obiekt uznaje za stały. Nie można modyfikować obiektu, np.

```
const int *pa;
```

Wskaźnik stały do stałego obiektu

```
const int *const pa = &a;
```

Pierwsze *const* **głębokie** dotyczy obiektu znajdującego się głęboko w pamięci. Drugie *const* **wierzchnie** określa nieruchomość wskaźnika.

Definicja wskaźnika słowem *auto*

```
int k = 12;
const int c_k = 5;
```

Definicja wskaźnika słowem *auto*

```
int k = 12;
const int c_k = 5;
```

Zwykły wskaźnik

```
int *p = &k;
auto *p = &k;
```

Definicja wskaźnika słowem *auto*

```
int k = 12;
const int c_k = 5;
```

Zwykły wskaźnik

```
int *p = &k;
auto *p = &k;
```

Wskaźnik do stałej

```
const int *p = &k;
const auto *p = &k;
auto *p = &c_k;
const auto *p = &c_k;
```

Definicja wskaźnika słowem *auto*

```
int k = 12;
const int c_k = 5;
```

Zwykły wskaźnik

```
int *p = &k;
auto *p = &k;
```

Wskaźnik do stałej

```
const int *p = &k;
const auto *p = &k;
auto *p = &c_k;
const auto *p = &c_k;
```

Wskaźnik stały

```
int * const p = &k;
auto * const p = &k;
```

Wskaźnik stały do stałej

```
const int * const p = &k;
const auto * const p = &k;
auto * const p = &c_k;
const auto * const p = &c_k;
```

Wskaźnik stały do stałej

```
const int * const p = &k;
const auto * const p = &k;
auto * const p = &c_k;
const auto * const p = &c_k;
```

W definicji wskaźnika słowem *auto*, gdy wyrażeniem inicjalizującym jest adres obiektu stałego, kompilator sam utworzy definicję wskaźnika do obiektu stałego.

Wskaźnik stały do stałej

```
const int * const p = &k;
const auto * const p = &k;
auto * const p = &c_k;
const auto * const p = &c_k;
```

W definicji wskaźnika słowem *auto*, gdy wyrażeniem inicjalizującym jest adres obiektu stałego, kompilator sam utworzy definicję wskaźnika do obiektu stałego.

Definicja wskaźnika ze słowem *auto* bez gwiazdki, nie pozwala na zdefiniowanie go jako obiektu stałego.
Kwalifikator *const* w definicji dotyczy typu stojącego bezpośrednio z jego lewej strony, chyba że definicja zaczyna się od *const* – wtedy dotyczy typu stojącego bezpośrednio z prawej.
Kwalifikator *const* stojący obok słowa *auto*, dotyczy całości reprezentowanej tym słowem.

Rodzaje przypisań wartości wskaźnikom:

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)
- *p = tab;* – nazwa tablicy

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)
- *p = tab;* – nazwa tablicy
- *p = &fun;* lub *p = fun;* – nazwa funkcji

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)
- *p = tab;* – nazwa tablicy
- *p = &fun;* lub *p = fun;* – nazwa funkcji
- *p = new int;* – adres właśnie utworzonego obiektu

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)
- *p = tab;* – nazwa tablicy
- *p = &fun;* lub *p = fun;* – nazwa funkcji
- *p = new int;* – adres właśnie utworzonego obiektu
- *p = reinterpret_cast<int *>(0x789abc);* – numeryczne miejsce w pamięci

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)
- *p = tab;* – nazwa tablicy
- *p = &fun;* lub *p = fun;* – nazwa funkcji
- *p = new int;* – adres właśnie utworzonego obiektu
- *p = reinterpret_cast<int *>(0x789abc);* – numeryczne miejsce w pamięci
- *const char *s = "tekst";* – *C-string*. Bez *const* działa jako starsza wersja. Ten sposób nie działa dla innych typów.

Rodzaje przypisań wartości wskaźnikom:

- *int *p = &obiekt;* – przypisanie adresu obiektu
- *p = p1;* – wartość innego wskaźnika tego samego typu
- *p = reinterpret_cast<int *>(q);* – wskaźnik innego typu (nie *const*)
- *void *t = p;* – wartość innego wskaźnika niż *void* (nie *const*)
- *p = tab;* – nazwa tablicy
- *p = &fun;* lub *p = fun;* – nazwa funkcji
- *p = new int;* – adres właśnie utworzonego obiektu
- *p = reinterpret_cast<int *>(0x789abc);* – numeryczne miejsce w pamięci
- *const char *s = "tekst";* – *C-string*. Bez *const* działa jako starsza wersja. Ten sposób nie działa dla innych typów.
- *p = nullptr;* – adres zerowy

Wskaźniki do funkcji

Funkcje w języku C++ nie są zmiennymi, ale można zdefiniować wskaźniki do funkcji. Takim wskaźnikom można nadawać wartości, umieszczać w tablicach, przekazywać do funkcji, zwracać je jako wartość funkcji itp.

Wskaźniki do funkcji

Funkcje w języku C++ nie są zmiennymi, ale można zdefiniować wskaźniki do funkcji. Takim wskaźnikom można nadawać wartości, umieszczać w tablicach, przekazywać do funkcji, zwracać je jako wartość funkcji itp.

```
int fun(); // deklaracja funkcji
int (*w_fun)(); // wskaźnik do funkcji takiej jak wyżej
w_fun = fun; // lub w_fun = &fun;
(*w_fun)(); // lub w_fun(); - wywołanie wskazanej funkcji
```

Wskaźniki do funkcji

Funkcje w języku C++ nie są zmiennymi, ale można zdefiniować wskaźniki do funkcji. Takim wskaźnikom można nadawać wartości, umieszczać w tablicach, przekazywać do funkcji, zwracać je jako wartość funkcji itp.

```
int fun(); // deklaracja funkcji
int (*w_fun)(); // wskaźnik do funkcji takiej jak wyżej
w_fun = fun; // lub w_fun = &fun;
(*w_fun)(); // lub w_fun(); - wywołanie wskazanej funkcji
```

Operacje arytmetyczne na wskaźnikach do funkcji są niedozwolone. Wskaźnik do funkcji może być użyty do wskazywania jedynie na funkcję, której typ jest zgodny z typem tego wskaźnika.

```
auto *TypWsk_f = &f;
// lub decltype(&f) TypWsk_f;
// lub using TypWsk_f = decltype(&f);
// lub using TypWsk_f = double (*)(double);
// lub typedef double (*TypWsk_f)(double);
```

```
auto *TypWsk_f = &f;
// lub decltype(&f) TypWsk_f;
// lub using TypWsk_f = decltype(&f);
// lub using TypWsk_f = double (*)(double);
// lub typedef double (*TypWsk_f)(double);
```

```
double pochodna(double (*f)(double), double x){
// lub double pochodna(typWsk_f f, double x){
   double h = 2e-9;
   return (f(x+h) - f(x)) / h;
   // lub return ((*f)(x+h) - (*f)(x)) / h;
}
   ...
   pochodna(f, 3); // wywołanie funkcji
   // lub pochodna(&f, 3);
```

```
void dodaj(int a, int b){ cout << "\nSuma: " << a+b; }
void odejmij(int a, int b){ cout << "\nRoznica: " << a-b; }
void mnoz(int a, int b){ cout << "\nIloczyn: " << a*b; }

int main(){
   void (*wskF[])(int, int) = { &dodaj, &odejmij, &mnoz};
   // decltype(&dodaj) wskF[] = { &dodaj, &odejmij, &mnoz};
   int a=7, b=12;
   cout << "Dzialania na liczbach: " << a << " i " << b;
   for(int i=0; i<3; ++i)
      wskF[i](a, b); // lub (*wskF[i])(a, b);
}
```

Argumenty wywołania programu

W wierszu polecenia wywołującego program można wpisać jego nazwę, a po niej parametry czyli argumenty wywołania (jakiś tekst). Działanie programu rozpoczyna się wywołaniem funkcji *main* z dwoma argumentami:

```
int main( int argc, char *argv[]){ ... }
```

Argumenty wywołania programu

W wierszu polecenia wywołującego program można wpisać jego nazwę, a po niej parametry czyli argumenty wywołania (jakiś tekst). Działanie programu rozpoczyna się wywołaniem funkcji *main* z dwoma argumentami:

```
int main( int argc, char *argv[]){ ... }
```

Pierwszy nazwany **argc** (*argument counter*) jest liczbą argumentów, z jakimi program został wywołany. Drugi **argv** (*argument vector*) jest wskaźnikiem do tablicy zawierającej argumenty w postaci *C-stringów*. Wartością *argv[0]* jest nazwa wywołanego programu wraz ze ścieżką dostępu, a *argv[argc]* jest wskaźnikiem pustym, tj. równym *nullptr*.

Struktury

Struktura jest obiektem złożonym z jednej lub kilku zmiennych, których typy mogą się różnić, zgrupowanych pod jedną nazwą.

Struktury

Struktura jest obiektem złożonym z jednej lub kilku zmiennych, których typy mogą się różnić, zgrupowanych pod jedną nazwą.

```
struct punkt {int x; int y; } a, b, c;
```

Słowo kluczowe **struct** rozpoczyna deklarację struktury, którą tworzy lista deklaracji zmiennych zawartych między nawiasami klamrowymi.
Zmienne te nazywa się składowymi struktury.
a, *b*, *c* są opcjonalnymi zmiennymi zdefiniowanego typu *struct punkt*.
Po słowie kluczowym *struct* może występować opcjonalna nazwa, zwana etykietą struktury (tutaj *punkt*). Etykieta identyfikuje dany wzorzec struktury, używany w dalszych definicjach struktur.

Struktury

Struktura jest obiektem złożonym z jednej lub kilku zmiennych, których typy mogą się różnić, zgrupowanych pod jedną nazwą.

```
struct punkt {int x; int y; } a, b, c;
```

Słowo kluczowe **struct** rozpoczyna deklarację struktury, którą tworzy lista deklaracji zmiennych zawartych między nawiasami klamrowymi.
Zmienne te nazywa się składowymi struktury.
a, *b*, *c* są opcjonalnymi zmiennymi zdefiniowanego typu *struct punkt*.
Po słowie kluczowym *struct* może występować opcjonalna nazwa, zwana etykietą struktury (tutaj *punkt*). Etykieta identyfikuje dany wzorzec struktury, używany w dalszych definicjach struktur.

Można zdefiniować zmienną jako strukturę tego typu:

```
struct punkt pt = {320, 200};
```

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Dozwolonymi operacjami dla struktury są:

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Dozwolonymi operacjami dla struktury są:

- przypisanie jej innej struktury w całości

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Dozwolonymi operacjami dla struktury są:

- przypisanie jej innej struktury w całości
- pobranie jej adresu za pomocą operatora &

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Dozwolonymi operacjami dla struktury są:

- przypisanie jej innej struktury w całości
- pobranie jej adresu za pomocą operatora &
- odwołania do jej składowych

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Dozwolonymi operacjami dla struktury są:

- przypisanie jej innej struktury w całości
- pobranie jej adresu za pomocą operatora &
- odwołania do jej składowych
- przesyłanie argumentów funkcjom i zwracanie przez funkcje wartości

Te same nazwy można nadać etykiecie, składowej struktury, jak również innej zmiennej poza strukturą oraz składowym różnych struktur.

Strukturę w deklaracji można zainicjalizować tylko wyrażeniami stałymi. Automatyczną strukturę można również zainicjalizować za pomocą przypisania.

Dozwolonymi operacjami dla struktury są:

- przypisanie jej innej struktury w całości
- pobranie jej adresu za pomocą operatora &
- odwołania do jej składowych
- przesyłanie argumentów funkcjom i zwracanie przez funkcje wartości

Dostęp do składowej struktury uzyskujemy poprzez operator „.”:

```
cout << pt.x << ", " << pt.y;
```

Struktury mogą być zagnieżdżone. Prostokąt może być reprezentowany parą jego przeciwległych wierzchołków:

```
struct prostokat {
    struct punkt pt1;
    struct punkt pt2;
};
```

Struktury mogą być zagnieżdżone. Prostokąt może być reprezentowany parą jego przeciwległych wierzchołków:

```
struct prostokat {
    struct punkt pt1;
    struct punkt pt2;
};
```

Nie dopuszcza się aby struktura zawierała w sobie swoje własne wcielenie, lecz może zawierać wskaźnik do samej siebie.

Struktury i funkcje

Dostęp do struktur poprzez funkcje odbywa się często na 3 sposoby:

Struktury i funkcje

Dostęp do struktur poprzez funkcje odbywa się często na 3 sposoby:

- przekazywanie składników oddzielnie

Struktury i funkcje

Dostęp do struktur poprzez funkcje odbywa się często na 3 sposoby:

- przekazywanie składników oddzielnie
- przekazywanie całej struktury

Struktury i funkcje

Dostęp do struktur poprzez funkcje odbywa się często na 3 sposoby:

- przekazywanie składników oddzielnie
- przekazywanie całej struktury
- przekazywanie wskaźnika do struktury

Struktury i funkcje

Dostęp do struktur poprzez funkcje odbywa się często na 3 sposoby:

- przekazywanie składników oddzielnie
- przekazywanie całej struktury
- przekazywanie wskaźnika do struktury

```
struct punkt UtworzPunkt(int x, int y){
    struct punkt tmp;
    tmp.x = x;
    tmp.y = y;
    return tmp;
}
```

```
struct prostokat ekran;
struct punkt srodek;
struct punkt UtworzPunkt(int, int);

ekran.pt1 = UtworzPunkt(0, 0);
ekran.pt2 = UtworzPunkt(XMAX, YMAX);
srodek = UtworzPunkt((ekran.pt1.x + ekran.pt2.x)/2,
   (ekran.pt1.y + ekran.pt2.y)/2);
struct prostokat r, *pr=&r;
```

Tutaj *pr* jest wskaźnikiem do struktury typu *struct prostokat*, a *(*pr).pt1* oraz *(*pr).pt2* są jej składowymi – nawiasy są konieczne.
Zapis *(*pr).pt1* jest równoważny skróconemu zapisowi *pr->pt1*
Równoważne wyrażenia: *r.pt1.x (r.pt1).x pr->pt1.x (pr->pt1).x*

```
struct prostokat ekran;
struct punkt srodek;
struct punkt UtworzPunkt(int, int);

ekran.pt1 = UtworzPunkt(0, 0);
ekran.pt2 = UtworzPunkt(XMAX, YMAX);
srodek = UtworzPunkt((ekran.pt1.x + ekran.pt2.x)/2,
   (ekran.pt1.y + ekran.pt2.y)/2);
struct prostokat r, *pr=&r;
```

Tutaj *pr* jest wskaźnikiem do struktury typu *struct prostokat*, a *(*pr).pt1* oraz *(*pr).pt2* są jej składowymi – nawiasy są konieczne.
Zapis *(*pr).pt1* jest równoważny skróconemu zapisowi *pr->pt1*
Równoważne wyrażenia: *r.pt1.x (r.pt1).x pr->pt1.x (pr->pt1).x*

Operatory strukturowe: „.” i „->” wraz z nawiasami okrągłymi „*()*” wywołania funkcji i prostokątnymi „*[]*” indeksowania tablicy, znajdują się na szczycie hierarchii priorytetów.

Tablice struktur

Następująca deklaracja strukturowa deklaruje strukturowy typ struct student, definiuje tablicę *grupa1* o elementach będących strukturami tego typu oraz rezerwuje dla nich pamięć:

```
struct student{
    char *imie;
    char *nazwisko;
    int punkty;
} grupa1[NMAX];
```

Tablice struktur

Następująca deklaracja strukturowa deklaruje strukturowy typ struct student, definiuje tablicę *grupa1* o elementach będących strukturami tego typu oraz rezerwuje dla nich pamięć:

```
struct student{
    char *imie;
    char *nazwisko;
    int punkty;
} grupa1[NMAX];
```

Aby zainicjalizować tablicę struktur, po definicji podaje się ujętą w klamry listę wartości początkowych:

```
struct student grupa2[] = {
    "Jan", "Kowlski", 0,
    "Małgorzata", "Nowak", 0
};
```

lub

```
struct student grupa2[] = {
    {"Jan", "Kowlski", 0},
    {"Małgorzata", "Nowak", 0}
};
```

lub

```
struct student grupa2[] = {
    {"Jan", "Kowlski", 0},
    {"Małgorzata", "Nowak", 0}
};
```

Dostęp do składowej struktury w tablicy może być realizowany np. *grupa2[i].punkty*

lub

```
struct student grupa2[] = {
    {"Jan", "Kowlski", 0},
    {"Małgorzata", "Nowak", 0}
};
```

Dostęp do składowej struktury w tablicy może być realizowany np. *grupa2[i].punkty*

Wyznaczenie liczby elementów tablicy struktur:

```
#define NMAX (sizeof grupa1 / sizeof(struct student))
```

lub

```
#define NMAX (sizeof grupa1 / sizeof grupa1[0])
```

Unie

Unia jest zmienną, która w różnych momentach może zawierać obiekty różnych typów i rozmiarów w tym samym miejscu w pamięci.

Unie

Unia jest zmienną, która w różnych momentach może zawierać obiekty różnych typów i rozmiarów w tym samym miejscu w pamięci.

Składnia unii jest wzorowana na strukturach:

```
union u_tag{
    int ival;
    double dval;
    char *sval;
} u;
```

Unie

Unia jest zmienną, która w różnych momentach może zawierać obiekty różnych typów i rozmiarów w tym samym miejscu w pamięci.

Składnia unii jest wzorowana na strukturach:

```
union u_tag{
    int ival;
    double dval;
    char *sval;
} u;
```

Zmienna *u* jest na tyle obszerna, aby pomieścić wartość największego z typów składowych. Unia jest strukturą, w której wszystkie składowe są umieszczone w tym samym miejscu w pamięci, nakładając się.

Można jej przypisać wartość każdego z tych typów, jednak typ wartości pobieranej musi być zgodny z typem ostatnio przypisanej wartości.
Zainicjalizować można ją jedynie wartością o typie jej pierwszej składowej.
Operacje dozwolone dla struktur są dozwolone także dla unii.

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

- **stdio** (*standard input/output*) – istnieje ze względu na kompatybilność z językiem C

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

- **stdio** (*standard input/output*) – istnieje ze względu na kompatybilność z językiem C
- **iostream** (*input/output stream*) – jej założenia weszły w skład obecnego standardu ISO

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

- **stdio** (*standard input/output*) – istnieje ze względu na kompatybilność z językiem C
- **iostream** (*input/output stream*) – jej założenia weszły w skład obecnego standardu ISO

Pliki nagłówkowe

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

- **stdio** (*standard input/output*) – istnieje ze względu na kompatybilność z językiem C
- **iostream** (*input/output stream*) – jej założenia weszły w skład obecnego standardu ISO

Pliki nagłówkowe

- **iostream** – dołącza się w każdym przypadku korzystania z tej biblioteki

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

- **stdio** (*standard input/output*) – istnieje ze względu na kompatybilność z językiem C
- **iostream** (*input/output stream*) – jej założenia weszły w skład obecnego standardu ISO

Pliki nagłówkowe

- **iostream** – dołącza się w każdym przypadku korzystania z tej biblioteki
- **fstream** – przy operacjach we/wy na plikach zewnętrznych

Operacje Wejścia/Wyjścia

Operacje wejścia/wyjścia nie wchodzą w skład definicji języka C++. Operacje te możliwe są dzięki bibliotekom, które dołącza producent danego kompilatora.

Mamy dwie podstawowe biblioteki wejścia/wyjścia:

- **stdio** (*standard input/output*) – istnieje ze względu na kompatybilność z językiem C
- **iostream** (*input/output stream*) – jej założenia weszły w skład obecnego standardu ISO

Pliki nagłówkowe

- **iostream** – dołącza się w każdym przypadku korzystania z tej biblioteki
- **fstream** – przy operacjach we/wy na plikach zewnętrznych
- **sstream** – przy operacjach we/wy na obiektach klasy *string*

Wprowadzanie i wyprowadzanie informacji można traktować jako płynący strumień bajtów. Wczytywanie lub wypisywanie informacji może odbywać się na 2 sposoby poprzez:

Wprowadzanie i wyprowadzanie informacji można traktować jako płynący strumień bajtów. Wczytywanie lub wypisywanie informacji może odbywać się na 2 sposoby poprzez:

- operacje we/wy **binarne** – bajty nie są w żaden sposób interpretowane. Służą do komunikacji z programem lub urządzeniem.

Wprowadzanie i wyprowadzanie informacji można traktować jako płynący strumień bajtów. Wczytywanie lub wypisywanie informacji może odbywać się na 2 sposoby poprzez:

- operacje we/wy **binarne** – bajty nie są w żaden sposób interpretowane. Służą do komunikacji z programem lub urządzeniem.
- operacje we/wy **tekstowe** – strumień przesyła i interpretuje (formatuje) informacje. Służą do pokazania pierwotnie binarnej informacji człowiekowi.

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

- **cout** – powiązany ze standardowym urządzeniem wyjścia (zwykle ekran)

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

- **cout** – powiązany ze standardowym urządzeniem wyjścia (zwykle ekran)
- **cin** – ze standardowym urządzeniem wejścia (zwykle klawiatura)

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

- **cout** – powiązany ze standardowym urządzeniem wyjścia (zwykle ekran)
- **cin** – ze standardowym urządzeniem wejścia (zwykle klawiatura)
- **cerr** – ze standardowym urządzeniem, na które wysyłamy komunikaty o błędach (zwykle ekran). Strumień niebuforowany.

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

- **cout** – powiązany ze standardowym urządzeniem wyjścia (zwykle ekran)
- **cin** – ze standardowym urządzeniem wejścia (zwykle klawiatura)
- **cerr** – ze standardowym urządzeniem, na które wysyłamy komunikaty o błędach (zwykle ekran). Strumień niebuforowany.
- **clog** – jak wyżej, lecz strumień jest buforowany.

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

- **cout** – powiązany ze standardowym urządzeniem wyjścia (zwykle ekran)
- **cin** – ze standardowym urządzeniem wejścia (zwykle klawiatura)
- **cerr** – ze standardowym urządzeniem, na które wysyłamy komunikaty o błędach (zwykle ekran). Strumień niebuforowany.
- **clog** – jak wyżej, lecz strumień jest buforowany.

Powyższe strumienie pracują na zwykłych znakach *char*. Do szerokich znaków *wchar_t* mamy ich odpowiedniki: *wcout*, *wcin*, *wcerr*, *wclog*.

Kompilator definiuje kilka gotowych strumieni (obiektów danych klas).
Po uruchomieniu program zakłada i otwiera te strumienie,
a po zakończeniu strumienie zamykane są automatycznie.

- **cout** – powiązany ze standardowym urządzeniem wyjścia (zwykle ekran)
- **cin** – ze standardowym urządzeniem wejścia (zwykle klawiatura)
- **cerr** – ze standardowym urządzeniem, na które wysyłamy komunikaty o błędach (zwykle ekran). Strumień niebuforowany.
- **clog** – jak wyżej, lecz strumień jest buforowany.

Powyższe strumienie pracują na zwykłych znakach *char*. Do szerokich znaków *wchar_t* mamy ich odpowiedniki: *wcout*, *wcin*, *wcerr*, *wclog*.

W obrębie klasy *ostream* operator „<<” został przeładowany tak,
że odpowiada za wysyłanie informacji do strumienia.
Odwrotny operator „>>” wczytuje informacje ze strumienia.

Domniemania dla standardowych strumieni

Gdy wstawiamy do strumienia wypisywane są:

Domniemania dla standardowych strumieni

Gdy wstawiamy do strumienia wypisywane są:

- w systemie dziesiątkowym – typy wbudowane przechowujące liczby całkowite

Domniemania dla standardowych strumieni

Gdy wstawiamy do strumienia wypisywane są:

- w systemie dziesiątkowym – typy wbudowane przechowujące liczby całkowite
- jako pojedyncze znaki ASCII – typy *char* i *unsigned char*

Domniemania dla standardowych strumieni

Gdy wstawiamy do strumienia wypisywane są:

- w systemie dziesiątkowym – typy wbudowane przechowujące liczby całkowite
- jako pojedyncze znaki ASCII – typy *char* i *unsigned char*
- z dokładnością do 6 miejsc, bez zbędnych zer – liczby typu *float* i *double*

Domniemania dla standardowych strumieni

Gdy wstawiamy do strumienia wypisywane są:

- w systemie dziesiątkowym – typy wbudowane przechowujące liczby całkowite
- jako pojedyncze znaki ASCII – typy *char* i *unsigned char*
- z dokładnością do 6 miejsc, bez zbędnych zer – liczby typu *float* i *double*
- heksadecymalne – wskaźniki (z wyjątkiem *char ** i *unsigned char **)

Domniemania dla standardowych strumieni

Gdy wstawiamy do strumienia wypisywane są:

- w systemie dziesiątkowym – typy wbudowane przechowujące liczby całkowite
- jako pojedyncze znaki ASCII – typy *char* i *unsigned char*
- z dokładnością do 6 miejsc, bez zbędnych zer – liczby typu *float* i *double*
- heksadecymalne – wskaźniki (z wyjątkiem *char ** i *unsigned char **)
- w postaci *C-stringu*, na który pokazują – wskaźniki *char ** i *unsigned char **

Gdy wczytujemy ze strumienia:

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów
- interpretowane jako podane w systemie dziesiątkowym, są typy reprezentujące liczby całkowite

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów
- interpretowane jako podane w systemie dziesiątkowym, są typy reprezentujące liczby całkowite
- między znakiem przed liczbą (+ lub -) a tą liczbą nie może być spacji

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów
- interpretowane jako podane w systemie dziesiątkowym, są typy reprezentujące liczby całkowite
- między znakiem przed liczbą (+ lub -) a tą liczbą nie może być spacji
- wczytywanie liczby całkowitej zakończy się po napotkaniu znaku różnego od cyfry

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów
- interpretowane jako podane w systemie dziesiątkowym, są typy reprezentujące liczby całkowite
- między znakiem przed liczbą (+ lub -) a tą liczbą nie może być spacji
- wczytywanie liczby całkowitej zakończy się po napotkaniu znaku różnego od cyfry
- w liczbie zmiennoprzecinkowej wystąpić może litera oznaczająca wykładnik (e lub E), po niej może być również znak (+ lub -). Wewnątrz nie może być spacji.

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów
- interpretowane jako podane w systemie dziesiątkowym, są typy reprezentujące liczby całkowite
- między znakiem przed liczbą (+ lub -) a tą liczbą nie może być spacji
- wczytywanie liczby całkowitej zakończy się po napotkaniu znaku różnego od cyfry
- w liczbie zmiennoprzecinkowej wystąpić może litera oznaczająca wykładnik (e lub E), po niej może być również znak (+ lub -). Wewnątrz nie może być spacji.
- do tablicy znakowej, wczytywanie zaczyna się od pierwszego czarnego znaku, a kończy na ostatnim czarnym znaku. Nie jest sprawdzane zapełnienie tablicy – nadmiarowe wczytywane znaki niszczą pamięć

Gdy wczytujemy ze strumienia:

- ignorowane są wszystkie białe znaki, poprzedzające każdy z wczytywanych typów
- interpretowane jako podane w systemie dziesiątkowym, są typy reprezentujące liczby całkowite
- między znakiem przed liczbą (+ lub -) a tą liczbą nie może być spacji
- wczytywanie liczby całkowitej zakończy się po napotkaniu znaku różnego od cyfry
- w liczbie zmiennoprzecinkowej wystąpić może litera oznaczająca wykładnik (e lub E), po niej może być również znak (+ lub -). Wewnątrz nie może być spacji.
- do tablicy znakowej, wczytywanie zaczyna się od pierwszego czarnego znaku, a kończy na ostatnim czarnym znaku. Nie jest sprawdzane zapełnienie tablicy – nadmiarowe wczytywane znaki niszczą pamięć
- do obiektu klasy *string*, jak wyżej – tyle, że sam się powiększa

Sterowanie formatem

Zasady formatowania zapisane są w tzw. *flagach stanu formatowania*. Umieszczono je w klasie **ios_base** (*input/output state*). Klasę *ios_base* dziedziczy klasa **ios**, więc możemy także posługiwać się krótszym kwalifikatorem.

Sterowanie formatem

Zasady formatowania zapisane są w tzw. *flagach stanu formatowania*. Umieszczono je w klasie **ios_base** (*input/output state*). Klasę *ios_base* dziedziczy klasa **ios**, więc możemy także posługiwać się krótszym kwalifikatorem.

Typ **fmtflags**

W klasie *ios_base* typ składników przechowujących zasady formatowania został instrukcją *typedef* nazwany **fmtflags**. Klasa *ios* jest podstawową dla klas *istream* i *ostream*.

Sterowanie formatem

Zasady formatowania zapisane są w tzw. *flagach stanu formatowania*. Umieszczono je w klasie **ios_base** (*input/output state*). Klasę *ios_base* dziedziczy klasa **ios**, więc możemy także posługiwać się krótszym kwalifikatorem.

Typ **fmtflags**

W klasie *ios_base* typ składników przechowujących zasady formatowania został instrukcją *typedef* nazwany **fmtflags**. Klasa *ios* jest podstawową dla klas *istream* i *ostream*.

Flagi zawierają informacje typu **tak**/nie. Domniemane ustawienia wymienionych dalej flag są wytłuszczone:

- **skipws** – ignoruj białe znaki

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)
- *showpoint* – pokaż kropkę dziesiętną i nieznaczące zera

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)
- *showpoint* – pokaż kropkę dziesiętną i nieznaczące zera
- *uppercase* – wielkie litery w liczbach (e lub x)

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)
- *showpoint* – pokaż kropkę dziesiętną i nieznaczące zera
- *uppercase* – wielkie litery w liczbach (e lub x)
- *showpos* – znak „+” przed liczbami dodatnimi

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)
- *showpoint* – pokaż kropkę dziesiętną i nieznaczące zera
- *uppercase* – wielkie litery w liczbach (e lub x)
- *showpos* – znak „+” przed liczbami dodatnimi
- *scientific* – notacja wykładnicza

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)
- *showpoint* – pokaż kropkę dziesiętną i nieznaczące zera
- *uppercase* – wielkie litery w liczbach (e lub x)
- *showpos* – znak „+” przed liczbami dodatnimi
- *scientific* – notacja wykładnicza
- *fixed* – notacja dziesiętna

- **skipws** – ignoruj białe znaki
- *left* – justowanie lewe (1 z 3 kolejnych może być ustawiona)
- **right** – justowanie prawe
- *internal* – justowanie wewnętrzne
- *boolalpha* – używaj słów *true-false*, zamiast liczb 1-0
- *dec* – konwersja decymalna (ustawia się najwyżej 1 z 3 kolejnych)
- *oct* – konwersja oktalna
- *hex* – konwersja heksadecymalna
- *showbase* – pokaż podstawę konwersji (czyli przed *hex 0x*, *oct 0*)
- *showpoint* – pokaż kropkę dziesiętną i nieznaczące zera
- *uppercase* – wielkie litery w liczbach (e lub x)
- *showpos* – znak „+” przed liczbami dodatnimi
- *scientific* – notacja wykładnicza
- *fixed* – notacja dziesiętna
- *unitbuf* – nie buforuj strumienia

Dla ułatwienia justowania, konwersji i notacji zdefiniowano grupy flag **maski** (**pola**):

Dla ułatwienia justowania, konwersji i notacji zdefiniowano grupy flag **maski** (**pola**):

- *adjustfield* – typ dopasowania: lewe, prawe, wewnętrzne

Dla ułatwienia justowania, konwersji i notacji zdefiniowano grupy flag **maski** (**pola**):

- *adjustfield* – typ dopasowania: lewe, prawe, wewnętrzne
- *basefield* – typ podstawy konwersji: *dec*, *hex*, *oct*

Dla ułatwienia justowania, konwersji i notacji zdefiniowano grupy flag **maski** (**pola**):

- *adjustfield* – typ dopasowania: lewe, prawe, wewnętrzne
- *basefield* – typ podstawy konwersji: *dec*, *hex*, *oct*
- *floatfield* – typ notacji zmiennopozycyjnej: wykładnicza, dziesiętna

Dla ułatwienia justowania, konwersji i notacji zdefiniowano grupy flag **maski (pola)**:

- *adjustfield* – typ dopasowania: lewe, prawe, wewnętrzne
- *basefield* – typ podstawy konwersji: *dec*, *hex*, *oct*
- *floatfield* – typ notacji zmiennopozycyjnej: wykładnicza, dziesiętna

Gdy żadna z flag: *dec*, *oct*, *hex* nie jest ustawiona, jest to równoważne ustawieniu flagi *dec*.

Dla ułatwienia justowania, konwersji i notacji zdefiniowano grupy flag **maski (pola)**:

- *adjustfield* – typ dopasowania: lewe, prawe, wewnętrzne
- *basefield* – typ podstawy konwersji: *dec*, *hex*, *oct*
- *floatfield* – typ notacji zmiennopozycyjnej: wykładnicza, dziesiętna

Gdy żadna z flag: *dec*, *oct*, *hex* nie jest ustawiona, jest to równoważne ustawieniu flagi *dec*.

Format wypisywanych liczb zmiennoprzecinkowych zależy od obowiązującej *dokładności*. Domniemana dokładność to 6 cyfr. Notacja wykładnicza (ustawiona *scientific*) składa się z: jednej cyfry, ewentualnej kropki dziesiętnej, dalszej części liczby ograniczonej przez dokładność, litery „*e*" i wykładnika 10 dla tej liczby.

Przy ustawionej *fixed*, dokładność to ilość cyfr po kropce.
Jeśli nie jest ustawiona żadna z flag: *fixed*, *scientific* – liczby wypisywane są zależnie od ich wartości: w notacji wykładniczej gdy wykładnik jest mniejszy od -4 lub większy od dokładności, w przeciwnym przypadku w dziesiętnej z całkowitą liczbą cyfr nie przekraczającą dokładności.

Przy ustawionej *fixed*, dokładność to ilość cyfr po kropce.
Jeśli nie jest ustawiona żadna z flag: *fixed*, *scientific* – liczby wypisywane są zależnie od ich wartości: w notacji wykładniczej gdy wykładnik jest mniejszy od -4 lub większy od dokładności, w przeciwnym przypadku w dziesiętnej z całkowitą liczbą cyfr nie przekraczającą dokładności.

Flagi i maski są publiczne w zakresie klasy *ios_base*, można więc ich używać poza tą klasą, poprzedzając kwalifikatorem zakresu np. *ios_base::oct* lub krócej *ios::oct*.

Klasę *ios_base* dziedziczą również klasy *ostream* i *istream*, a obiektami tych klas są strumienie *cout* i *cin*. Klasa *ios_base* oprócz flag, zawiera też narzędzia do ich ustawiania.

Klasę *ios_base* dziedziczą również klasy *ostream* i *istream*, a obiektami tych klas są strumienie *cout* i *cin*. Klasa *ios_base* oprócz flag, zawiera też narzędzia do ich ustawiania.

Rodzaje narzędzi do modyfikacji parametrów formatowania:

Klasę *ios_base* dziedziczą również klasy *ostream* i *istream*, a obiektami tych klas są strumienie *cout* i *cin*. Klasa *ios_base* oprócz flag, zawiera też narzędzia do ich ustawiania.

Rodzaje narzędzi do modyfikacji parametrów formatowania:

- elementarne funkcje klasy *ios* do ustawiania i kasowania flag: *setf* i *unsetf*

Klasę *ios_base* dziedziczą również klasy *ostream* i *istream*, a obiektami tych klas są strumienie *cout* i *cin*. Klasa *ios_base* oprócz flag, zawiera też narzędzia do ich ustawiania.

Rodzaje narzędzi do modyfikacji parametrów formatowania:

- elementarne funkcje klasy *ios* do ustawiania i kasowania flag: *setf* i *unsetf*
- funkcje klasy *ios* zmieniające parametry np. szerokość, precyzję

Klasę *ios_base* dziedziczą również klasy *ostream* i *istream*, a obiektami tych klas są strumienie *cout* i *cin*. Klasa *ios_base* oprócz flag, zawiera też narzędzia do ich ustawiania.

Rodzaje narzędzi do modyfikacji parametrów formatowania:

- elementarne funkcje klasy *ios* do ustawiania i kasowania flag: *setf* i *unsetf*
- funkcje klasy *ios* zmieniające parametry np. szerokość, precyzję
- manipulatory wykonujące w prosty sposób oba zadania

Funkcja setf

ustawia flagi, które są jej argumentami (jedną lub sumę bitową kilku). Zwraca dotychczasowy stan wszystkich flag formatowania np.

```
cin.setf(ios::skipws | ios:: boolalpha);
```

Funkcja setf

ustawia flagi, które są jej argumentami (jedną lub sumę bitową kilku). Zwraca dotychczasowy stan wszystkich flag formatowania np.

```
cin.setf(ios::skipws | ios:: boolalpha);
```

Przeładowana funkcja *setf* służy do ustawiania flag tworzących pole:

```
fmtflags setf(fmtflags flaga, fmtflags maska);
```

Funkcja kasuje wszystkie flagi, a podaną jako pierwszy argument ustawia. Można nią skasować wszystkie flagi tworzące pole np.

```
cout.setf(0, ios::basefield);
```

Funkcja setf

ustawia flagi, które są jej argumentami (jedną lub sumę bitową kilku). Zwraca dotychczasowy stan wszystkich flag formatowania np.

```
cin.setf(ios::skipws | ios:: boolalpha);
```

Przeładowana funkcja *setf* służy do ustawiania flag tworzących pole:

```
fmtflags setf(fmtflags flaga, fmtflags maska);
```

Funkcja kasuje wszystkie flagi, a podaną jako pierwszy argument ustawia. Można nią skasować wszystkie flagi tworzące pole np.

```
cout.setf(0, ios::basefield);
```

Funkcja unsetf

kasuje flagi. W pełni analogiczna do *setf*.

```
fmtflags unsetf(fmtflags flagi);
```

Funkcja flags

zwraca dotychczasowy stan flag formatowania.
Przypisuje flagi formatowania podane w zestawie jako jej argument.
Jej przeładowanie *flags()* tylko zwraca stan flag.

```
fmtflags flags(fmtflags flagi);
```

Funkcja flags

zwraca dotychczasowy stan flag formatowania.
Przypisuje flagi formatowania podane w zestawie jako jej argument.
Jej przeładowanie *flags()* tylko zwraca stan flag.

```
fmtflags flags(fmtflags flagi);
```

Funkcja width

ustawia minimalną liczbę znaków, w których wypisana będzie zmienna.
Dotyczy to tylko najbliższej operacji we/wy.
Dalej obowiązuje domniemana liczba znaków 0.
Przy wczytywaniu tekstu ustawia maksymalną liczbę jego znaków.
Przeładowana *width()* zwraca obowiązującą szerokość.

```
streamsize width(streamsize);
```

Funkcja fill

zmienia znak wypełniający (domyślnie spacja) na znak, będący jej argumentem. Zwraca dotychczasowy znak wypełniający.
Może pracować również na znakach *wchar_t*.
Przeładowana funkcja *fill()* zwraca znak wypełniający.

```
char fill(char);
```

Funkcja fill

zmienia znak wypełniający (domyślnie spacja) na znak, będący jej argumentem. Zwraca dotychczasowy znak wypełniający.
Może pracować również na znakach *wchar_t*.
Przeładowana funkcja *fill()* zwraca znak wypełniający.

```
char fill(char);
```

Funkcja precision

ustawia dokładność wypisywania liczb zmiennoprzecinkowych. Zwraca dotychczasową wartość dokładności. Domniemana dokładnoś wynosi 6.
Przeładowana funkcja *precision()* zwraca obowiązującą dokładność.

```
streamsize precision(streamsize);
```

Funkcja fill

zmienia znak wypełniający (domyślnie spacja) na znak, będący jej argumentem. Zwraca dotychczasowy znak wypełniający.
Może pracować również na znakach *wchar_t*.
Przeładowana funkcja *fill()* zwraca znak wypełniający.

```
char fill(char);
```

Funkcja precision

ustawia dokładność wypisywania liczb zmiennoprzecinkowych. Zwraca dotychczasową wartość dokładności. Domniemana dokładnoś wynosi 6.
Przeładowana funkcja *precision()* zwraca obowiązującą dokładność.

```
streamsize precision(streamsize);
```

Funkcja copyfmt

kopiuje stan strumienia z innego strumienia np. *stru2.copyfmt(stru1);*

```
ios & copyfmt(ios & strumien);
```

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory parametryzowane

wymagają włączenia pliku *<iomanip>*:

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory parametryzowane

wymagają włączenia pliku *<iomanip>*:

- *setw(int)* – działa jak funkcja *width*

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory parametryzowane

wymagają włączenia pliku *<iomanip>*:

- *setw(int)* – działa jak funkcja *width*
- *setfill(char)* – działa jak funkcja *fill*

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory parametryzowane

wymagają włączenia pliku *<iomanip>*:

- *setw(int)* – działa jak funkcja *width*
- *setfill(char)* – działa jak funkcja *fill*
- *setprecision(int)* – działa jak funkcja *precision*

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory parametryzowane

wymagają włączenia pliku *<iomanip>*:

- *setw(int)* – działa jak funkcja *width*
- *setfill(char)* – działa jak funkcja *fill*
- *setprecision(int)* – działa jak funkcja *precision*
- *setbase(int)* – ustawia podstawę konwersji: 10, 16, 8 lub 0 (domyślna)

Manipulatory

Manipulatory wstawiamy do strumienia tak, jak wstawia się obiekt do wypisania. W ten prosty sposób zmieniamy formatowanie.
Zostały one umieszczone w klasie *ios* i przestrzeni nazw *std*.

Manipulatory parametryzowane

wymagają włączenia pliku *<iomanip>*:

- *setw(int)* – działa jak funkcja *width*
- *setfill(char)* – działa jak funkcja *fill*
- *setprecision(int)* – działa jak funkcja *precision*
- *setbase(int)* – ustawia podstawę konwersji: 10, 16, 8 lub 0 (domyślna)
- *setiosflags(fmtflags)*, *resetiosflags(fmtflags)* – działają jak funkcje *setf* i *unsetf*

Manipulatory bezargumentowe

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*
- *boolalpha*, *noboolalpha* – ustawia i kasuje flagę *boolalpha*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*
- *boolalpha*, *noboolalpha* – ustawia i kasuje flagę *boolalpha*
- *showpoint*, *noshowpoint* – ustawia i kasuje flagę *showpoint*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*
- *boolalpha*, *noboolalpha* – ustawia i kasuje flagę *boolalpha*
- *showpoint*, *noshowpoint* – ustawia i kasuje flagę *showpoint*
- *showpos*, *noshowpos* – ustawia i kasuje flagę *showpos*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*
- *boolalpha*, *noboolalpha* – ustawia i kasuje flagę *boolalpha*
- *showpoint*, *noshowpoint* – ustawia i kasuje flagę *showpoint*
- *showpos*, *noshowpos* – ustawia i kasuje flagę *showpos*
- *unitbuf*, *nounitbuf* – ustawia i kasuje flagę *unitbuf*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*
- *boolalpha*, *noboolalpha* – ustawia i kasuje flagę *boolalpha*
- *showpoint*, *noshowpoint* – ustawia i kasuje flagę *showpoint*
- *showpos*, *noshowpos* – ustawia i kasuje flagę *showpos*
- *unitbuf*, *nounitbuf* – ustawia i kasuje flagę *unitbuf*
- *showbase*, *noshowbase* – ustawia i kasuje flagę *showbase*

Manipulatory bezargumentowe

- *fixed*, *scientific* – ustawiają jedną z tych flag i kasują drugą
- *hex*, *dec*, *oct* – ustawiają odpowiednią flagę i kasują pozostałe dwie
- *left*, *right*, *internal* – ustawiają odpowiednią flagę i kasują pozostałe
- *flush* – powoduje wypisanie zawartości bufora
- *endl* – wstawia do strumienia znak '\n' i wywołuje funkcję *flush()*
- *ends* – wstawia do strumienia znak '\0'
- *ws* – usuwa w buforze białe znaki do pierwszego czarnego
- *skipws*, *noskipws* – ustawia i kasuje flagę *skipws*
- *boolalpha*, *noboolalpha* – ustawia i kasuje flagę *boolalpha*
- *showpoint*, *noshowpoint* – ustawia i kasuje flagę *showpoint*
- *showpos*, *noshowpos* – ustawia i kasuje flagę *showpos*
- *unitbuf*, *nounitbuf* – ustawia i kasuje flagę *unitbuf*
- *showbase*, *noshowbase* – ustawia i kasuje flagę *showbase*
- *uppercase*, *nouppercase* – ustawia i kasuje flagę *uppercase*

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Aby zapisywać (czytać) do pliku:

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Aby zapisywać (czytać) do pliku:

1. definiujemy strumień, tworząc obiekt odpowiedniej klasy np. *ofstream struZapis;*

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Aby zapisywać (czytać) do pliku:

1. definiujemy strumień, tworząc obiekt odpowiedniej klasy np. *ofstream struZapis;*
2. otwieramy konkretny plik np. *struZapis.open("nazwa.txt");*

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Aby zapisywać (czytać) do pliku:

1. definiujemy strumień, tworząc obiekt odpowiedniej klasy np. *ofstream struZapis;*
2. otwieramy konkretny plik np. *struZapis.open("nazwa.txt");*
3. wykonujemy operacje we/wy np. *struZapis << "tekst";*

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Aby zapisywać (czytać) do pliku:

1. definiujemy strumień, tworząc obiekt odpowiedniej klasy np. *ofstream struZapis;*
2. otwieramy konkretny plik np. *struZapis.open("nazwa.txt");*
3. wykonujemy operacje we/wy np. *struZapis << "tekst";*
4. likwidujemy strumień np. *struZapis.close();*

Strumienie wpływające i wypływające z plików

Zapisywać do plików możemy dzięki klasie *ofstream*, odczytywać dzięki *ifstream*. Obie operacje zdefiniowano w *fstream*. Deklaracje związane z tymi klasami umieszczono w pliku nagłówkowym *<fstream>*, a nazwy w przestrzeni nazw *std*.

Aby zapisywać (czytać) do pliku:

1. definiujemy strumień, tworząc obiekt odpowiedniej klasy np. *ofstream struZapis;*
2. otwieramy konkretny plik np. *struZapis.open("nazwa.txt");*
3. wykonujemy operacje we/wy np. *struZapis << "tekst";*
4. likwidujemy strumień np. *struZapis.close();*

Punkty 1 i 2 łączy się często przy pomocy konstruktora klasy np.

```
ofstream struZapis("nazwa.txt");
```

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

- *in* – czytanie, strumień jest klasy *ifstream* lub *fstream*

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

- *in* – czytanie, strumień jest klasy *ifstream* lub *fstream*
- *out* – pisanie, jeśli plik nie istnieje to jest tworzony, jeśli istnieje to kasowana jest jego stara zawartość

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

- *in* – czytanie, strumień jest klasy *ifstream* lub *fstream*
- *out* – pisanie, jeśli plik nie istnieje to jest tworzony, jeśli istnieje to kasowana jest jego stara zawartość
- *ate* – ustawia wskaźnik na końcu

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

- *in* – czytanie, strumień jest klasy *ifstream* lub *fstream*
- *out* – pisanie, jeśli plik nie istnieje to jest tworzony, jeśli istnieje to kasowana jest jego stara zawartość
- *ate* – ustawia wskaźnik na końcu
- *app* – dopisuje na końcu

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

- *in* – czytanie, strumień jest klasy *ifstream* lub *fstream*
- *out* – pisanie, jeśli plik nie istnieje to jest tworzony, jeśli istnieje to kasowana jest jego stara zawartość
- *ate* – ustawia wskaźnik na końcu
- *app* – dopisuje na końcu
- *trunc* – kasuje starą zawartość

Funkcja **open**

otwiera plik. Pierwszy argument to nazwa pliku z ewentualną ścieżką w postaci wskaźnika do *C-stringu*. Drugi określa tryb pracy. Poprzez domniemanie w klasie *ifstream* mamy tryb *in*, a w klasie *ofstream* tryb *out*.

```
void open(char *nazwa, ios_base::openmode tryb);
```

Tryby dostępu do pliku

stosuje się pojedynczo lub kilka jednocześnie:

- *in* – czytanie, strumień jest klasy *ifstream* lub *fstream*
- *out* – pisanie, jeśli plik nie istnieje to jest tworzony, jeśli istnieje to kasowana jest jego stara zawartość
- *ate* – ustawia wskaźnik na końcu
- *app* – dopisuje na końcu
- *trunc* – kasuje starą zawartość
- *binary* – binarny

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Funkcja close

kończy pracę strumienia z plikiem. Jeśli jest to strumień wyjściowy, przedtem wywołana jest funkcja *flush*.

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Funkcja close

kończy pracę strumienia z plikiem. Jeśli jest to strumień wyjściowy, przedtem wywołana jest funkcja *flush*.

Bity stanu błędów

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Funkcja close

kończy pracę strumienia z plikiem. Jeśli jest to strumień wyjściowy, przedtem wywołana jest funkcja *flush*.

Bity stanu błędów

- *goodbit* – bity stanu błędów są wyzerowane

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Funkcja close

kończy pracę strumienia z plikiem. Jeśli jest to strumień wyjściowy, przedtem wywołana jest funkcja *flush*.

Bity stanu błędów

- *goodbit* – bity stanu błędów są wyzerowane
- *eofbit* – ustawiona flaga, gdy nastąpił koniec pliku

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Funkcja close

kończy pracę strumienia z plikiem. Jeśli jest to strumień wyjściowy, przedtem wywołana jest funkcja *flush*.

Bity stanu błędów

- *goodbit* – bity stanu błędów są wyzerowane
- *eofbit* – ustawiona flaga, gdy nastąpił koniec pliku
- *failbit* – po wyzerowaniu flagi, strumień nadaje się do pracy

Tryby *in* i *out* można stosować równocześnie, gdy strumień jest klasy *fstream* np.

```
fstream strum("notki.txt", ios::in | ios::out);
```

Funkcja close

kończy pracę strumienia z plikiem. Jeśli jest to strumień wyjściowy, przedtem wywołana jest funkcja *flush*.

Bity stanu błędów

- *goodbit* – bity stanu błędów są wyzerowane
- *eofbit* – ustawiona flaga, gdy nastąpił koniec pliku
- *failbit* – po wyzerowaniu flagi, strumień nadaje się do pracy
- *badbit* – błąd uniemożliwiający dalszą pracę ze strumieniem

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

- *good()* – zwraca *true*, gdy mamy stan *goodbit*

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

- *good()* – zwraca *true*, gdy mamy stan *goodbit*
- *eof()* – *true*, gdy ustawiona flaga *eofbit*

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

- *good()* – zwraca *true*, gdy mamy stan *goodbit*
- *eof()* – *true*, gdy ustawiona flaga *eofbit*
- *fail()* – *true*, gdy ustawiona flaga *failbit* lub *badbit*

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

- *good()* – zwraca *true*, gdy mamy stan *goodbit*
- *eof()* – *true*, gdy ustawiona flaga *eofbit*
- *fail()* – *true*, gdy ustawiona flaga *failbit* lub *badbit*
- *bad()* – *true*, gdy ustawiona flaga *badbit*

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

- *good()* – zwraca *true*, gdy mamy stan *goodbit*
- *eof()* – *true*, gdy ustawiona flaga *eofbit*
- *fail()* – *true*, gdy ustawiona flaga *failbit* lub *badbit*
- *bad()* – *true*, gdy ustawiona flaga *badbit*

Zamiast *cin.fail();* można stosować instrukcję *!cin;*
Gdy czytanie nie powiedzie się, wartością wyrażenia czytającego ze strumienia jest *NULL*, zamiast referencji do strumienia.

Funkcje operujące na flagach błędu

typ powrotu *bool*, klasa *basic_ios*

- *good()* – zwraca *true*, gdy mamy stan *goodbit*
- *eof()* – *true*, gdy ustawiona flaga *eofbit*
- *fail()* – *true*, gdy ustawiona flaga *failbit* lub *badbit*
- *bad()* – *true*, gdy ustawiona flaga *badbit*

Zamiast *cin.fail();* można stosować instrukcję *!cin;*
Gdy czytanie nie powiedzie się, wartością wyrażenia czytającego ze strumienia jest *NULL*, zamiast referencji do strumienia.

Jeśli w danym strumieniu wystąpi błąd, to dalsze próby wstawiania lub wyjmowania z tego strumienia są ignorowane.

Funkcje pracujące na flagach stanu błędu (klasa *ios_base*)

Funkcje pracujące na flagach stanu błędu (klasa *ios_base*)

- *io_state rdstate()* – zwraca słowo stanu błędu strumienia

Funkcje pracujące na flagach stanu błędu (klasa *ios_base*)

- *io_state rdstate()* – zwraca słowo stanu błędu strumienia
- *void clear(io_state =goodbit)* – zmienia wszystkie flagi błędu, wartość domniemana argumentu to *ios_base::goodbit*

Funkcje pracujące na flagach stanu błędu (klasa *ios_base*)

- *io_state rdstate()* – zwraca słowo stanu błędu strumienia
- *void clear(io_state =goodbit)* – zmienia wszystkie flagi błędu, wartość domniemana argumentu to *ios_base::goodbit*
- *void setstate(io_state)* – ustawia wybrane flagi

Pozycjonowanie wskaźników w pliku

Wskaźniki **get** i **put**

Strumień wejściowy ma wskaźnik do czytania **get**, a wyjściowy do pisania **put** – pokazują one na miejsce pomiędzy dwoma znakami.
Domyślnie *get* znajduje się na początku, a *put* na końcu pliku.

Pozycjonowanie wskaźników w pliku

Wskaźniki **get** i **put**

Strumień wejściowy ma wskaźnik do czytania **get**, a wyjściowy do pisania **put** – pokazują one na miejsce pomiędzy dwoma znakami.
Domyślnie *get* znajduje się na początku, a *put* na końcu pliku.

Funkcje informujące o pozycji wskaźników

Pozycjonowanie wskaźników w pliku

Wskaźniki **get** i **put**

Strumień wejściowy ma wskaźnik do czytania **get**, a wyjściowy do pisania **put** – pokazują one na miejsce pomiędzy dwoma znakami.
Domyślnie *get* znajduje się na początku, a *put* na końcu pliku.

Funkcje informujące o pozycji wskaźników

- *pos_type tellg();* – zwraca pozycję wskaźnika do czytania (*ifstream*)

Pozycjonowanie wskaźników w pliku

Wskaźniki **get** i **put**

Strumień wejściowy ma wskaźnik do czytania **get**, a wyjściowy do pisania **put** – pokazują one na miejsce pomiędzy dwoma znakami.
Domyślnie *get* znajduje się na początku, a *put* na końcu pliku.

Funkcje informujące o pozycji wskaźników

- *pos_type tellg();* – zwraca pozycję wskaźnika do czytania (*ifstream*)
- *pos_type tellp();* – wskaźnika do pisania (funkcja klasy *ofstream*)

Funkcje do pozycjonowania wskaźników **get** i **put**

Funkcje do pozycjonowania wskaźników **get** i **put**

- *istream & seekg(off_type, seek_dir =ios::beg);* – do czytania

Funkcje do pozycjonowania wskaźników **get** i **put**

- *istream & seekg(off_type, seek_dir =ios::beg);* – do czytania
- *ostream & seekp(off_type, seek_dir =ios::beg);* – do pisania

Funkcje do pozycjonowania wskaźników **get** i **put**

- *istream & seekg(off_type, seek_dir =ios::beg);* – do czytania
- *ostream & seekp(off_type, seek_dir =ios::beg);* – do pisania

Pierwszy argument określa pozycję wskazanego bajtu pliku względem miejsca podanego jako drugi argument. Drugi jest zdefiniowany w klasie *ios_base* i przyjmuje wartości: *beg*, *cur*, *end* – czyli pozycje początku, bieżącą i końca pliku.
Wpisywanie w środku pliku powoduje zastępowanie kolejnych bajtów.

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

- *ostringstream* – do wypisania tekstu do obiektu *string*

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

- *ostringstream* – do wypisania tekstu do obiektu *string*
- *istringstream* – do odczytania tekstu z obiektu *string*

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

- *ostringstream* – do wypisania tekstu do obiektu *string*
- *istringstream* – do odczytania tekstu z obiektu *string*

Konstruktory klasy **ostringstream**

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

- *ostringstream* – do wypisania tekstu do obiektu *string*
- *istringstream* – do odczytania tekstu z obiektu *string*

Konstruktory klasy **ostringstream**

- *ostringstream(ios_base::openmode);* – domyślny tryb *ios::out* np. *ostringstream tekst;* lub *ostringstream komunikat(ios::app);*

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

- *ostringstream* – do wypisania tekstu do obiektu *string*
- *istringstream* – do odczytania tekstu z obiektu *string*

Konstruktory klasy **ostringstream**

- *ostringstream(ios_base::openmode);* – domyślny tryb *ios::out* np. *ostringstream tekst;* lub *ostringstream komunikat(ios::app);*
- *ostringstream(string const &, ios_base openmode);* – pierwszy argument to tekst inicjalizujący tworzony obiekt, a drugi jak wyżej np. *ostringstream tekst("*****");*
  lub *ostringstream komunikat("*****", ios::app);*

Zapis i odczyt z obiektu klasy *string*

Aby kierować strumień do obiektu klasy *string*, należy włączyć plik nagłówkowy *<sstream>*, zawierający deklaracje odpowiednich klas:

- *ostringstream* – do wypisania tekstu do obiektu *string*
- *istringstream* – do odczytania tekstu z obiektu *string*

Konstruktory klasy **ostringstream**

- *ostringstream(ios_base::openmode);* – domyślny tryb *ios::out* np. *ostringstream tekst;* lub *ostringstream komunikat(ios::app);*
- *ostringstream(string const &, ios_base openmode);* – pierwszy argument to tekst inicjalizujący tworzony obiekt, a drugi jak wyżej np. *ostringstream tekst("*****");*
  lub *ostringstream komunikat("*****", ios::app);*

string str() const; – funkcja ta zwraca obiekt klasy *string*
void str(const string &) const; – funkcja inicjalizuje nowym tekstem

Konstruktory klasy **istringstream**

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później
- *istringstream(string const &);* – inicjalizujemy obiekt tekstem

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później
- *istringstream(string const &);* – inicjalizujemy obiekt tekstem

string str() const; – funkcja zwraca obiekt *string*
void str(const string &) const; – daje strumieniowi nowy tekst

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później
- *istringstream(string const &);* – inicjalizujemy obiekt tekstem

string str() const; – funkcja zwraca obiekt *string*
void str(const string &) const; – daje strumieniowi nowy tekst

Konstruktory klasy **stringstream**

Klasa ta łączy właściwości poprzednich.

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później
- *istringstream(string const &);* – inicjalizujemy obiekt tekstem

string str() const; – funkcja zwraca obiekt *string*
void str(const string &) const; – daje strumieniowi nowy tekst

Konstruktory klasy **stringstream**

Klasa ta łączy właściwości poprzednich.

- *stringstream(ios_base openmode);* – domyślnie *ios::out | ios::in*

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później
- *istringstream(string const &);* – inicjalizujemy obiekt tekstem

string str() const; – funkcja zwraca obiekt *string*
void str(const string &) const; – daje strumieniowi nowy tekst

Konstruktory klasy **stringstream**

Klasa ta łączy właściwości poprzednich.

- *stringstream(ios_base openmode);* – domyślnie *ios::out | ios::in*
- *stringstream(string const &, ios_base openmode);*

Konstruktory klasy **istringstream**

- *istringstream();* – tekst dla strumienia podany jest później
- *istringstream(string const &);* – inicjalizujemy obiekt tekstem

string str() const; – funkcja zwraca obiekt *string*
void str(const string &) const; – daje strumieniowi nowy tekst

Konstruktory klasy **stringstream**

Klasa ta łączy właściwości poprzednich.

- *stringstream(ios_base openmode);* – domyślnie *ios::out | ios::in*
- *stringstream(string const &, ios_base openmode);*

string str() const;
void str(const string &) const;

Klasy

Klasa jest pewnego rodzaju typem danych. Definicja klasy składa się ze słowa kluczowego **class**, nazwy danej klasy, ciała klasy ujętego w nawiasy klamrowe i średnika.

Klasy

Klasa jest pewnego rodzaju typem danych. Definicja klasy składa się ze słowa kluczowego **class**, nazwy danej klasy, ciała klasy ujętego w nawiasy klamrowe i średnika.

```
class ksiazka{
   public:
      char tytul[200];
      int iloscStron;
      float cena;
      void Wartosc(float liczba);
};
```

Klasy

Klasa jest pewnego rodzaju typem danych. Definicja klasy składa się ze słowa kluczowego **class**, nazwy danej klasy, ciała klasy ujętego w nawiasy klamrowe i średnika.

```
class ksiazka{
   public:
      char tytul[200];
      int iloscStron;
      float cena;
      void Wartosc(float liczba);
};
```

```
void ksiazka::Wartosc(float liczba){
   cena = liczba;
}
```

Obiekty danej klasy tworzymy wypisując nazwę klasy, następnie nazwę obiektu i średnik – analogicznie jak przy deklaracji z typami wbudowanymi. Dostęp do składników klasy otrzymujemy pisząc po nazwie obiektu (lub referencji do obiektu), kropkę i nazwę jej składnika. Dostęp uzyskujemy również pisząc po nazwie wskaźnika do obiektu, strzałkę „->” i nazwę składnika klasy – podobnie jak w strukturach.

Obiekty danej klasy tworzymy wypisując nazwę klasy, następnie nazwę obiektu i średnik – analogicznie jak przy deklaracji z typami wbudowanymi. Dostęp do składników klasy otrzymujemy pisząc po nazwie obiektu (lub referencji do obiektu), kropkę i nazwę jej składnika. Dostęp uzyskujemy również pisząc po nazwie wskaźnika do obiektu, strzałkę „->” i nazwę składnika klasy – podobnie jak w strukturach.

```
ksiazka historia;
ksiazka &refHistoria = historia;
ksiazka *wskHistoria = &historia;
historia.iloscStron = 324;
wskHistoria->Wartosc(44.99);
```

Funkcja składowa klasy może być zdefiniowana na dwa sposoby:

Funkcja składowa klasy może być zdefiniowana na dwa sposoby:

- wewnątrz definicji klasy – wtedy jej typ jest interpretowany jako *inline* (w praktyce dla krótkich definicji)

Funkcja składowa klasy może być zdefiniowana na dwa sposoby:

- wewnątrz definicji klasy – wtedy jej typ jest interpretowany jako *inline* (w praktyce dla krótkich definicji)
- wewnątrz definicji klasy jest tylko deklaracja, a definicja jest poza klasą – wtedy przed jej nazwą występuje nazwa klasy i operator zakresu „::”.

Funkcja składowa klasy może być zdefiniowana na dwa sposoby:

- wewnątrz definicji klasy – wtedy jej typ jest interpretowany jako *inline* (w praktyce dla krótkich definicji)
- wewnątrz definicji klasy jest tylko deklaracja, a definicja jest poza klasą – wtedy przed jej nazwą występuje nazwa klasy i operator zakresu „::”.

W skład ciała klasy mogą wchodzić m.in. dane i funkcje składowe, jak również obiekty innych klas. Nazwy deklarowane w klasie mają zakres ważności obejmujący całą klasę.

Kapsułowanie

odzwierciedla nasze codzienne myślenie o obiektach. Zdefiniowanie klasy jakby zamyka w kapsule dane i funkcje, wcześniej nie związane ze sobą formalnie.

Kapsułowanie

odzwierciedla nasze codzienne myślenie o obiektach. Zdefiniowanie klasy jakby zamyka w kapsule dane i funkcje, wcześniej nie związane ze sobą formalnie.

W programowaniu orientowanym obiektowo nie posługujemy się tylko prostymi obiektami, lecz tworzymy klasy, dzięki którym używamy obiektów o nazwach odpowiadających pojęciom ze świata rzeczywistego.

Inicjalizacja w klasie

oznacza wstępne nadanie wartości składnikowi jeszcze w definicji klasy.

Inicjalizacja w klasie

oznacza wstępne nadanie wartości składnikowi jeszcze w definicji klasy.

```
class kolo{
    int srednica {50}; // lub int srednica = 50;
    int kolor {0xffff00}; // żółty
}
```

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,
- **protected** – jak *private*, ale również dla klas pochodnych od danej.

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,
- **protected** – jak *private*, ale również dla klas pochodnych od danej.

Etykiety można stawiać w dowolnej kolejności i je powtarzać. Dana etykieta obowiązuje do wystąpienia następnej. Poprzez domniemanie składniki klasy posiadają rodzaj dostępu *private*, dopóki nie wystąpi etykieta.

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,
- **protected** – jak *private*, ale również dla klas pochodnych od danej.

Etykiety można stawiać w dowolnej kolejności i je powtarzać. Dana etykieta obowiązuje do wystąpienia następnej. Poprzez domniemanie składniki klasy posiadają rodzaj dostępu *private*, dopóki nie wystąpi etykieta.

Praktyczne ustawienie kolejności etykiet:

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,
- **protected** – jak *private*, ale również dla klas pochodnych od danej.

Etykiety można stawiać w dowolnej kolejności i je powtarzać. Dana etykieta obowiązuje do wystąpienia następnej. Poprzez domniemanie składniki klasy posiadają rodzaj dostępu *private*, dopóki nie wystąpi etykieta.

Praktyczne ustawienie kolejności etykiet:

- **public** – na górze, ważne dla użytkownika

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,
- **protected** – jak *private*, ale również dla klas pochodnych od danej.

Etykiety można stawiać w dowolnej kolejności i je powtarzać. Dana etykieta obowiązuje do wystąpienia następnej. Poprzez domniemanie składniki klasy posiadają rodzaj dostępu *private*, dopóki nie wystąpi etykieta.

Praktyczne ustawienie kolejności etykiet:

- **public** – na górze, ważne dla użytkownika
- **protected** – dostępne w ograniczonym zakresie

Etykiety

Przed składnikami klasy mogą występować etykiety zakończone dwukropkiem, określające rodzaj dostępu do składników:

- **private** – dostępne tylko z wnętrza klasy, poprzez funkcje tej klasy i funkcje zaprzyjaźnione z tą klasą,
- **public** – dostępne z wnętrza i spoza klasy,
- **protected** – jak *private*, ale również dla klas pochodnych od danej.

Etykiety można stawiać w dowolnej kolejności i je powtarzać. Dana etykieta obowiązuje do wystąpienia następnej. Poprzez domniemanie składniki klasy posiadają rodzaj dostępu *private*, dopóki nie wystąpi etykieta.

Praktyczne ustawienie kolejności etykiet:

- **public** – na górze, ważne dla użytkownika
- **protected** – dostępne w ograniczonym zakresie
- **private** – najniżej, ważne dla twórcy klasy

Wskaźnik **this**

wskazujący na obiekt danej klasy, odbierany jest przez funkcję wywołaną dla tego obiektu. Postawiony przed składnikiem klasy, powoduje wykonanie operacji związanej z tym składnikiem na wywołanym obiekcie.
Może być podawany innym funkcjom jako argument o typie wskaźnika do bieżącego obiektu.

Wskaźnik this

wskazujący na obiekt danej klasy, odbierany jest przez funkcję wywołaną dla tego obiektu. Postawiony przed składnikiem klasy, powoduje wykonanie operacji związanej z tym składnikiem na wywołanym obiekcie.
Może być podawany innym funkcjom jako argument o typie wskaźnika do bieżącego obiektu.

Zakres ważności

Nazwy składników klasy mają zakres ważności całej klasy, czyli wewnątrz klasy zasłaniają elementy zewnętrzne o tej samej nazwie.
Dostęp do zmiennej globalnej o nazwie tej samej co składnik klasy otrzymujemy poprzedzając jej nazwę operatorem zakresu.
Wewnątrz funkcji danej klasy dostęp do zasłoniętego składnika tej klasy otrzymujemy poprzedzając jego nazwę, nazwą klasy i operatorem zakresu.

Wskaźnik this

wskazujący na obiekt danej klasy, odbierany jest przez funkcję wywołaną dla tego obiektu. Postawiony przed składnikiem klasy, powoduje wykonanie operacji związanej z tym składnikiem na wywołanym obiekcie.
Może być podawany innym funkcjom jako argument o typie wskaźnika do bieżącego obiektu.

Zakres ważności

Nazwy składników klasy mają zakres ważności całej klasy, czyli wewnątrz klasy zasłaniają elementy zewnętrzne o tej samej nazwie.
Dostęp do zmiennej globalnej o nazwie tej samej co składnik klasy otrzymujemy poprzedzając jej nazwę operatorem zakresu.
Wewnątrz funkcji danej klasy dostęp do zasłoniętego składnika tej klasy otrzymujemy poprzedzając jego nazwę, nazwą klasy i operatorem zakresu.

Nazwy zasłaniane są bez rozróżnienia nazw funkcji czy zmiennych.
Aby klasa była samowystarczalna, nie powinna korzystać ze zmiennych globalnych bezpośrednio.

Każda klasa w oddzielnym pliku

W rozbudowanym programie używa się wielu klas, więc ze względu na przejrzystość i możliwość późniejszego ich zastosowania w innych programach, klasy powinno się umieszczać w osobnych plikach nagłówkowych (z rozszerzeniem „*.h”) o nazwie klasy.

Każda klasa w oddzielnym pliku

W rozbudowanym programie używa się wielu klas, więc ze względu na przejrzystość i możliwość późniejszego ich zastosowania w innych programach, klasy powinno się umieszczać w osobnych plikach nagłówkowych (z rozszerzeniem „*.h”) o nazwie klasy.

Funkcje definiowane wewnątrz klasy

są traktowane jako *inline*. Jeśli definicję funkcji *inline* piszemy poza definicją jej klasy, poprzedzamy ją słowem *inline* oraz piszemy tę definicję zaraz pod definicją klasy, w tym samym pliku nagłówkowym.

Każda klasa w oddzielnym pliku

W rozbudowanym programie używa się wielu klas, więc ze względu na przejrzystość i możliwość późniejszego ich zastosowania w innych programach, klasy powinno się umieszczać w osobnych plikach nagłówkowych (z rozszerzeniem „*.h”) o nazwie klasy.

Funkcje definiowane wewnątrz klasy

są traktowane jako *inline*. Jeśli definicję funkcji *inline* piszemy poza definicją jej klasy, poprzedzamy ją słowem *inline* oraz piszemy tę definicję zaraz pod definicją klasy, w tym samym pliku nagłówkowym.

Aby uniknąć wstawiania do pliku źródłowego łącznie z plikiem nagłówkowym klasy, dyrektywy używania przestrzeni nazw, należy unikać wstawiania jej do pliku nagłówkowego klasy.

Strażnik nagłówka

W celu zabezpieczenia przed omyłkowym wstawieniem tego pliku dwukrotnie, całą zawartość pliku obejmujemy konstrukcją zwaną *strażnikiem nagłówka*:

Strażnik nagłówka

W celu zabezpieczenia przed omyłkowym wstawieniem tego pliku dwukrotnie, całą zawartość pliku obejmujemy konstrukcją zwaną *strażnikiem nagłówka*:

```
#ifndef NAZWA_H
#define NAZWA_H
    ...
#endif
```

Strażnik nagłówka

W celu zabezpieczenia przed omyłkowym wstawieniem tego pliku dwukrotnie, całą zawartość pliku obejmujemy konstrukcją zwaną *strażnikiem nagłówka*:

```
#ifndef NAZWA_H
#define NAZWA_H
    ...
#endif
```

. . . a zwykle w dwóch plikach

Definicje funkcji składowych klasy, nie będących typu *inline*, umieszczamy w zwykłych plikach (z rozszerzeniem „*.cpp”) o nazwie klasy, ponieważ ich definicje mogą tylko raz wystąpić w programie.
Definicję statycznej składowej klasy umieszcza się na górze pliku źródłowego, a jej deklarację w definicji klasy.
Instrukcją *#include* włączamy do takiego pliku plik nagłówkowy klasy.

Obiekt jako argument funkcji

Obiekty tak samo jak dane typów prostych, przesyłane są do funkcji przez wartość – czyli funkcja tworzy kopię tego obiektu i na niej pracuje.
Aby uniknąć kopiowania dużych obiektów, można przesyłać je przez referencje. Pracujemy wtedy na oryginale obiektu. W deklaracji funkcji jej argument poprzedza się wtedy dodatkowo znakiem „&”.

Konstruktor klasy

W trakcie definiowania obiektu przydziela się mu miejsce w pamięci, a następnie jeśli klasa ma odpowiedni konstruktor, to jest on automatycznie uruchamiany. Konstruktor służy do nadawania wartości początkowej tworzonemu obiektowi.

Konstruktor klasy

W trakcie definiowania obiektu przydziela się mu miejsce w pamięci, a następnie jeśli klasa ma odpowiedni konstruktor, to jest on automatycznie uruchamiany. Konstruktor służy do nadawania wartości początkowej tworzonemu obiektowi.

Inicjalizacja "w klasie" jest ignorowana, gdy dany składnik ma być inicjalizowany konstruktorem.
Występowanie konstruktora jest opcjonalne.

Konstruktor spełnia poniższe warunki:

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,
- nie można posłużyć się jego adresem,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,
- nie można posłużyć się jego adresem,
- deklaracja konstruktora nie zawiera słowa oznaczającego typ zwracanej wartości, a definicja jeśli zawiera słowo kluczowe *return*, to z samym średnikiem,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,
- nie można posłużyć się jego adresem,
- deklaracja konstruktora nie zawiera słowa oznaczającego typ zwracanej wartości, a definicja jeśli zawiera słowo kluczowe *return*, to z samym średnikiem,
- możne być wywołany przy tworzeniu obiektów typu *const* czy *volatile*, pomimo braku tych słów w jego deklaracji,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,
- nie można posłużyć się jego adresem,
- deklaracja konstruktora nie zawiera słowa oznaczającego typ zwracanej wartości, a definicja jeśli zawiera słowo kluczowe *return*, to z samym średnikiem,
- możne być wywołany przy tworzeniu obiektów typu *const* czy *volatile*, pomimo braku tych słów w jego deklaracji,
- deklaracja może posiadać słowo *constexpr*,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,
- nie można posłużyć się jego adresem,
- deklaracja konstruktora nie zawiera słowa oznaczającego typ zwracanej wartości, a definicja jeśli zawiera słowo kluczowe *return*, to z samym średnikiem,
- możne być wywołany przy tworzeniu obiektów typu *const* czy *volatile*, pomimo braku tych słów w jego deklaracji,
- deklaracja może posiadać słowo *constexpr*,
- nie może być typu *static*,

Konstruktor spełnia poniższe warunki:

- jest funkcją składową danej klasy o nazwie identycznej z nazwą klasy,
- jest funkcją przy której najczęściej spotyka się przeładowanie nazwy,
- nie można posłużyć się jego adresem,
- deklaracja konstruktora nie zawiera słowa oznaczającego typ zwracanej wartości, a definicja jeśli zawiera słowo kluczowe *return*, to z samym średnikiem,
- możne być wywołany przy tworzeniu obiektów typu *const* czy *volatile*, pomimo braku tych słów w jego deklaracji,
- deklaracja może posiadać słowo *constexpr*,
- nie może być typu *static*,
- nie może być typu *virtual*.

Specyfikator **explicit**

występujący przed deklararacją konstruktora w klasie, nie dopuszcza jego użycia w niejawnych konwersjach. Gdy konstruktor posiada jeden argument i w danym miejscu oczekiwany jest obiekt danej klasy, może nastąpić niejawna konwersja argumentu dla konstruktora w obiekt tej klasy.

Specyfikator **explicit**

występujący przed deklararacją konstruktora w klasie, nie dopuszcza jego użycia w niejawnych konwersjach. Gdy konstruktor posiada jeden argument i w danym miejscu oczekiwany jest obiekt danej klasy, może nastąpić niejawna konwersja argumentu dla konstruktora w obiekt tej klasy.

Konstruktor może być użyty w deklaracji obiektu, pisząc za jego nazwą listę argumentów ujętą w nawiasy:

```
kolor zielony{0x8000}; // zalecany sposób
// lub kolor zielony(0x8000);
```

Specyfikator **explicit**

występujący przed deklararacją konstruktora w klasie, nie dopuszcza jego użycia w niejawnych konwersjach. Gdy konstruktor posiada jeden argument i w danym miejscu oczekiwany jest obiekt danej klasy, może nastąpić niejawna konwersja argumentu dla konstruktora w obiekt tej klasy.

Konstruktor może być użyty w deklaracji obiektu, pisząc za jego nazwą listę argumentów ujętą w nawiasy:

```
kolor zielony{0x8000}; // zalecany sposób
// lub kolor zielony(0x8000);
```

Wywołujemy go również gdy po nazwie obiektu umieścimy operator przypisania i nazwę klasy z listą argumentów w nawiasach:

```
kolor zielony = kolor(0x8000);
// lub kolor zielony = kolor( 0, 0x80, 0);
```

Konstruktor domniemany

to ten, który można wywołać beż żadnego argumentu (czyli również ze wszystkimi argumentami domniemanymi). Używany jest gdy nie wskażemy bezpośrednio danego konstruktora.
Gdy klasa nie ma zdefiniowanego konstruktora, kompilator wygeneruje domniemany pusty konstruktor o cechach *public* i *inline*.

Konstruktor domniemany

to ten, który można wywołać beż żadnego argumentu (czyli również ze wszystkimi argumentami domniemanymi). Używany jest gdy nie wskażemy bezpośrednio danego konstruktora.
Gdy klasa nie ma zdefiniowanego konstruktora, kompilator wygeneruje domniemany pusty konstruktor o cechach *public* i *inline*.

Taki wygenerowany konstruktor przy ustawianiu wartości składników uwzględnia inicjalizatory "w klasie".
Gdy nie ma takich inicjalizatorów, to dla składników będących obiektami innych klas, zostaną wywołane ich domniemane konstruktory.
Składnik typu wbudowanego nie otrzyma żadnej wartości początkowej, a w przypadku gdy jest on statyczny, otrzyma stosownego rodzaju zero.

Dopisek = *default*

w deklaracji konstruktora domniemanego powoduje wygenerowanie takiego konstruktora, pomimo obecności innych konstruktorów w tej klasie.
Przykład dla klasy *K* wewnątrz klasy: *K() = default;*
Drugi sposób wymaga zwykłej deklaracji wewnątrz klasy: *K();*
wraz z deklaracją poza ciałem klasy (z dopiskiem): *K::K() = default;*
Drugi sposób nie nadaje konstruktorowi cechy *inline*, chyba że wewnątrz klasy dodamy słowo *inline*.

Dopisek = *default*

w deklaracji konstruktora domniemanego powoduje wygenerowanie takiego konstruktora, pomimo obecności innych konstruktorów w tej klasie.
Przykład dla klasy *K* wewnątrz klasy: *K() = default;*
Drugi sposób wymaga zwykłej deklaracji wewnątrz klasy: *K();*
wraz z deklaracją poza ciałem klasy (z dopiskiem): *K::K() = default;*
Drugi sposób nie nadaje konstruktorowi cechy *inline*, chyba że wewnątrz klasy dodamy słowo *inline*.

Dopisek *delete*

zabrania kompilatorowi automatyczne generowanie konstruktora, a dopisany w deklaracjach innych funkcji składowych klasy zabrania ich użycia.
W ciele klasy umieszczamy deklarację z dopiskiem, np.: *K() = delete;*
Zastosowanie dopisku *delete* w deklaracji destruktora, uniemożliwia likwidowanie obiektów danej klasy.

Konstruktorowa lista inicjalizacyjna składników klasy

służy do inicjalizacji składników klasy (w tym składników stałych).
W definicji konstruktora przed ciałem klasy, zaraz za nawiasem zamykającym listę jego argumentów stawiamy dwukropek,
po którym oddzielone przecinkami wypisywane są nazwy składników klasy,
a po nich w nawiasach klamrowych wartości inicjalizujące, np.

Konstruktorowa lista inicjalizacyjna składników klasy

służy do inicjalizacji składników klasy (w tym składników stałych).
W definicji konstruktora przed ciałem klasy, zaraz za nawiasem zamykającym listę jego argumentów stawiamy dwukropek,
po którym oddzielone przecinkami wypisywane są nazwy składników klasy,
a po nich w nawiasach klamrowych wartości inicjalizujące, np.

```
class K{
   const int n;
   int p;
   char c = 't';
   double &d;
public:
   K(int a, char z, double &ref) : n{a}, p{8}, d{ref} {
      c = z;
   }
};
```

Każdy konstruktor może mieć inną listę inicjalizacyjną. W defninicji konstruktora poza ciałem klasy również można podać listę inicjalizacyjną. Jeśli składnik zostanie zainicjalizowany z listy inicjalizacyjnej, to jego inicjalizacja w klasie jest ignorowana.

Każdy konstruktor może mieć inną listę inicjalizacyjną. W defninicji konstruktora poza ciałem klasy również można podać listę inicjalizacyjną. Jeśli składnik zostanie zainicjalizowany z listy inicjalizacyjnej, to jego inicjalizacja w klasie jest ignorowana.

Składnik *const* nie może zostać zainicjalizowany w ciele konstruktora, lecz z jego listy inicjalizacyjnej. Podobnie składnik klasy będący referencją musi zostać zainicjalizowany z listy lub "w klasie".
Składnik *static* nie może być inicjalizowany z listy inicjalizacyjnej.

Każdy konstruktor może mieć inną listę inicjalizacyjną. W defninicji konstruktora poza ciałem klasy również można podać listę inicjalizacyjną. Jeśli składnik zostanie zainicjalizowany z listy inicjalizacyjnej, to jego inicjalizacja w klasie jest ignorowana.

Składnik *const* nie może zostać zainicjalizowany w ciele konstruktora, lecz z jego listy inicjalizacyjnej. Podobnie składnik klasy będący referencją musi zostać zainicjalizowany z listy lub "w klasie".
Składnik *static* nie może być inicjalizowany z listy inicjalizacyjnej.

Inicjalizatorem składnika na liście inicjalizacyjnej może być argument konstruktora lub proste wyrażenie arytmetyczne (może zawierać funkcję składową lub zwykłą).
Inicjalizacja składników odbywa się w kolejności deklaracji składników w ciele klasy, niezależnie od kolejności na liście inicjalizacyjnej.

Łapanie wyjątków z listy inicjalizacyjnej konstruktora

Słowo *try* umieszczamy przed dwukropkiem i listą inicjalizacyjną.
W ten sposób instrukcje listy objęte są blokiem *try*. Bloki *catch* położone są bezpośrednio za nawiasem zamykającym ciało konstruktora, np.

```
class K{
   double* wsk = nullptr;
public:
   K(int m) try : wsk{new double[n]}
   { }
   catch(exception &e){
      cerr << "Wyjatek typu " << e.what();
   }
}
```

Łapanie wyjątków z listy inicjalizacyjnej konstruktora

Słowo *try* umieszczamy przed dwukropkiem i listą inicjalizacyjną.
W ten sposób instrukcje listy objęte są blokiem *try*. Bloki *catch* położone są bezpośrednio za nawiasem zamykającym ciało konstruktora, np.

```
class K{
   double* wsk = nullptr;
public:
   K(int m) try : wsk{new double[n]}
   { }
   catch(exception &e){
      cerr << "Wyjatek typu " << e.what();
   }
}
```

Funkcja składowa wyjątku **what** informuje o tym jakiego typu jest wyjątek.
Po zakończeniu bloku *catch* konstruktora, wyjątek rzucany jest dalej do miejsca wywołującego ten konstruktor.

Destruktor klasy

Destruktor jest funkcją składową wywoływaną gdy obiekt danej klasy jest likwidowany, a trzeba przy tej okazji wykonać jakieś działania.
Nie wywołuje się destruktora gdy przestaje istnieć referencja lub wskaźnik do obiektu danej klasy.
Jego wystąpienie w ciele klasy jest opcjonalne.

Destruktor klasy

Destruktor jest funkcją składową wywoływaną gdy obiekt danej klasy jest likwidowany, a trzeba przy tej okazji wykonać jakieś działania.
Nie wywołuje się destruktora gdy przestaje istnieć referencja lub wskaźnik do obiektu danej klasy.
Jego wystąpienie w ciele klasy jest opcjonalne.

Podobnie jak dla konstruktora nazwa destruktora jest identyczna z nazwą klasy, lecz poprzedzona jest tyldą. Destruktor również nie ma określenia zwracanego typu wartości. Nie posiada argumentów wywołania, więc nie może być przeładowany.

Destruktor klasy

Destruktor jest funkcją składową wywoływaną gdy obiekt danej klasy jest likwidowany, a trzeba przy tej okazji wykonać jakieś działania.
Nie wywołuje się destruktora gdy przestaje istnieć referencja lub wskaźnik do obiektu danej klasy.
Jego wystąpienie w ciele klasy jest opcjonalne.

Podobnie jak dla konstruktora nazwa destruktora jest identyczna z nazwą klasy, lecz poprzedzona jest tyldą. Destruktor również nie ma określenia zwracanego typu wartości. Nie posiada argumentów wywołania, więc nie może być przeładowany.

Destruktor można uruchomić jawnie, lecz w tym przypadku obiekt nie jest likwidowany.
Zapis jawnego wywołania destruktora nie może zaczynać się od tyldy "~".
W ciele funkcji składowej klasy K można go wywołać np. this ->~K();

Nie można pobrać adresu destruktora. Jako funkcja nie może być typu *const* czy *volatile*, lecz może być uruchomiony dla obiektów tego typu. Jeśli klasa nie ma zdefiniowanego destruktora, kompilator wygeneruje pusty destruktor.

Nie można pobrać adresu destruktora. Jako funkcja nie może być typu *const* czy *volatile*, lecz może być uruchomiony dla obiektów tego typu. Jeśli klasa nie ma zdefiniowanego destruktora, kompilator wygeneruje pusty destruktor.

Nie należy rzucać wyjątków z destruktora

Jeśli rzucimy wyjątek w czasie gdy kompilator obsługuje poprzedni wyjątek, program zakończy działanie. Kompilator podczas przenoszenia argumentu pomiędzy instrukcjami *throw* i *catch* wykonuje "odwikłanie stosu", likwidując pewne obiekty na stosie. Jeśli taki obiekt posiada destruktor z instrukcją *throw*, program zostaje zakończony!

Składnik statyczny

Dana statyczna jest tworzona w pamięci jednokrotnie – istnieje jeszcze przed zdefiniowaniem obiektów danej klasy.
Jest ona wspólna dla wszystkich tych obiektów, np. deklaracja w klasie

```
static int skladnik;
```

Składnik statyczny

Dana statyczna jest tworzona w pamięci jednokrotnie – istnieje jeszcze przed zdefiniowaniem obiektów danej klasy.
Jest ona wspólna dla wszystkich tych obiektów, np. deklaracja w klasie

```
static int skladnik;
```

Deklaracja składnika statycznego w ciele klasy nie jest jego definicją. Definicję musimy umieścić podobnie jak definicję zmiennej globalnej, najlepiej na początku pliku z definicjami funkcji składowych. Definicja może zawierać inicjalizację – gdy nie zawiera inicjalizowana jest zerem np.

```
int klasa::skladnik = 5;
```

Taka inicjalizacja możliwa jest także dla składnika prywatnego.

Składnik statyczny może być argumentem domniemanym funkcji składowej.
Klasy definiowane lokalnie nie mogą zawierać składników statycznych.

Składnik statyczny może być argumentem domniemanym funkcji składowej. Klasy definiowane lokalnie nie mogą zawierać składników statycznych.

Inicjalizacja składnika statycznego "w klasie"

- może wystąpić dla typu całkowitego z kwalifikatorem *const*,
- musi wystąpić dla typu *literalnego* z kwalifikatorem *constexpr*.

Definicja składnika statycznego poza klasą nie jest konieczna (chociaż zalecana), gdy jest zainicjalizowany "w klasie" i nie odwołujemy się do jego adresu. Nie można stosować jednocześnie inicjalizacji w klasie i poza klasą.

Funkcja statyczna

pracuje tylko na statycznych składnikach klasy. Jej deklaracja poprzedzona jest słowem *static*. Wywołać ją można poprzedzając nazwą klasy z kwalifikatorem zakresu lub w zwykły sposób. Gdy jest wywołana na rzecz obiektu, interesuje ją tylko klasa, do której on należy. Nie posiada wskaźnika *this*, można więc odwołać się do składnika klasy podając wprost nazwę jakiegoś obiektu.

Funkcja statyczna

pracuje tylko na statycznych składnikach klasy. Jej deklaracja poprzedzona jest słowem *static*. Wywołać ją można poprzedzając nazwą klasy z kwalifikatorem zakresu lub w zwykły sposób. Gdy jest wywołana na rzecz obiektu, interesuje ją tylko klasa, do której on należy. Nie posiada wskaźnika *this*, można więc odwołać się do składnika klasy podając wprost nazwę jakiegoś obiektu.

Funkcje składowe typu *const* oraz *volatile*

zmieniają odpowiednio typ wskaźnika *this*. Funkcja typu *const* zabezpiecza przed zmianą składowe obiektu.

```
int fun(void) const; // int fun(void) const volatile;
```

Statyczne funkcje składowe oraz konstruktory i destruktory nie mogą zawierać kwalifikatora *const*/*volatile*.

Klasa - agregat

grupuje składowe różnych typów oraz dodatkowo:

- wszystkie składowe są publiczne
- nie zawiera konstruktorów zdefiniowanych przez użytkownika
- niestatyczne składniki nie mają inicjalizatorów "w klasie"
- nie ma klas podstawowych i funkcji wirtualnych

Klasa - agregat

grupuje składowe różnych typów oraz dodatkowo:

- wszystkie składowe są publiczne
- nie zawiera konstruktorów zdefiniowanych przez użytkownika
- niestatyczne składniki nie mają inicjalizatorów "w klasie"
- nie ma klas podstawowych i funkcji wirtualnych

Obiekty tej klasy można inicjalizować za pomocą **inicjalizatora klamrowego** (*inicjalzacją agregatową*), np.

```
// według modelu przestrzeni barw RGB
kolory kol1 {"czerwony", 0xff0000};
kolory kol2 {"niebieski", 0xff};
kolory kol3 {"czarny"};
```

Funkcje składowe z kwalifikatorem *constexpr*

mogą występować w wyrażeniach inicjalizujących obiekty *constexpr*.
Funkcja ta musi spełniać warunki:

- nie może być wirtualna
- zwracany typ musi być *literalny* (zdolny do posiadania stałych dosłownych)
- wszystkie jej argumenty muszą być typu *literalnego* (dosłownego)
- ciało jej składa się z przypisku = *delete* lub = *default* lub składa się tylko z instrukcji *return*

Funkcje składowe z kwalifikatorem *constexpr*

mogą występować w wyrażeniach inicjalizujących obiekty *constexpr*.
Funkcja ta musi spełniać warunki:

- nie może być wirtualna
- zwracany typ musi być *literalny* (zdolny do posiadania stałych dosłownych)
- wszystkie jej argumenty muszą być typu *literalnego* (dosłownego)
- ciało jej składa się z przypisku = *delete* lub = *default* lub składa się tylko z instrukcji *return*

Specyfikator **mutable**

pozwala funkcji składowej *const*, dokonać zmian tych składników obiektu *const*, które są poprzedzone specyfikatorem *mutable*.

Deklaracje przyjaźni

Funkcja zaprzyjaźniona z klasą

ma dostęp do wszystkich składników tej klasy, pomimo że nie jest jej składnikiem. Dana funkcja może być zaprzyjaźniona z wieloma klasami. Może to być funkcja globalna czy będąca składnikiem innej klasy.

Deklaracje przyjaźni

Funkcja zaprzyjaźniona z klasą

ma dostęp do wszystkich składników tej klasy, pomimo że nie jest jej składnikiem. Dana funkcja może być zaprzyjaźniona z wieloma klasami. Może to być funkcja globalna czy będąca składnikiem innej klasy.

Zaprzyjaźnienie następuje przez deklarację ze słowem **friend**

```
void Wypisz(punkt p){
   cout << "Wspolrzedne: (" << p.x << ", " << p.y << ")";
}

class punkt{
   double x, y;
   friend void Wypisz(punkt);
}
```

Nazwa klasy przed użyciem musi zostać zadeklarowana. Przed definicją klasy można posłużyć się jej **deklaracją zapowiadającą** (*zwiastującą*). Składa się ona ze słowa *class*, nazwy klasy i średnika, np.

```
class punkt;
```

Nazwa klasy przed użyciem musi zostać zadeklarowana. Przed definicją klasy można posłużyć się jej **deklaracją zapowiadającą** (*zwiastującą*). Składa się ona ze słowa *class*, nazwy klasy i średnika, np.

```
class punkt;
```

Deklaracja przyjaźni deklaruje tylko przyjaźń i nic więcej. Jej nazwa nie staje się nazwą z zakresu klasy. Nie jest składnikiem klasy, więc nie ma wskaźnika *this* do obiektów. Funkcja zaprzyjaźniona jest zwykłą funkcją, której nie obowiązują słowa *private* i *protected* w zaprzyjaźnionych klasach. Deklaracja przyjaźni może być umieszczona w dowolnym miejscu definicji klasy – etykiety dostępu *public*, *protected*, *private* nie mają tu znaczenia. W przypadku funkcji statycznej, trzeba słowo *static* umieścić w deklaracji przyjaźni.

Definicja funkcji zaprzyjaźnionej wewnątrz klasy

Zaprzyjaźniona funkcja zdefiniowana w danej klasie nie jest jej składnikiem. Jest ona typu *inline*. Może skorzystać z obowiązujących wewnątrz definicji klasy instrukcji *using* i *typedef* oraz typów wyliczeniowych *enum*.
Klasa ta nie może być *klasą lokalną*. Funkcja ta nie jest widoczna na zewnątrz klasy, chyba że ma na zewnątrz swoją deklarację.

Definicja funkcji zaprzyjaźnionej wewnątrz klasy

Zaprzyjaźniona funkcja zdefiniowana w danej klasie nie jest jej składnikiem. Jest ona typu *inline*. Może skorzystać z obowiązujących wewnątrz definicji klasy instrukcji *using* i *typedef* oraz typów wyliczeniowych *enum*.
Klasa ta nie może być *klasą lokalną*. Funkcja ta nie jest widoczna na zewnątrz klasy, chyba że ma na zewnątrz swoją deklarację.

Jeśli dana klasa deklaruje przyjaźń z inną, to deklaruje ją ze wszystkimi jej funkcjami. Wewnątrz danej klasy wpisuje się wtedy deklarację przyjaźni z inną klasą, np.

```
friend class punkt;
```

Deklaracja przyjaźni nie może zawierać specyfikatorów, określających sposób umieszczenia w pamięci zaprzyjaźnionej klasy, czyli: *static*, *register*, *extern*, *muteble* i *thread_local*. Dobrym zwyczajem jest umieszczanie deklaracji przyjaźni na samym początku defnicji klasy.

Zaprzyjaźniona klasa może używać składników prywatnych tej drugiej w swoich funkcjach składowych oraz przy inicjalizacji swych składników statycznych (również na zewnątrz). Zaprzyjaźniona klasa może używać przy deklaracji swoich składników, definicji typów *typedef*, *using* czy *enum* z tej drugiej klasy. Zaprzyjaźniona klasa swoją definicję musi mieć na zewnątrz klasy zawierającej z nią deklarację przyjaźni.

Zaprzyjaźniona klasa może używać składników prywatnych tej drugiej w swoich funkcjach składowych oraz przy inicjalizacji swych składników statycznych (również na zewnątrz). Zaprzyjaźniona klasa może używać przy deklaracji swoich składników, definicji typów *typedef*, *using* czy *enum* z tej drugiej klasy. Zaprzyjaźniona klasa swoją definicję musi mieć na zewnątrz klasy zawierającej z nią deklarację przyjaźni.

Deklaracja przyjaźni jest jednostronna.
Lecz dwie klasy mogą też przyjaźnić się z wzajemnością.
Przyjaźń nie jest przechodnia ani dziedziczona. Im mniej klas zaprzyjaźnionych, tym łatwiej panować nad działaniem danej klasy.

Klasa składowa innej klasy

Klasa składowa

zwykle pełni rolę pomocniczą dla klasy, wewnątrz której jest zdefiniowana. Podobnie jak dla innych składników danej klasy, obiekty klasy składowej będą dostępne poza tą daną klasą, gdy definicja klasy składowej znajduje się w części *public* tej danej klasy.

Klasa składowa innej klasy

Klasa składowa

zwykle pełni rolę pomocniczą dla klasy, wewnątrz której jest zdefiniowana. Podobnie jak dla innych składników danej klasy, obiekty klasy składowej będą dostępne poza tą daną klasą, gdy definicja klasy składowej znajduje się w części *public* tej danej klasy.

Dostęp do składników klasy wewnętrznej i zewnętrznej odbywa się tak, jak by klasa zewnętrzna zadeklarowała przyjaźń z klasą wewnętrzną.
Jeśli definicja funkcji klasy składowej nie znajduje się bezpośrednio w tej klasie, to powinna wystąpić na zewnątrz obu klas, opatrzona podwójnym kwalifikatorem zakresu. Klasę składową można podobnie definiować jak funkcję składową, czyli umieścić deklarację w klasie zewnętrznej, a definicję poza tą klasą.

W miejscu pliku poniżej deklaracji klasy a powyżej jej definicji, klasa ta uznawana jest za *typ niekompletny*.
Nie można tu zdefiniować obiektu tej klasy, ale wskaźnik do niego – tak.

W miejscu pliku poniżej deklaracji klasy a powyżej jej definicji, klasa ta uznawana jest za *typ niekompletny*.
Nie można tu zdefiniować obiektu tej klasy, ale wskaźnik do niego – tak.

Lokalna definicja klasy

Gdy zdefiniujemy klasę wewnątrz bloku (np. funkcji), jej zakres ważności bedzię lokalny – wewnątrz tego bloku.
Klasa lokalna nie może posiadać składników statycznych, a jej funkcje składowe muszą być zdefiniowane wewnątrz niej (będą *inline*) – powinny więc być krótkie.

W miejscu pliku poniżej deklaracji klasy a powyżej jej definicji, klasa ta uznawana jest za *typ niekompletny*.
Nie można tu zdefiniować obiektu tej klasy, ale wskaźnik do niego – tak.

Lokalna definicja klasy

Gdy zdefiniujemy klasę wewnątrz bloku (np. funkcji), jej zakres ważności bedzię lokalny – wewnątrz tego bloku.
Klasa lokalna nie może posiadać składników statycznych, a jej funkcje składowe muszą być zdefiniowane wewnątrz niej (będą *inline*) – powinny więc być krótkie.

Klasa lokalna nie może używać zmiennych automatycznych funkcji, w której występuje. Może jednak korzystać ze wszystkich składników zaistniałych już podczas kompilacji i linkowania.

Klasa std::string

Klasa **std::string**

powstała w celu ulepszenia pracy na ciągach znaków. Klasa ta wraz z jej funkcjami weszła w skład biblioteki standardowej.

Klasa std::string

Klasa **std::string**

powstała w celu ulepszenia pracy na ciągach znaków. Klasa ta wraz z jej funkcjami weszła w skład biblioteki standardowej.

Klasa *string* pracuje na zwykłych znakach typu *char*.
Biblioteka standardowa zawiera również bliźniacze wersje tej klasy dla znaków o większej szerokości:

- klasa **wstring** – pracuje na znakach *wchar_t*
- klasa **u16string** – pracuje na znakach *char16_t*
- klasa **u32string** – pracuje na znakach *char32_t*

Klasy te mają ten sam zestaw funkcji składowych.

Klasa std::string

Klasa **std::string**

powstała w celu ulepszenia pracy na ciągach znaków. Klasa ta wraz z jej funkcjami weszła w skład biblioteki standardowej.

Klasa *string* pracuje na zwykłych znakach typu *char*.
Biblioteka standardowa zawiera również bliźniacze wersje tej klasy dla znaków o większej szerokości:

- klasa **wstring** – pracuje na znakach *wchar_t*
- klasa **u16string** – pracuje na znakach *char16_t*
- klasa **u32string** – pracuje na znakach *char32_t*

Klasy te mają ten sam zestaw funkcji składowych.

Aby korzystać z bibliotecznej klasy *string* musimy dołączyć plik nagłówkowy <string>, zawierający deklarację tej klasy.
Nazwa klasy *string* wchodzi do przestrzeni nazw *std*.

Operatory „=”, „+”, „+=”

Operator „=” kopiuje zawartość obiektu klasy *string* po jego prawej stronie, do obiektu *string* po lewej.
Operator „+” łączy ze sobą dwa obiekty klasy *string*.
Operator „+=” dopisuje na końcu obiektu klasy *string* zawartość innego obiektu tej klasy, *C-string*, stałą lub zmienną typu *char* czy też znaków zgromadzonych na *liście inicjalizatorów* np. { 'a', 'b', 'c'}.

Konstruktory klasy string

Konstruktory klasy string

- *string();* – powstaje pusty obiekt klasy *string*. Jego zawartość odpowiada *C-stringowi* "",

Konstruktory klasy string

- *string();* – powstaje pusty obiekt klasy *string*. Jego zawartość odpowiada *C-stringowi* "",
- *string(const char *cstr);* – obiekt inicjalizowany jest podanym *C-stringiem* (wskaźnikiem do *C-stringu*),

Konstruktory klasy string

- *string();* – powstaje pusty obiekt klasy *string*. Jego zawartość odpowiada *C-stringowi* "",
- *string(const char *cstr);* – obiekt inicjalizowany jest podanym *C-stringiem* (wskaźnikiem do *C-stringu*),
- *string(const char *cstr, size_t n);* – obiekt inicjalizowany pierwszymi *n* znakami podanego *C-stringu*. Typ *size_t* określa nieujemną liczbę całkowitą,

Konstruktory klasy string

- *string();* – powstaje pusty obiekt klasy *string*. Jego zawartość odpowiada *C-stringowi* "",
- *string(const char *cstr);* – obiekt inicjalizowany jest podanym *C-stringiem* (wskaźnikiem do *C-stringu*),
- *string(const char *cstr, size_t n);* – obiekt inicjalizowany pierwszymi *n* znakami podanego *C-stringu*. Typ *size_t* określa nieujemną liczbę całkowitą,
- *string(size_t n, char znak);* – obiekt inicjalizowany *n* jednakowymi znakami *znak*,

Konstruktory klasy string

- *string();* – powstaje pusty obiekt klasy *string*. Jego zawartość odpowiada *C-stringowi* "",
- *string(const char *cstr);* – obiekt inicjalizowany jest podanym *C-stringiem* (wskaźnikiem do *C-stringu*),
- *string(const char *cstr, size_t n);* – obiekt inicjalizowany pierwszymi *n* znakami podanego *C-stringu*. Typ *size_t* określa nieujemną liczbę całkowitą,
- *string(size_t n, char znak);* – obiekt inicjalizowany *n* jednakowymi znakami *znak*,
- *string(lista inicjalizatorów);* – w skład *listy inicjalizatorów* wchodzą pojedyncze znaki, np. {'p', 'q', 'r'} – powstanie *string* "pqr",

Konstruktory klasy string

- *string();* – powstaje pusty obiekt klasy *string*. Jego zawartość odpowiada *C-stringowi* "",
- *string(const char *cstr);* – obiekt inicjalizowany jest podanym *C-stringiem* (wskaźnikiem do *C-stringu*),
- *string(const char *cstr, size_t n);* – obiekt inicjalizowany pierwszymi *n* znakami podanego *C-stringu*. Typ *size_t* określa nieujemną liczbę całkowitą,
- *string(size_t n, char znak);* – obiekt inicjalizowany *n* jednakowymi znakami *znak*,
- *string(lista inicjalizatorów);* – w skład *listy inicjalizatorów* wchodzą pojedyncze znaki, np. {'p', 'q', 'r'} – powstanie *string* "pqr",
- *string(const string &napis, size_t pozycja, size_t n);* – obiekt inicjalizowany jest *n* znakami stringu *napis*, począwszy od znaku na miejscu *pozycja* (licząc od 0).

Funkcje składowe klasy string

Funkcje składowe klasy string

- *size_t size();* – zwraca ilość znaków w obiekcie klasy *string*.

Funkcje składowe klasy string

- *size_t size();* – zwraca ilość znaków w obiekcie klasy *string*.
- *size_t length();* – również zwraca ilość znaków w obiekcie klasy *string*.

Funkcje składowe klasy string

- *size_t size();* – zwraca ilość znaków w obiekcie klasy *string*.
- *size_t length();* – również zwraca ilość znaków w obiekcie klasy *string*.
- *bool empty();* – zwraca *true*, gdy obiekt nie zawiera żadnego znaku.

Funkcje składowe klasy string

- *size_t size();* – zwraca ilość znaków w obiekcie klasy *string*.
- *size_t length();* – również zwraca ilość znaków w obiekcie klasy *string*.
- *bool empty();* – zwraca *true*, gdy obiekt nie zawiera żadnego znaku.
- *size_t max_size();* – podaje maksymalną ilość znaków, które mógłby pomieścić obiekt. Pośrednio podaje w ten sposób informację o ilości dostępnej pamięci komputera,

Funkcje składowe klasy string

- *size_t size();* – zwraca ilość znaków w obiekcie klasy *string*.
- *size_t length();* – również zwraca ilość znaków w obiekcie klasy *string*.
- *bool empty();* – zwraca *true*, gdy obiekt nie zawiera żadnego znaku.
- *size_t max_size();* – podaje maksymalną ilość znaków, które mógłby pomieścić obiekt. Pośrednio podaje w ten sposób informację o ilości dostępnej pamięci komputera,
- *size_t capacity();* – podaje rozmiar pamięci zarezerwowanej dla obiektu. Pamięć rezerwowana jest często z pewnym zapasem.

Funkcje składowe klasy string

- *size_t size();* – zwraca ilość znaków w obiekcie klasy *string*.
- *size_t length();* – również zwraca ilość znaków w obiekcie klasy *string*.
- *bool empty();* – zwraca *true*, gdy obiekt nie zawiera żadnego znaku.
- *size_t max_size();* – podaje maksymalną ilość znaków, które mógłby pomieścić obiekt. Pośrednio podaje w ten sposób informację o ilości dostępnej pamięci komputera,
- *size_t capacity();* – podaje rozmiar pamięci zarezerwowanej dla obiektu. Pamięć rezerwowana jest często z pewnym zapasem.
- *void reserve(size_t n=0);* – rezerwuje pamięć dla co najmniej *n* znaków. Gdy podamy mniejszą wartość, niż bieżący rozmiar pamięci obiektu, rozmiar ten może się zmniejszyć maksymalnie do podanej wartości, nie mniej jednak niż do ilości znaków zawartych w obiekcie. Każda zmiana ilości pamięci zarezerwowanej dla obiektu, może zmienić ustawienia wskaźników pokazujących na miejsca w stringu, leżącym w tym obiekcie. Odpowiednio duża zapobiega tym zmianom.

Funkcje składowe klasy string cd..

Funkcje składowe klasy string cd..

- *void shrink_to_fit();* – odpowiada wywołaniu funkcji *reserve();*

Funkcje składowe klasy string cd..

- *void shrink_to_fit();* – odpowiada wywołaniu funkcji *reserve();*
- *void resize(size_t n, char znak='\0');* – *n* to nowa ilość znaków w obiekcie. W przypadku zwiększenia rozmiaru pozostałe miejsca wypełniane są argumentem *znak*. W przypadku podania liczby mniejszej niż ilość znaków w obiekcie, pozostałe znaki zostaną skasowane. Chociaż domyślny znak (bajt zerowy) nie oznacza końca stringu, znak ten stawiany jest na jego końcu.

Funkcje składowe klasy string cd..

- *void shrink_to_fit();* – odpowiada wywołaniu funkcji *reserve();*
- *void resize(size_t n, char znak='\0');* – *n* to nowa ilość znaków w obiekcie. W przypadku zwiększenia rozmiaru pozostałe miejsca wypełniane są argumentem *znak*. W przypadku podania liczby mniejszej niż ilość znaków w obiekcie, pozostałe znaki zostaną skasowane. Chociaż domyślny znak (bajt zerowy) nie oznacza końca stringu, znak ten stawiany jest na jego końcu.
- *void clear();* – kasuje zawartość obiektu string.

Operator []

Operator ten działa podobnie jak dla tablic znakowych, włącznie z niebezpieczeństwem wyjścia indeksu poza zakres obiektu np.

```
string napis("Tresc napisu");
cout << napis[3] << '\n';
```

Operator []

Operator ten działa podobnie jak dla tablic znakowych, włącznie z niebezpieczeństwem wyjścia indeksu poza zakres obiektu np.

```
string napis("Tresc napisu");
cout << napis[3] << '\n';
```

Funkcja składowa **at**

Funkcja *at* różni sie od operatora [] tym, że gdy odnosimy się do nieistniejącego elementu stringu, rzuca wyjątek klasy *out_of_range*. Deklaracja nazwy tego wyjątku znajduje się w pliku *stdexcept*.

```
#include <stdexcept>
...
try {
   napis.at(4) = 'z'; // zwraca referencję do znaku
} catch(out_of_range){
   cout << "Proba przypisania poza stringiem.";
}
```

Przebieganie po wszystkich znakach stringu zakresowym for

Kolejny znak stringu reprezentowany jest przez zmienną za słowem *auto* np.

```
for(auto &znak : napis)
    cout << znak;
```

Przebieganie po wszystkich znakach stringu zakresowym for

Kolejny znak stringu reprezentowany jest przez zmienną za słowem *auto* np.

```
for(auto &znak : napis)
    cout << znak;
```

Funkcje składowe front i back

Funkcji tych używamy pod warunkiem, że string nie jest pusty.
char &front(); – udostępnia referencję do pierwszego znaku stringu,
char &back(); – udostępnia referencję do ostatniego znaku stringu.

```
cout << napis.front(); // odpowiada: napis[0];
napis.back() = 'z';
```

Operacje na liczbach

Wczytanie liczby ze stringu

Funkcje zwracające wartości różnych typów:

```
int stoi(const string &str, size_t *idx = 0, int base = 10);
long stol( ..., ..., ...);
unsigned long stoul( ..., ..., ...);
long long stoll( ..., ..., ...);
unsigned long long stoull( ..., ..., ...);
float stof(const string &str, size_t *idx = 0);
double stod( ..., ...);
long double stold( ..., ...);
```

Argumenty funkcji

Argumenty funkcji

- *str* – referencja do stringu z tekstowym zapisem liczby.

Argumenty funkcji

- *str* – referencja do stringu z tekstowym zapisem liczby.
- *idx* – podaje indeks pierwszego nieodczytanego znaku stringu (nie będącego częścią liczby), czyli ilość przeczytanych znaków. Domyślna wartość *nullptr* oznacza, że nie potrzebujemy tej informacji.

Argumenty funkcji

- *str* – referencja do stringu z tekstowym zapisem liczby.
- *idx* – podaje indeks pierwszego nieodczytanego znaku stringu (nie będącego częścią liczby), czyli ilość przeczytanych znaków. Domyślna wartość *nullptr* oznacza, że nie potrzebujemy tej informacji.
- *base* – określa w jakim systemie liczbowym należy odczytać liczbę.

Argumenty funkcji

- *str* – referencja do stringu z tekstowym zapisem liczby.
- *idx* – podaje indeks pierwszego nieodczytanego znaku stringu (nie będącego częścią liczby), czyli ilość przeczytanych znaków. Domyślna wartość *nullptr* oznacza, że nie potrzebujemy tej informacji.
- *base* – określa w jakim systemie liczbowym należy odczytać liczbę.

Rzucanie wyjątków

Funkcje te ignorują początkowe białe znaki. Jeśli na samym początku stringu funkcja nie znajdzie zapisu liczby, rzucany jest wyjątek typu *invalid_argument*. Gdy odczytywana liczba wychodzi poza zakres zadanego typu, rzucany jest wyjątek typu *out_of_range*.

Zamiana liczby na string: funkcja *string to_string(int);*

Argument funkcji zamieniany jest na tekst.
Istnieją przeładowania tej funkcji dla argumentów o typach całkowitych: *int*, *long*, *long long* oraz w połączeniu z kwalifikatorem *unsigned*.
Dla typów zmiennoprzecinkowych: *float*, *double* oraz *long double*.

Funkcja pobierająca fragment stringu **substr**

```
string substr(size_t pozycja, size_t n = string::npos);
```

Funkcja zwraca fragment stringu dla poniższych argumentów:

Funkcja pobierająca fragment stringu **substr**

```
string substr(size_t pozycja, size_t n = string::npos);
```

Funkcja zwraca fragment stringu dla poniższych argumentów:

- *pozycja* – indeks pierwszego znaku wynikowego fragmentu stringu. Gdy przekroczy on długość stringu, funkcja ta rzuci wyjątek typu *out_of_range*.

Funkcja pobierająca fragment stringu **substr**

```
string substr(size_t pozycja, size_t n = string::npos);
```

Funkcja zwraca fragment stringu dla poniższych argumentów:

- *pozycja* – indeks pierwszego znaku wynikowego fragmentu stringu. Gdy przekroczy on długość stringu, funkcja ta rzuci wyjątek typu *out_of_range*.
- *n* – ilość znaków wynikowego fragmentu. Wartość domniemana oznacza pobranie wszystkich znaków do końca stringu. Wartość *npos* oznacza największą możliwą liczbę całkowitą w danym komputerze.

Funkcja **find** szukająca ciągu znaków

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję pierwszego wystąpienia tego ciągu znaków (nie wcześniejszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

Funkcja **find** szukająca ciągu znaków

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję pierwszego wystąpienia tego ciągu znaków (nie wcześniejszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0) const noexcept;*

Funkcja **find** szukająca ciągu znaków

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję pierwszego wystąpienia tego ciągu znaków (nie wcześniejszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0) const noexcept;*
- *(const string &ciag, size_t start = 0) const noexcept;*

Funkcja **find** szukająca ciągu znaków

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję pierwszego wystąpienia tego ciągu znaków (nie wcześniejszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0) const noexcept;*
- *(const string &ciag, size_t start = 0) const noexcept;*
- *(const char *ciag, size_t start = 0) const;*

Funkcja **find** szukająca ciągu znaków

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję pierwszego wystąpienia tego ciągu znaków (nie wcześniejszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0) const noexcept;*
- *(const string &ciag, size_t start = 0) const noexcept;*
- *(const char *ciag, size_t start = 0) const;*
- *(const char *ciag, size_t start, size_t n) const;*

Funkcja **find** szukająca ciągu znaków

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję pierwszego wystąpienia tego ciągu znaków (nie wcześniejszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0) const noexcept;*
- *(const string &ciag, size_t start = 0) const noexcept;*
- *(const char *ciag, size_t start = 0) const;*
- *(const char *ciag, size_t start, size_t n) const;*

Aby funkcje *find* mogły pracować na stringach stałych, na końcu ich deklaracji znajduje się *const*.
noexcept zabezpiecza przed rzucaniem wyjątków.

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.
- *n* – ilość wyszukiwanych, początkowych znaków ciągu.

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.
- *n* – ilość wyszukiwanych, początkowych znaków ciągu.

Funkcja **rfind** szukająca ciągu znaków od końca stringu

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję ostatniego wystąpienia tego ciągu znaków (nie dalszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.
- *n* – ilość wyszukiwanych, początkowych znaków ciągu.

Funkcja **rfind** szukająca ciągu znaków od końca stringu

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję ostatniego wystąpienia tego ciągu znaków (nie dalszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = npos) const noexcept;*

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.
- *n* – ilość wyszukiwanych, początkowych znaków ciągu.

Funkcja **rfind** szukająca ciągu znaków od końca stringu

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję ostatniego wystąpienia tego ciągu znaków (nie dalszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = npos) const noexcept;*
- *(const string &ciag, size_t start = npos) const noexcept;*

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.
- *n* – ilość wyszukiwanych, początkowych znaków ciągu.

Funkcja **rfind** szukająca ciągu znaków od końca stringu

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję ostatniego wystąpienia tego ciągu znaków (nie dalszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = npos) const noexcept;*
- *(const string &ciag, size_t start = npos) const noexcept;*
- *(const char *ciag, size_t start = npos) const;*

Znaczenie parametrów

- *start* – pozycja znaku, od którego rozpoczyna szukanie.
- *n* – ilość wyszukiwanych, początkowych znaków ciągu.

Funkcja **rfind** szukająca ciągu znaków od końca stringu

Funkcja zwraca pozycję znaku typu *size_t*, będącego początkiem szukanego ciągu. Zwraca pozycję ostatniego wystąpienia tego ciągu znaków (nie dalszą niż wartość drugiego argumentu). W przypadku braku tego ciągu znaków, funkcja zwraca wartość *string::npos*.
Funkcja posiada 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = npos) const noexcept;*
- *(const string &ciag, size_t start = npos) const noexcept;*
- *(const char *ciag, size_t start = npos) const;*
- *(const char *ciag, size_t start, size_t n) const;*

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu
- *find_last_not_of* – ostatnie wystąpienie znaku spoza zestawu

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu
- *find_last_not_of* – ostatnie wystąpienie znaku spoza zestawu

Funkcje te posiadają 4 przeładowania (zestawy parametrów):

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu
- *find_last_not_of* – ostatnie wystąpienie znaku spoza zestawu

Funkcje te posiadają 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0 (npos)) const noexcept;*

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu
- *find_last_not_of* – ostatnie wystąpienie znaku spoza zestawu

Funkcje te posiadają 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0 (npos)) const noexcept;*
- *(const string &ciag, size_t start = 0 (npos)) const noexcept;*

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu
- *find_last_not_of* – ostatnie wystąpienie znaku spoza zestawu

Funkcje te posiadają 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0 (npos)) const noexcept;*
- *(const string &ciag, size_t start = 0 (npos)) const noexcept;*
- *(const char *ciag, size_t start = 0 (npos)) const;*

Funkcje szukające jednego ze znaków podanych poprzez zestaw

Mamy 4 rodzaje fukcji typu *size_t*, zwracających pozycję znaku:

- *find_first_of* – pierwsze wystąpienie znaku z zestawu
- *find_last_of* – otatnie wystąpienie znaku z zestawu
- *find_first_not_of* – pierwsze wystąpienie znaku spoza zestawu
- *find_last_not_of* – ostatnie wystąpienie znaku spoza zestawu

Funkcje te posiadają 4 przeładowania (zestawy parametrów):

- *(char znak, size_t start = 0 (npos)) const noexcept;*
- *(const string &ciag, size_t start = 0 (npos)) const noexcept;*
- *(const char *ciag, size_t start = 0 (npos)) const;*
- *(const char *ciag, size_t start, size_t n) const;*

Funkcja **erase** usuwa znaki ze stringu

w ilości *n*, począwszy od znaku na pozycji *start*.

```
string &erase(size_t start = 0, size_t n = npos);
```

Funkcja **erase** usuwa znaki ze stringu

w ilości *n*, począwszy od znaku na pozycji *start*.

```
string &erase(size_t start = 0, size_t n = npos);
```

- funkcja zwraca referencję do obiektu z którym pracowała,

Funkcja **erase** usuwa znaki ze stringu

w ilości *n*, począwszy od znaku na pozycji *start*.

```
string &erase(size_t start = 0, size_t n = npos);
```

- funkcja zwraca referencję do obiektu z którym pracowała,
- jeśli pierwszy argument wskazuje pozycję poza stringiem, funkcja rzuci wyjątek *out_of_range*,

Funkcja **erase** usuwa znaki ze stringu

w ilości *n*, począwszy od znaku na pozycji *start*.

```
string &erase(size_t start = 0, size_t n = npos);
```

- funkcja zwraca referencję do obiektu z którym pracowała,
- jeśli pierwszy argument wskazuje pozycję poza stringiem, funkcja rzuci wyjątek *out_of_range*,
- jeśli do końca stringu jest mniej niż *n* znaków funkcja skasuje znaki tylko do końca stringu.

Funkcja **erase** usuwa znaki ze stringu

w ilości *n*, począwszy od znaku na pozycji *start*.

```
string &erase(size_t start = 0, size_t n = npos);
```

- funkcja zwraca referencję do obiektu z którym pracowała,
- jeśli pierwszy argument wskazuje pozycję poza stringiem, funkcja rzuci wyjątek *out_of_range*,
- jeśli do końca stringu jest mniej niż *n* znaków funkcja skasuje znaki tylko do końca stringu.

Funkcja **pop_back** usuwa ostatni znak stringu

```
void pop_back();
```

Funkcje **insert** wstawiające znaki do istniejącego *stringu*

zwracają referencje do obiektu z którym zakończyły pracę.
Znaki są wstawiane w miejscu określonym przez pierwszy argument: *i1*.
Obiekt jest rozszerzany od tego miejsca, aby pomieścić nowe znaki.

Funkcje **insert** wstawiające znaki do istniejącego *stringu*

zwracają referencje do obiektu z którym zakończyły pracę.
Znaki są wstawiane w miejscu określonym przez pierwszy argument: *i1*.
Obiekt jest rozszerzany od tego miejsca, aby pomieścić nowe znaki.

_string &insert(size_t i1, const string &tekst)_

treść *tekst* podana jako drugi argument, wstawiana jest do stringu.

Funkcje **insert** wstawiające znaki do istniejącego *stringu*

zwracają referencje do obiektu z którym zakończyły pracę.
Znaki są wstawiane w miejscu określonym przez pierwszy argument: *i1*.
Obiekt jest rozszerzany od tego miejsca, aby pomieścić nowe znaki.

_string &insert(size_t i1, const string &tekst)_

treść *tekst* podana jako drugi argument, wstawiana jest do stringu.

_string &insert(size_t i1, const string &tekst, size_t i2, size_t n)_

treść *tekst* od miejsca *i2* w ilości *n* znaków, wstawiana jest do stringu.

*string &insert(size_t i1, const char *tekst)*

treść *tekst* podana jako drugi argument, wstawiana jest do stringu.

*string &insert(size_t i1, const char *tekst)*

treść *tekst* podana jako drugi argument, wstawiana jest do stringu.

*string &insert(size_t i1, const char *tekst, size_t n)*

treść *tekst* w ilości *n* znaków, wstawiana jest do stringu.

*string &insert(size_t i1, const char *tekst)*

treść *tekst* podana jako drugi argument, wstawiana jest do stringu.

*string &insert(size_t i1, const char *tekst, size_t n)*

treść *tekst* w ilości *n* znaków, wstawiana jest do stringu.

string &insert(size_t i1, size_t n, const char znak)

znaki *znak* w ilości *n*, wstawiane są do stringu.

Przeładowania **insert**

```
string cyfry("012345");
string wLitery("ABCDEF");
cyfry.insert( 4, wLitery); // "0123ABCDEF45"
cyfry.insert( 4, wLitery, 1, 3); // "0123BCD45"

const char *mLitery = "abcdef";
cyfry.insert( 4, mLitery); // "0123abcdef45"
cyfry.replace( 4, mLitery+1, 3); // "0123bcd45"

cyfry.replace( 4, 3, '*'); // "0123***45"
```

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku
- *n1* – całkowita ilość kolejnych znaków do zastąpienia

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku
- *n1* – całkowita ilość kolejnych znaków do zastąpienia

_string &replace(size_t i1, size_t n1, const string &tekst)_

wyznaczony fragment zastępowany jest trzecim argumentem: *tekst*.

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku
- *n1* – całkowita ilość kolejnych znaków do zastąpienia

_string &replace(size_t i1, size_t n1, const string &tekst)_

wyznaczony fragment zastępowany jest trzecim argumentem: *tekst*.

_...(size_t i1, size_t n1, const string &tekst, size_t i2, size_t n2)_

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku
- *n1* – całkowita ilość kolejnych znaków do zastąpienia

_string &replace(size_t i1, size_t n1, const string &tekst)_

wyznaczony fragment zastępowany jest trzecim argumentem: *tekst*.

_...(size_t i1, size_t n1, const string &tekst, size_t i2, size_t n2)_

- *tekst* – nowa, zastępująca treść

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku
- *n1* – całkowita ilość kolejnych znaków do zastąpienia

string &replace(size_t i1, size_t n1, const string &tekst)

wyznaczony fragment zastępowany jest trzecim argumentem: *tekst*.

...(size_t i1, size_t n1, const string &tekst, size_t i2, size_t n2)

- *tekst* – nowa, zastępująca treść
- *i2* – indeks pierwszego zastępującego znaku obiektu *tekst*

Funkcje **repalce** zamieniające część znaków na inne

zwracają referencje do obiektu z którym zakończyły pracę. Pierwsze 2 argumenty typu *size_t* wyznaczają zakres zastępowanych znaków:

- *i1* – indeks pierwszego zastępowanego znaku
- *n1* – całkowita ilość kolejnych znaków do zastąpienia

_string &replace(size_t i1, size_t n1, const string &tekst)_

wyznaczony fragment zastępowany jest trzecim argumentem: *tekst*.

_...(size_t i1, size_t n1, const string &tekst, size_t i2, size_t n2)_

- *tekst* – nowa, zastępująca treść
- *i2* – indeks pierwszego zastępującego znaku obiektu *tekst*
- *n2* – całkowita ilość kolejnych znaków zastępujących

*string &replace(size_t i1, size_t n1, const char *tekst)*

wyznaczony fragment zastępowany jest C-stringiem: *tekst*.

*string &replace(size_t i1, size_t n1, const char *tekst)*

wyznaczony fragment zastępowany jest C-stringiem: *tekst*.

*string &replace(size_t i1, size_t n1, const char *tekst, size_t n2)*

*string &replace(size_t i1, size_t n1, const char *tekst)*

wyznaczony fragment zastępowany jest C-stringiem: *tekst*.

*string &replace(size_t i1, size_t n1, const char *tekst, size_t n2)*

- *tekst* – zastępująca treść (początkowy znak tego *C-stringu* można ustalić dodając do tego argumentu liczbę, stanowiącą indeks tego znaku)

*string &replace(size_t i1, size_t n1, const char *tekst)*

wyznaczony fragment zastępowany jest C-stringiem: *tekst*.

*string &replace(size_t i1, size_t n1, const char *tekst, size_t n2)*

- *tekst* – zastępująca treść (początkowy znak tego *C-stringu* można ustalić dodając do tego argumentu liczbę, stanowiącą indeks tego znaku)
- *n2* – całkowita ilość kolejnych znaków zastępujących

*string &replace(size_t i1, size_t n1, const char *tekst)*

wyznaczony fragment zastępowany jest C-stringiem: *tekst*.

*string &replace(size_t i1, size_t n1, const char *tekst, size_t n2)*

- *tekst* – zastępująca treść (początkowy znak tego *C-stringu* można ustalić dodając do tego argumentu liczbę, stanowiącą indeks tego znaku)
- *n2* – całkowita ilość kolejnych znaków zastępujących

string &replace(size_t i1, size_t n1, size_t n, const char znak)

znaki *znak* w ilości *n*, zastępują wyznaczony fragment.

Przeładowania replace

```
string cyfry("0123456789");
string wLitery("ABCDEF");
cyfry.replace( 4, 2, wLitery); // "0123ABCDEF6789"
cyfry.replace( 4, 2, wLitery, 1, 3); // "0123BCD6789"

const char *mLitery = "abcdef";
cyfry.replace( 4, 2, mLitery); // "0123abcdef6789"
cyfry.replace( 4, 2, mLitery+1, 3); // "0123bcd6789"

cyfry.replace( 4, 2, 3, '*'); // "0123***6789"
```

Funkcje zwracające adres obiektu klasy *string*

Funkcja *const char *data() const noexcept;*

podaje bieżący adres tablicy przechowującej znaki obiektu *string*.
Dozwolone jest tylko czytanie tej tablicy.
Od wersji standardu *C++11* gwarantowane jest zakończenie ciągu znaków znakiem *null*, aby tworzył poprawny *C-string*.

Funkcje zwracające adres obiektu klasy *string*

Funkcja *const char *data() const noexcept;*

podaje bieżący adres tablicy przechowującej znaki obiektu *string*.
Dozwolone jest tylko czytanie tej tablicy.
Od wersji standardu *C++11* gwarantowane jest zakończenie ciągu znaków znakiem *null*, aby tworzył poprawny *C-string*.

Funkcja *const char *c_str() const noexcept;*

podaje bieżący adres tablicy przechowującej znaki obiektu *string* w postaci stałego *C-stringu*.
Dozwolone jest tylko czytanie tej tablicy. Zakończona jest znakiem *null*.

Porównywanie *stringów*

Wszystkie wielkie litery są w kolejności alfabetycznej przed wszystkimi małymi literami.
W celu porównania ciągów liter niezależnie od ich wielkości, można porównywać ich kopie składające się z odpowiednich małych liter.

Porównywanie *stringów*

Wszystkie wielkie litery są w kolejności alfabetycznej przed wszystkimi małymi literami.
W celu porównania ciągów liter niezależnie od ich wielkości, można porównywać ich kopie składające się z odpowiednich małych liter.

Funkcja **compare**

zwraca wartość typu *int*:

Porównywanie *stringów*

Wszystkie wielkie litery są w kolejności alfabetycznej przed wszystkimi małymi literami.
W celu porównania ciągów liter niezależnie od ich wielkości, można porównywać ich kopie składające się z odpowiednich małych liter.

Funkcja **compare**

zwraca wartość typu *int*:

- **ujemną**, gdy porównywany *string* w kolejności alfabetycznej znajduje się przed *stringiem* podanym jako argument

Porównywanie *stringów*

Wszystkie wielkie litery są w kolejności alfabetycznej przed wszystkimi małymi literami.
W celu porównania ciągów liter niezależnie od ich wielkości, można porównywać ich kopie składające się z odpowiednich małych liter.

Funkcja **compare**

zwraca wartość typu *int*:

- **ujemną**, gdy porównywany *string* w kolejności alfabetycznej znajduje się przed *stringiem* podanym jako argument
- **zero**, gdy porównywane *stringi* są identyczne

Porównywanie *stringów*

Wszystkie wielkie litery są w kolejności alfabetycznej przed wszystkimi małymi literami.
W celu porównania ciągów liter niezależnie od ich wielkości, można porównywać ich kopie składające się z odpowiednich małych liter.

Funkcja **compare**

zwraca wartość typu *int*:

- **ujemną**, gdy porównywany *string* w kolejności alfabetycznej znajduje się przed *stringiem* podanym jako argument
- **zero**, gdy porównywane *stringi* są identyczne
- **dodatnią**, gdy porównywany *string* w kolejności alfabetycznej znajduje się po *stringu* podanym jako argument

Porównywanie *stringów*

Wszystkie wielkie litery są w kolejności alfabetycznej przed wszystkimi małymi literami.
W celu porównania ciągów liter niezależnie od ich wielkości, można porównywać ich kopie składające się z odpowiednich małych liter.

Funkcja **compare**

zwraca wartość typu *int*:

- **ujemną**, gdy porównywany *string* w kolejności alfabetycznej znajduje się przed *stringiem* podanym jako argument
- **zero**, gdy porównywane *stringi* są identyczne
- **dodatnią**, gdy porównywany *string* w kolejności alfabetycznej znajduje się po *stringu* podanym jako argument

int compare(const string &str2) const noexcept;

porównuje dany *string* z podanym jako argument.

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

(size_t i1, size_t n1, const string &str2, size_t i2, size_t n2) const;

porównuje fragment danego *stringu* z fragmentem innego podanego jako argument:

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

(size_t i1, size_t n1, const string &str2, size_t i2, size_t n2) const;

porównuje fragment danego *stringu* z fragmentem innego podanego jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

(size_t i1, size_t n1, const string &str2, size_t i2, size_t n2) const;

porównuje fragment danego *stringu* z fragmentem innego podanego jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

(size_t i1, size_t n1, const string &str2, size_t i2, size_t n2) const;

porównuje fragment danego *stringu* z fragmentem innego podanego jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu
- *str2*, inny porównywany *string*

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

(size_t i1, size_t n1, const string &str2, size_t i2, size_t n2) const;

porównuje fragment danego *stringu* z fragmentem innego podanego jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu
- *str2*, inny porównywany *string*
- *i2*, indeks pierwszego znaku fragmentu drugiego porównywanego *stringu*

int compare(size_t i1, size_t n1, const string &str2) const;

porównuje fragment danego *stringu* z podanym *str2* jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu (gdy jest większa niż ilość znaków do końca stringu, porównywana jest cała reszta do końca *stringu* – może więc być *string::npos* podany jako argument)

(size_t i1, size_t n1, const string &str2, size_t i2, size_t n2) const;

porównuje fragment danego *stringu* z fragmentem innego podanego jako argument:

- *i1*, indeks pierwszego znaku fragmentu porównywanego *stringu*
- *n1*, ilość kolejnych znaków tego fragmentu
- *str2*, inny porównywany *string*
- *i2*, indeks pierwszego znaku fragmentu drugiego porównywanego *stringu*
- *n2*, ilość kolejnych znaków tego fragmentu

*int compare(const char *tekst) const;*

obiekt *string* porównujemy z *C-stringiem*.

*int compare(const char *tekst) const;*

obiekt *string* porównujemy z *C-stringiem*.

*... (size_t i1, size_t n1, const char *tekst, size_t n2) const;*

porównuje fragment danego *stringu* z fragmentem *C-stringu* np.

```
string poeta("A. Mickiewicz");
char *malarz = "Jan Matejko";
if(poeta.compare(3, string::npos, malarz+4, 300) < 0)
   cout << poeta << " alfabetycznie poprzedza " << malarz;
```

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

- *s1* == *s2*, równość stringów *s1* i *s2*

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

- *s1 == s2*, równość stringów *s1* i *s2*
- *s1 != s2*, czy *s1* i *s2* są różne

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

- *s1 == s2*, równość stringów *s1* i *s2*
- *s1 != s2*, czy *s1* i *s2* są różne
- *s1 < s2*, czy w porządku alfabetycznym *s1* poprzedza *s2*

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

- *s1* == *s2*, równość stringów *s1* i *s2*
- *s1* != *s2*, czy *s1* i *s2* są różne
- *s1* < *s2*, czy w porządku alfabetycznym *s1* poprzedza *s2*
- *s1* > *s2*, czy w porządku alfabetycznym *s2* poprzedza *s1*

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

- *s1 == s2*, równość stringów *s1* i *s2*
- *s1 != s2*, czy *s1* i *s2* są różne
- *s1 < s2*, czy w porządku alfabetycznym *s1* poprzedza *s2*
- *s1 > s2*, czy w porządku alfabetycznym *s2* poprzedza *s1*
- *s1 <= s2*, czy w porządku alfabetycznym *s1* poprzedza *s2* lub są one równe

Przeładowanie operatorów „==”, „!=”, „<”, „>”, „<=”, „>=”.

Przynajmniej jeden z porównywanych obiektów musi być obiektem klasy *string*. Nie można przy pomocy tych operatorów porównywać dwóch *C-stringów*. Rezultatem takiego porównania jest wartość typu *bool*. Podobnie jak dla funkcji *compare* w porządku alfabetycznym wszystkie wielkie litery poprzedzają wszystkie małe litery.

Sprawdzenia porządku alfabetycznego stringów *s1* i *s2*:

- *s1 == s2*, równość stringów *s1* i *s2*
- *s1 != s2*, czy *s1* i *s2* są różne
- *s1 < s2*, czy w porządku alfabetycznym *s1* poprzedza *s2*
- *s1 > s2*, czy w porządku alfabetycznym *s2* poprzedza *s1*
- *s1 <= s2*, czy w porządku alfabetycznym *s1* poprzedza *s2* lub są one równe
- *s1 >= s2*, czy w porządku alfabetycznym *s2* poprzedza *s1* lub są one równe

Zamiana liter stringu na małe. Funkacja *std::tolower*

zamienia jedną literę na odpowiednią małą. Deklaracja funkcji *tolower* dostępna jest w pliku nagłówkowym *cctype*. Funkcja ta przetwarza angielskie litery i nie potrafi zamieniać polskich liter.

Zamiana liter stringu na małe. Funkacja *std::tolower*

zamienia jedną literę na odpowiednią małą. Deklaracja funkcji *tolower* dostępna jest w pliku nagłówkowym *cctype*. Funkcja ta przetwarza angielskie litery i nie potrafi zamieniać polskich liter.

Definicja funkcji zwracającej nowy *string*, składający się z małych liter:

```
#include <cctype>
...
string NaMale(string tekst){
   for(auto znak : tekst)
      znak = tolower(znak);
   return tekst;
}
```

Kopiowanie fragmentu *stringu* do fragmentu tablicy znakowej

Funkcja zwraca liczbę skopiowanych znaków i nie dodaje na końcu znaku *null*.

*size_t copy(char *wskTab, size_t n, size_t i=0) const;*

Kopiowanie fragmentu *stringu* do fragmentu tablicy znakowej

Funkcja zwraca liczbę skopiowanych znaków i nie dodaje na końcu znaku *null*.

*size_t copy(char *wskTab, size_t n, size_t i=0) const;*

- *wskTab*, jest adresem tablicy w której ma być zapisana kopia

Kopiowanie fragmentu *stringu* do fragmentu tablicy znakowej

Funkcja zwraca liczbę skopiowanych znaków i nie dodaje na końcu znaku *null*.

*size_t copy(char *wskTab, size_t n, size_t i=0) const;*

- *wskTab*, jest adresem tablicy w której ma być zapisana kopia
- *n*, ilość znaków do skopiowania ze *stringu*

Kopiowanie fragmentu *stringu* do fragmentu tablicy znakowej

Funkcja zwraca liczbę skopiowanych znaków i nie dodaje na końcu znaku *null*.

*size_t copy(char *wskTab, size_t n, size_t i=0) const;*

- *wskTab*, jest adresem tablicy w której ma być zapisana kopia
- *n*, ilość znaków do skopiowania ze *stringu*
- *i*, indeks znaku w obiekcie klasy *string* od którego rozpoczynamy kopiowanie

Zamiana zawartości dwóch obiektów klasy *string*

Funkcja składowa klasy *string*

void string::swap(string &s) – *s* wymienia się treścią z obiektem dla którego wywołano tę funkcję.

Zamiana zawartości dwóch obiektów klasy *string*

Funkcja składowa klasy *string*

void string::swap(string &s) – *s* wymienia się treścią z obiektem dla którego wywołano tę funkcję.

Funkcja globalna spoza klasy *string*

void swap(string &s1, string &s2) – *s1* zamienia się zawartością z *s2*.

Funkcja *getline*

umożliwia wczytanie z klawiatury lub z pliku dyskowego tekstu składającego się z wielu wyrazów. Funkcja ta zwraca referencję do strumienia obsługującego wczytywanie znaków.

```
istream &getline(istream &we, string &tekst, char sep='\n')
```

Funkcja *getline*

umożliwia wczytanie z klawiatury lub z pliku dyskowego tekstu składającego się z wielu wyrazów. Funkcja ta zwraca referencję do strumienia obsługującego wczytywanie znaków.

```
istream &getline(istream &we, string &tekst, char sep='\n')
```

- *we* – strumień wejściowy określający co jest źródłem wczytywanych znaków

Funkcja *getline*

umożliwia wczytanie z klawiatury lub z pliku dyskowego tekstu składającego się z wielu wyrazów. Funkcja ta zwraca referencję do strumienia obsługującego wczytywanie znaków.

```
istream &getline(istream &we, string &tekst, char sep='\n')
```

- *we* – strumień wejściowy określający co jest źródłem wczytywanych znaków
- *tekst* – obiekt w którym jest umieszczany wczytywany tekst

Funkcja *getline*

umożliwia wczytanie z klawiatury lub z pliku dyskowego tekstu składającego się z wielu wyrazów. Funkcja ta zwraca referencję do strumienia obsługującego wczytywanie znaków.

```
istream &getline(istream &we, string &tekst, char sep='\n')
```

- *we* – strumień wejściowy określający co jest źródłem wczytywanych znaków
- *tekst* – obiekt w którym jest umieszczany wczytywany tekst
- *sep* – *ogranicznik* czyli znak oznaczający koniec wczytywanego tekstu

Funkcja *getline*

umożliwia wczytanie z klawiatury lub z pliku dyskowego tekstu składającego się z wielu wyrazów. Funkcja ta zwraca referencję do strumienia obsługującego wczytywanie znaków.

```
istream &getline(istream &we, string &tekst, char sep='\n')
```

- *we* – strumień wejściowy określający co jest źródłem wczytywanych znaków
- *tekst* – obiekt w którym jest umieszczany wczytywany tekst
- *sep* – *ogranicznik* czyli znak oznaczający koniec wczytywanego tekstu

Po napotkaniu ogranicznika, funkcja *getline* wyrzuca go, przerywa wczytywanie, a wczytane znaki umieszcza w podanym obiekcie *tekst*.

Funkcja *getline* pracująca w trybie nieformatowanym jak i operator >> pracujący w trybie formatowanym ze strumieniem *cin*, nie zacznie działania dopóki nie wciśniemy klawisza *'Enter'*.
W konsekwencji w buforze (np. klawiatury) po operacji wczytania czekają na przekazanie do kolejnej operacji wczytywania następujące znaki (łącznie z końcowym *'Enter'*):

Funkcja *getline* pracująca w trybie nieformatowanym jak i operator >> pracujący w trybie formatowanym ze strumieniem *cin*, nie zacznie działania dopóki nie wciśniemy klawisza *'Enter'*.
W konsekwencji w buforze (np. klawiatury) po operacji wczytania czekają na przekazanie do kolejnej operacji wczytywania następujące znaki (łącznie z końcowym *'Enter'*):

- dla *getline* – po ograniczniku różnym od *'\n'*, wszystkie wpisane znaki

Funkcja *getline* pracująca w trybie nieformatowanym jak i operator >> pracujący w trybie formatowanym ze strumieniem *cin*, nie zacznie działania dopóki nie wciśniemy klawisza *'Enter'*.
W konsekwencji w buforze (np. klawiatury) po operacji wczytania czekają na przekazanie do kolejnej operacji wczytywania następujące znaki (łącznie z końcowym *'Enter'*):

- dla *getline* – po ograniczniku różnym od *'\n'*, wszystkie wpisane znaki
- dla *cin* >> – wszystkie znaki następujące po ostatnim wczytanym znaku (ostatnim z pierwszego nieprzerwanego ciągu czarnych znaków)

Funkcja *getline* pracująca w trybie nieformatowanym jak i operator >> pracujący w trybie formatowanym ze strumieniem *cin*, nie zacznie działania dopóki nie wciśniemy klawisza *'Enter'*.
W konsekwencji w buforze (np. klawiatury) po operacji wczytania czekają na przekazanie do kolejnej operacji wczytywania następujące znaki (łącznie z końcowym *'Enter'*):

- dla *getline* – po ograniczniku różnym od *'\n'*, wszystkie wpisane znaki
- dla *cin* >> – wszystkie znaki następujące po ostatnim wczytanym znaku (ostatnim z pierwszego nieprzerwanego ciągu czarnych znaków)

W odróżnieniu od funkcji *getline*, domyślnie dla strumienia wejściowego operator >> przeskakuje wstępne białe znaki.
Gdy w buforze strumienia wejściowego znajduje się ciąg znaków, ewentualna funkcja *getline* automatycznie podejmie próbę ich wczytania aż do jej ogranicznika, zaś operator >> zignoruje wszystkie białe znaki.

Jeśli znajdujące się w buforze znaki chcemy zignorować, powinniśmy te znaki usunąć.
Przed następującym po operatorze >>, użyciem funkcji *getline* można usunąć znaki poprzez:

Jeśli znajdujące się w buforze znaki chcemy zignorować, powinniśmy te znaki usunąć.
Przed następującym po operatorze >>, użyciem funkcji *getline* można usunąć znaki poprzez:

- wstawienie do strumienia manipulatora *ws* – usuwa w buforze białe znaki do pierwszego czarnego np. *cin >> ws; getline(cin, tekst, '\n');* lub *getline((cin >> ws), tekst, '\n');*

Jeśli znajdujące się w buforze znaki chcemy zignorować, powinniśmy te znaki usunąć.
Przed następującym po operatorze >>, użyciem funkcji *getline* można usunąć znaki poprzez:

- wstawienie do strumienia manipulatora *ws* – usuwa w buforze białe znaki do pierwszego czarnego np. *cin >> ws; getline(cin, tekst, '\n');* lub *getline((cin >> ws), tekst, '\n');*
- wywołanie dla strumienia funkcji *ignore* – usuwa w buforze wszystkie znaki w ilości do podanej liczby lub do podanego ogranicznika (który również usuwa) np. *cin.ignore(100, '\n');* lub dla dużej nieznanej liczby *cin.ignore(numeric_limits<streamsize>::max(), '\n');*

Jeśli znajdujące się w buforze znaki chcemy zignorować, powinniśmy te znaki usunąć.
Przed następującym po operatorze >>, użyciem funkcji *getline* można usunąć znaki poprzez:

- wstawienie do strumienia manipulatora *ws* – usuwa w buforze białe znaki do pierwszego czarnego np. *cin >> ws; getline(cin, tekst, '\n');* lub *getline((cin >> ws), tekst, '\n');*
- wywołanie dla strumienia funkcji *ignore* – usuwa w buforze wszystkie znaki w ilości do podanej liczby lub do podanego ogranicznika (który również usuwa) np. *cin.ignore(100, '\n');* lub dla dużej nieznanej liczby *cin.ignore(numeric_limits<streamsize>::max(), '\n');*
- poprzedzające wywołanie *getline* – wyczyści bufor ze znaków aż do ogranicznika włącznie np. *getline(cin, tekst, '\n');*

Iteratory stringu

Ze względu na zgodność z wieloma funkcjami bibliotecznymi bazującymi na wskaźnikach do napisów, w obiektach klasy *string* wprowadzono **iteratory**. Mogą one pokazywać na znak obiektu klasy *string* np.

```
string::iterator iZnak;
```

Iteratory stringu

Ze względu na zgodność z wieloma funkcjami bibliotecznymi bazującymi na wskaźnikach do napisów, w obiektach klasy *string* wprowadzono **iteratory**. Mogą one pokazywać na znak obiektu klasy *string* np.

```
string::iterator iZnak;
```

Operatory dla *iteratorów*

w znacznej większości działają w identyczny sposób jak na wskaźniki.

Iteratory stringu

Ze względu na zgodność z wieloma funkcjami bibliotecznymi bazującymi na wskaźnikach do napisów, w obiektach klasy *string* wprowadzono **iteratory**. Mogą one pokazywać na znak obiektu klasy *string* np.

```
string::iterator iZnak;
```

Operatory dla *iteratorów*

w znacznej większości działają w identyczny sposób jak na wskaźniki.

- **iZnak* – zwraca znak pokazywany przez *iterator iZnak*

Iteratory stringu

Ze względu na zgodność z wieloma funkcjami bibliotecznymi bazującymi na wskaźnikach do napisów, w obiektach klasy *string* wprowadzono **iteratory**. Mogą one pokazywać na znak obiektu klasy *string* np.

```
string::iterator iZnak;
```

Operatory dla *iteratorów*

w znacznej większości działają w identyczny sposób jak na wskaźniki.

- **iZnak* – zwraca znak pokazywany przez *iterator iZnak*
- „*++*”, „*−−*” – przesuwa wskazanie iteratora na sąsiedni znak *stringu*

Iteratory stringu

Ze względu na zgodność z wieloma funkcjami bibliotecznymi bazującymi na wskaźnikach do napisów, w obiektach klasy *string* wprowadzono **iteratory**. Mogą one pokazywać na znak obiektu klasy *string* np.

```
string::iterator iZnak;
```

Operatory dla *iteratorów*

w znacznej większości działają w identyczny sposób jak na wskaźniki.

- **iZnak* – zwraca znak pokazywany przez *iterator iZnak*
- „*++*”, „*−−*” – przesuwa wskazanie iteratora na sąsiedni znak *stringu*
- „*+=*”, „*−=*”, „*+*”, „*−*” – przesuwa wskazanie iteratora o podaną liczbę znaków

Iterator do obiektu stałego

jest obiektem innej klasy *iteratorów* wskazującym znak obiektu *string* z kwalifikatorem *const*. Sam może zmieniać wartość np.

```
string::const_iterator iCznak;
```

Funkcje klasy *string* pracujące z *iteratorami*

Funkcje **begin** i **end**

zwracają *iterator* odpowiednio do początkowego znaku i do miejsca bespośrednio za końcowym znakiem obiektu klasy *string* czyli *null*, np.

```
string tytul("W pustyni i w puszczy");
iZnak = tytul.begin() + 2;
while(iZnak != tytul.end()) *iZnak++ = '*';
```

Funkcje klasy *string* pracujące z *iteratorami*

Funkcje **begin** i **end**

zwracają *iterator* odpowiednio do początkowego znaku i do miejsca bespośrednio za końcowym znakiem obiektu klasy *string* czyli *null*, np.

```
string tytul("W pustyni i w puszczy");
iZnak = tytul.begin() + 2;
while(iZnak != tytul.end()) *iZnak++ = '*';
```

Konstruktor klasy *string*

```
string(iterator i1, iterator i2);
```

inicjalizujący fragmentem innego stringu, od znaku pokazywanego iteratorem *i1* do znaku bezpośrednio poprzedzającego wskazanie *i2* np.

```
string imie("Magdalena");
string zdrobnienie(imie.begin(), imie.begin()+5);
```

Funkcje **erase** usuwające znaki *stringu*

zwracają *iterator* do znaku bespośrednio następującego po skasowanych.

Funkcje **erase** usuwające znaki *stringu*

zwracają *iterator* do znaku bespośrednio następującego po skasowanych.

iterator erase(const_iterator i);

usuwa z obiektu znak wskazany iteratorem *i*.

Funkcje **erase** usuwające znaki *stringu*

zwracają *iterator* do znaku bespośrednio następującego po skasowanych.

_iterator erase(const_iterator i);_

usuwa z obiektu znak wskazany iteratorem *i*.

_iterator erase(const_iterator i1, const_iterator i2);_

usuwa z obiektu znaki z przedziału wskazanego iteratorami *[i1; i2)*.
W praktyce zwraca wartość iteratora *i2*. Gdy skasowane zostały by znaki do końca *stringu*, zwrócona została by wartość *string::end()*.

Funkcje **insert** wstawiające znaki do *stringu*

iterator insert(const iterator i, const char znak);_

wstawia *znak* do obiektu w miejscu wskazanym iteratorem *i*.
Funkcja zwraca *iterator* do wstawionego znaku.

Funkcje **insert** wstawiające znaki do *stringu*

iterator insert(const_iterator i, const char znak);

wstawia *znak* do obiektu w miejscu wskazanym iteratorem *i*.
Funkcja zwraca *iterator* do wstawionego znaku.

iterator insert(const_iterator i, size_t n, char znak);

wstawia *n* razy *znak* do obiektu w miejscu wskazanym iteratorem *i*.

Funkcje **insert** wstawiające znaki do *stringu*

iterator insert(const_iterator i, const char znak);

wstawia *znak* do obiektu w miejscu wskazanym iteratorem *i*.
Funkcja zwraca *iterator* do wstawionego znaku.

iterator insert(const_iterator i, size_t n, char znak);

wstawia *n* razy *znak* do obiektu w miejscu wskazanym iteratorem *i*.

iterator insert(iterator i0, const_iterator i1, const_iterator i2);

wstawia do obiektu w miejscu wskazanym iteratorem *i0*, znaki z fragmentu innego *stringu* wskazane przedziałem *[i1; i2)*.

Funkcje **replace** zamieniające część znaków *stringu*

Pierwsze 2 argumenty typu *const_iterator* wyznaczają wymieniany fragment stringu:

Funkcje **replace** zamieniające część znaków *stringu*

Pierwsze 2 argumenty typu *const_iterator* wyznaczają wymieniany fragment stringu:

- *i1* – iterator wskazujący początkowy znak do wymiany

Funkcje **replace** zamieniające część znaków *stringu*

Pierwsze 2 argumenty typu *const_iterator* wyznaczają wymieniany fragment stringu:

- *i1* – iterator wskazujący początkowy znak do wymiany
- *i2* – iterator wskazujący miejsce bezpośrednio następujące za ostatnim wymienianym znakiem

Funkcje **replace** zamieniające część znaków *stringu*

Pierwsze 2 argumenty typu *const_iterator* wyznaczają wymieniany fragment stringu:

- *i1* – iterator wskazujący początkowy znak do wymiany
- *i2* – iterator wskazujący miejsce bezpośrednio następujące za ostatnim wymienianym znakiem

string &replace(..., ..., const string &tekst);

miejsce wyznaczone poprzez iteratory *[i1; i2)* zamienia obiektem *tekst*.

Funkcje **replace** zamieniające część znaków *stringu*

Pierwsze 2 argumenty typu *const_iterator* wyznaczają wymieniany fragment stringu:

- *i1* – iterator wskazujący początkowy znak do wymiany
- *i2* – iterator wskazujący miejsce bezpośrednio następujące za ostatnim wymienianym znakiem

string &replace(..., ..., const string &tekst);

miejsce wyznaczone poprzez iteratory *[i1; i2)* zamienia obiektem *tekst*.

string &replace(..., ..., const char *tekst, size_t n);

zamienia fragmentem *n* kolejnych znaków *C-stringu* o początku danym wskaźnikiem *tekst*.

*string &replace(..., ..., const char *tekst);*

zamienia *C-stringiem* o początku danym wskaźnikiem *tekst*.

*string &replace(..., ..., const char *tekst);*

zamienia *C-stringiem* o początku danym wskaźnikiem *tekst*.

string &replace(..., ..., size_t n, const char znak);

zamienia *n* znakami *znak*.

*string &replace(..., ..., const char *tekst);*

zamienia *C-stringiem* o początku danym wskaźnikiem *tekst*.

string &replace(..., ..., size_t n, const char znak);

zamienia *n* znakami *znak*.

```
string pesel("48110258614");
string::iterator i1 = pesel.begin()+2;
string::iterator i2 = i1+7;
pesel.replace(i1, i2, 7, '*'); // "48*******14"
```

*string &replace(..., ..., const char *tekst);*

zamienia *C-stringiem* o początku danym wskaźnikiem *tekst*.

string &replace(..., ..., size_t n, const char znak);

zamienia *n* znakami *znak*.

```
string pesel("48110258614");
string::iterator i1 = pesel.begin()+2;
string::iterator i2 = i1+7;
pesel.replace(i1, i2, 7, '*'); // "48*******14"
```

string &replace(..., ..., std::initializer_list<char> lista);

zamienia listą inicjalizacyjną w nawiasach klamrowych: *lista*.

Technika przenoszenia funkcją **std::move**

przyspiesza działanie programu, poprzez poddanie recyklingowi obiektów będących argumentami funkcji *move*. Operacja przenoszenia nie niszczy obiektów, lecz ich treść mogła zostać usunięta, np. roboczy wynik sumy zamiast kopiowania, jest wprost przenoszony do wyniku:

Technika przenoszenia funkcją **std::move**

przyspiesza działanie programu, poprzez poddanie recyklingowi obiektów będących argumentami funkcji *move*. Operacja przenoszenia nie niszczy obiektów, lecz ich treść mogła zostać usunięta, np. roboczy wynik sumy zamiast kopiowania, jest wprost przenoszony do wyniku:

```
string nazwa("tomasz");
string domena("@wp.pl");
string adres = move(nazwa) + move(domena);
// obiekty nazwa lub domena mogą być teraz puste
```

Obsługa wyjątków

Mechanizm rzucenia wyjątku

wykorzystujemy gdy wykonujemy zadanie, które może nie zakończyć się powodzeniem. Jest to również inny sposób wyjścia z funkcji.
Takie zadanie umieszczamy w bloku **try**.
Powinniśmy wtedy przygotować co najmniej jedno miejsce (blok **catch**), które złapie wyjątek danego typu – rzucony instrukcją **throw**.

Obsługa wyjątków

Mechanizm rzucenia wyjątku

wykorzystujemy gdy wykonujemy zadanie, które może nie zakończyć się powodzeniem. Jest to również inny sposób wyjścia z funkcji.
Takie zadanie umieszczamy w bloku **try**.
Powinniśmy wtedy przygotować co najmniej jedno miejsce (blok **catch**), które złapie wyjątek danego typu – rzucony instrukcją **throw**.

Taki typ wyjątku można samodzielnie przygotować definiując pustą klasę o danej nazwie. Rzucamy wtedy wcześniej zdefiniowany obiekt takiej klasy. Można rzucić wyjątek typu wbudowanego, co nie jest zalecane. Dostępne są również standardowe typy wyjątków.

```
class Blad_typu_A { };
class Blad_typu_B { };
void fun(int a){
   Blad_typu_A blA;
   Blad_typu_B blB;
   if(a==0) throw blA;
   if(a<0) throw blB;
}
try{
   fun(4); // ryzykowne zadania
}
catch(Blad_typu_A blA){
   // instrukcje w przypadku niepowodzenia
}
catch(Blad_typu_B blB){
   // instrukcje w przypadku niepowodzenia
}
```

Kolejność bloków *catch*

ma znaczenie – pierwszy który się nadaje, przechwyci dany wyjątek. Gdy niektóre typy obiektów wyjątków nadają się dla więcej niż jednego bloku *catch*, powinniśmy je ustawić od najbardziej szczegółowych do bardziej ogólnych.

Kolejność bloków *catch*

ma znaczenie – pierwszy który się nadaje, przechwyci dany wyjątek. Gdy niektóre typy obiektów wyjątków nadają się dla więcej niż jednego bloku *catch*, powinniśmy je ustawić od najbardziej szczegółowych do bardziej ogólnych.

Dany blok *catch* nadaje się do obsługi danego typu wyjątku gdy:

Kolejność bloków *catch*

ma znaczenie – pierwszy który się nadaje, przechwyci dany wyjątek. Gdy niektóre typy obiektów wyjątków nadają się dla więcej niż jednego bloku *catch*, powinniśmy je ustawić od najbardziej szczegółowych do bardziej ogólnych.

Dany blok *catch* nadaje się do obsługi danego typu wyjątku gdy:

- typ obiektu wyjątku jest taki sam jak typ argumentu oczekiwanego przez *catch*, lub ma dodatkowo specyfikator *const* lub oczekuje referencji do tego typu.

Kolejność bloków *catch*

ma znaczenie – pierwszy który się nadaje, przechwyci dany wyjątek. Gdy niektóre typy obiektów wyjątków nadają się dla więcej niż jednego bloku *catch*, powinniśmy je ustawić od najbardziej szczegółowych do bardziej ogólnych.

Dany blok *catch* nadaje się do obsługi danego typu wyjątku gdy:

- typ obiektu wyjątku jest taki sam jak typ argumentu oczekiwanego przez *catch*, lub ma dodatkowo specyfikator *const* lub oczekuje referencji do tego typu.
- dla tego danego wyjątku, podany typ jest publiczną klasą podstawową. Zasada ta nie obowiązuje przy dziedziczeniu prywatnym.

Kolejność bloków *catch*

ma znaczenie – pierwszy który się nadaje, przechwyci dany wyjątek. Gdy niektóre typy obiektów wyjątków nadają się dla więcej niż jednego bloku *catch*, powinniśmy je ustawić od najbardziej szczegółowych do bardziej ogólnych.

Dany blok *catch* nadaje się do obsługi danego typu wyjątku gdy:

- typ obiektu wyjątku jest taki sam jak typ argumentu oczekiwanego przez *catch*, lub ma dodatkowo specyfikator *const* lub oczekuje referencji do tego typu.
- dla tego danego wyjątku, podany typ jest publiczną klasą podstawową. Zasada ta nie obowiązuje przy dziedziczeniu prywatnym.
- możliwa jest konwersja standardowa wskaźnika (referencji) do przychodzącego typu pochodnego wyjątku, na wskaźnik (referencję) do łapanego typu podstawowego.

Zagnieżdżanie bloków *try*

W bloku *try* można umieścić inny blok *try* wraz z jego własnymi blokami *catch*. Łapanie wyjątków wykonuje się wtedy w naturalny sposób od najbardziej zagnieżdżonych bloków *catch*. Możliwe jest np. zagnieżdżanie poprzez wywoływanie funkcji, zawierających po jednym bloku *try*.
Blok *try* może być również zagnieżdżony w bloku *catch*.

Zagnieżdżanie bloków *try*

W bloku *try* można umieścić inny blok *try* wraz z jego własnymi blokami *catch*. Łapanie wyjątków wykonuje się wtedy w naturalny sposób od najbardziej zagnieżdżonych bloków *catch*. Możliwe jest np. zagnieżdżanie poprzez wywoływanie funkcji, zawierających po jednym bloku *try*.
Blok *try* może być również zagnieżdżony w bloku *catch*.

Jeśli w żadnym bloku *catch* wyjątek nie zostanie złapany, zostaje uruchomiona standardowa funkcja *terminate*, która zakończy program.
W takim przypadku nie zostaną uruchomione nawet destruktory obiektów naszego programu. Na ekranie zostaje wyświetkona informacja o typie nie złapanego wyjątku.

Złapany przy pomocy *catch* wyjątek uznany zostaje za obsłużony.
Instrukcja *throw* bez argumentu, wewnątrz bloku *catch* wykonuje powtórne rzucenie właśnie złapanego wyjątku.
Blok **catch(...)** łapie każdy obiekt wyjątku, więc stawiany jest jako ostatni.

Złapany przy pomocy *catch* wyjątek uznany zostaje za obsłużony.
Instrukcja *throw* bez argumentu, wewnątrz bloku *catch* wykonuje powtórne rzucenie właśnie złapanego wyjątku.
Blok **catch(...)** łapie każdy obiekt wyjątku, więc stawiany jest jako ostatni.

Specyfikator **noexcept**

umieszczony w deklaracji funkcji, upraszcza i optymalizuje kod jej wywołania, ponieważ ewentualnie rzucane wyjątki nie będą obsługiwane.
W przypadku rzucenia wyjątku uruchomiona zostanie funkcja *std::terminate*, kończąca działanie programu.

noexcept może wystąpić samo lub z argumentem w nawiasie typu *constexpr bool*, np.

```
int f(int); // lub int f(int) noexcept(false);
// z obsługą wyjątków:
int f(char) noexcept; // lub auto f(char) noexcept -> int;
// lub int f(char) noexcept(true);
```

noexcept może wystąpić samo lub z argumentem w nawiasie typu *constexpr bool*, np.

```
int f(int); // lub int f(int) noexcept(false);
// z obsługą wyjątków:
int f(char) noexcept; // lub auto f(char) noexcept -> int;
// lub int f(char) noexcept(true);
```

Operator **noexcept**

zwraca wartość typu *constexpr bool*. Czyli jeszcze podczas kompilacji zostaje sprawdzony argument operatora pod względem obsługi wyjątków. Przykład argumentu jako wyrażenia – ewentualnego wywołania funkcji:

```
constexpr bool w1 = noexcept(f(2)); // false
constexpr bool w2 = noexcept(f('c')); // true
```

Powyższe argumenty wyglądające jak wywołania funkcji, nie są obliczane – sprawdzane są jedynie deklaracje tych funkcji.

Przykład wykorzystania specyfikatora i operatora *noexcept* w deklaracji funkcji, aby miała to samo podejście co do obsługi wujątków:

```
void f1(float) noexcept( noexcept(f(int)) );
// lub f1(float) noexcept(w1);
```

Przykład wykorzystania specyfikatora i operatora *noexcept* w deklaracji funkcji, aby miała to samo podejście co do obsługi wujątków:

```
void f1(float) noexcept( noexcept(f(int)) );
// lub f1(float) noexcept(w1);
```

Specyfikator użyty w deklaracji funkcji nie jest częścią typu funkcji, lecz instrukcją postępowania dla kompilatora. Nie stosuje się go w instrukcjach *alias* czy *typedef*.

Literatura


Jerzy Grębosz, Opus magnum C++11. Programowanie w języku C++. Tomy 1, 2 i 3, Helion (2018).


Working Draft, Standard for Programming Language C++ `http://open-std.org/JTC1/SC22/WG21/docs/standards`


Brian Kernighan, Dennis Ritchie, Język ANSI C, Wydawnictwa Naukowo-Techniczne, Warszawa (1994)