Wstęp do programowania, część II

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

8 listopada 2011

Sprawdzanie warunków – `if ()`

```
if (warunek)
    instrukcja;  // A
else
    instrukcja;  // B
```

```
if (warunek) {  // A
    instrukcja;
    ...
} else {  // B
    instrukcja;
    ...
}
```

Sprawdzanie warunków – `if ()`

```
if (warunek)
    instrukcja;  // A
else
    instrukcja;  // B
```

```
if (warunek) {  // A
    instrukcja;
    ...
} else {  // B
    instrukcja;
    ...
}
```

1. Jeżeli `warunek` ma wartość `true`, zostanie wykonana instrukcja (lub blok) A, zaś w przeciwnym wypadku – instrukcja (lub blok) B.
2. Warunek powinien być wyrażeniem o wartości typu `bool`. Jeżeli tak nie jest, następuje konwersja wyniku do typu `bool` (zgodnie z zasadami przedstawionymi wcześniej).

Sprawdzanie warunków – `if ()` c. d.

Klauzula (*ang. clause*) `else` oraz instrukcja lub blok występujący po niej są opcjonalne.

Przykład

```
x = funkcja(arg);
if (x < 0)
    x = 0;
```

Sprawdzanie warunków – `if ()` c. d.

Klauzula (*ang. clause*) `else` oraz instrukcja lub blok występujący po niej są opcjonalne.

Przykład

```
x = funkcja(arg);
if (x < 0)
    x = 0;
```

W niektórych przypadkach użycie trójargumentowego operatora prowadzi do bardziej zwartego kodu.

Przykład

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

Pętla while ()

```
while (warunek)
    instrukcja;
```

```
while (warunek) {
    instrukcja;
    ...
}
```

Pętla `while ()`

```
while (warunek)
    instrukcja;
```

```
while (warunek) {
    instrukcja;
    ...
}
```

1. Powtarzaj instrukcję lub blok tak długo, jak długo `warunek` ma wartość `true`.
2. Warunek powinien być wyrażeniem o wartości typu `bool`. Jeżeli tak nie jest, następuje konwersja wyniku do typu `bool` (zgodnie z zasadami przedstawionymi wcześniej).

Pętla do ... while ()

```
do
    instrukcja;
while (warunek)
```

```
do {
    instrukcja;
    ...
} while (warunek);
```

Pętla do ... while ()

```
do
    instrukcja;
while (warunek)
```

```
do {
    instrukcja;
    ...
} while (warunek);
```

1. Powtarzaj instrukcję lub blok tak długo, jak długo `warunek` ma wartość `true`.
2. Warunek powinien być wyrażeniem o wartości typu `bool`. Jeżeli tak nie jest, następuje konwersja wyniku do typu `bool` (zgodnie z zasadami przedstawionymi wcześniej).
3. Różni się od `while ()` tym, że instrukcja (lub blok) będzie wykonana co najmniej raz, niezależnie od początkowej wartości warunku.

Pętla `for ()`

```
for (ins1; war; ins2)
    instrukcja;
```

```
for (ins1; war; ins2) {
    instrukcja;
    ...
}
```

Pętla `for ()`

```
for (ins1; war; ins2)
    instrukcja;
```

```
for (ins1; war; ins2) {
    instrukcja;
    ...
}
```

1. Wykonaj instrukcję `ins1`;
2. Oblicz `war` i jeżeli ma on wartość `false` lub 0, przerwij pętlę.
3. Wykonaj instrukcję (lub blok) za nawiasem.
4. Wykonaj instrukcję `ins2`.
5. Przejdź do kroku 2.

Pętle `for ()` i `while ()`

```
for (ins1; war; ins2) {
    instrukcja;
    ...
}
```

```
ins1;
while (war) {
    instrukcja;
    ...
    ins2;
}
```

Powyższe dwa sposoby zapisu kodu pętli są równoważne.

Pętle `for ()` i `while ()` c. d.

Przykład

```
for (int i = 1; i <= 10; i++)
    cout << i << "^2 = " << i*i << endl;
```

```
int i = 1;
while (i <= 10) {
    cout << i << "^2 = " << i*i << endl;
    i++;
}
```

W jednym i w drugim przypadku zostaną wydrukowane kwadraty liczb od 1 do 10.

Zasięg deklaracji zmiennych

Zasady określania zasięgu (*ang. scope*) deklaracji

1. Zmienna (lub stała) zadeklarowana wewnątrz bloku nie może być wykorzystywana poza tym blokiem.
2. Zmienna zadeklarowana w pierwszej instrukcji pętli `for ()` jest traktowana tak, jakby została zadeklarowana wewnątrz bloku obejmującego tę pętlę.
3. Instrukcje zapisane przed deklaracją zmiennej nie mogą zawierać odwołań do niej.
4. Jeżeli w bloku zadeklarowana jest zmienna o takiej samej nazwie, jaką ma zmienna zadeklarowana poza tym blokiem, to instrukcje w bloku będą odwoływać się w pierwszej kolejności do zmiennej zadeklarowanej wewnątrz bloku (przesłanianie deklaracji).

"Skracanie" kroku pętli – `continue`

`continue`

Powoduje przejście do następnego kroku pętli:

- Dla pętli `while ()` i `do ... while()` powoduje przejście do sprawdzania warunku.
- Dla pętli `for ()` powoduje przejście do kroku 4 w algorytmie wykonywania jej (wykonywanie instrukcji w trzecim polu w nawiasie).

"Skracanie" kroku pętli – continue

continue

Powoduje przejście do następnego kroku pętli:

- Dla pętli `while ()` i `do ... while()` powoduje przejście do sprawdzania warunku.
- Dla pętli `for ()` powoduje przejście do kroku 4 w algorytmie wykonywania jej (wykonywanie instrukcji w trzecim polu w nawiasie).

Przykład

```
for (j = 1; j <= 20; j++) {
    if (j % 2)
        continue;
    count << j << "^2 = " << j*j << endl;
}
```

"Skracanie" pętli – `break`

`break`

Powoduje natychmiastowe przerwanie wykonywania pętli.

"Skracanie" pętli – break

break

Powoduje natychmiastowe przerwanie wykonywania pętli.

Przykład

```
k = 0;
a_k = 1;
suma = 1;
while (k < N) {
    k++;
    a_k /= k;
    suma += a_k;
    if (a_k < epsilon)
        break;
}
```

Instrukcja wyboru – `switch`

1. Słowo kluczowe `switch`.
2. Wyrażenie w nawiasie okrągłym.
3. Początek bloku.
4. Lista wartości (stałych) do porównania z wyrażeniem, każda ze słowem kluczowym `case` (np. `case 100:`).
5. Opcjonalnie słowo kluczowe `default` oznaczające domyślną akcję.
6. Koniec bloku.

- Jeżeli wartość wyrażenia jest równa wartości za słowem kluczowym `case`, to następuje przeskok do miejsca w bloku, gdzie znajduje się słowo kluczowe `case` z tą wartością.
- W przypadku braku dopasowania następuje przeskok do miejsca w bloku oznaczonego przez `default`.
- `break` powoduje natychmiastowe opuszczenie bloku.

switch - przykład

```
switch (req) {
case RPM_REQ_NONE:
    break;
case RPM_REQ_IDLE:
    rpm_idle(dev, RPM_NOWAIT);
    break;
case RPM_REQ_SUSPEND:
    rpm_suspend(dev, RPM_NOWAIT);
    break;
case RPM_REQ_AUTOSUSPEND:
    rpm_suspend(dev, RPM_NOWAIT | RPM_AUTO);
    break;
case RPM_REQ_RESUME:
    rpm_resume(dev, RPM_NOWAIT);
}
```

Skoki (*ang. jump*) – goto

```
    suma = a_k = 1;
    for (k = 0; k < N; suma += a_k) {
        if (a_k < epsilon)
            goto dalej;
        a_k /= ++k;
    }
dalej:
```

`dalej:` jest etykietą (*ang. label*) oznaczającą miejsce, od którego należy kontynuować wykonywanie programu.

Instrukcji `goto` nie można używać do „przeskoków" między funkcjami (tzn. docelowa etykieta musi być w treści tej samej funkcji, w której jest instrukcja `goto` prowadząca do niej).

Łączenie wyrażeń – , (przecinek)

, (przecinek)

1. Oblicz wyrażenie (lub wykonaj instrukcję) po lewej stronie przecinka (łącznie ze wszystkimi efektami ubocznymi).
2. Zaniedbaj wynik obliczonego wyrażenia.
3. Oblicz wyrażenie (lub wykonaj instrukcję) po prawej stronie przecinka.

Łączenie wyrażeń – , (przecinek)

, (przecinek)

1. Oblicz wyrażenie (lub wykonaj instrukcję) po lewej stronie przecinka (łącznie ze wszystkimi efektami ubocznymi).
2. Zaniedbaj wynik obliczonego wyrażenia.
3. Oblicz wyrażenie (lub wykonaj instrukcję) po prawej stronie przecinka.

Przykład

Następujący kod spowoduje wydrukowanie liczb 20 i 30 (w tej kolejności):

```
int i, b = 20, c= 30;
i = b, c;
cout << i << endl;
i = (b, c);
cout << i << endl;
```

Definiowanie funkcji z argumentami

Argumenty funkcji

1. Definiuje się w nagłówku funkcji, w nawiasie okrągłym za nazwą.
2. Ich definicje rozdziela się przecinkami.
3. Są definiowane podobnie, jak zmienne wewnątrz funkcji.
4. Mogą być wykorzystywane jako zmienne wewnątrz funkcji.

Definiowanie funkcji z argumentami

Argumenty funkcji

1. Definiuje się w nagłówku funkcji, w nawiasie okrągłym za nazwą.
2. Ich definicje rozdziela się przecinkami.
3. Są definiowane podobnie, jak zmienne wewnątrz funkcji.
4. Mogą być wykorzystywane jako zmienne wewnątrz funkcji.

Przykład – funkcja z jednym argumentem

```
double f(double x)
{
    return x*x + 1;
}
```

Zwraca wynik będący kwadratem jej argumentu zwiększonym o 1.

Wywołanie funkcji (*ang. function call*)

Na poziomie kodu źródłowego

Polega na umieszczeniu w instrukcji nazwy funkcji z listą wartości parametrów w nawiasie okrągłym.

Wywołanie funkcji (*ang. function call*)

Na poziomie kodu źródłowego

Polega na umieszczeniu w instrukcji nazwy funkcji z listą wartości parametrów w nawiasie okrągłym.

Podczas wykonywania programu

Kod (tzn. rozkazy dla procesora) wygenerowany na podstawie instrukcji zawartych w treści funkcji jest wykonywany w punkcie programu, w którym w kodzie źródłowym była umieszczona nazwa funkcji.

Wywołanie funkcji (*ang. function call*)

Na poziomie kodu źródłowego

Polega na umieszczeniu w instrukcji nazwy funkcji z listą wartości parametrów w nawiasie okrągłym.

Podczas wykonywania programu

Kod (tzn. rozkazy dla procesora) wygenerowany na podstawie instrukcji zawartych w treści funkcji jest wykonywany w punkcie programu, w którym w kodzie źródłowym była umieszczona nazwa funkcji.

Podczas wykonywania programu

Wynik generowany przez funkcję jest wykorzystywany w wyrażeniu w tym miejscu, w którym następuje wywołanie funkcji. Za każdym razem wygenerowanie wyniku następuje w efekcie wykonania kodu funkcji (kodu utworzonego na podstawie instrukcji zapisanych w treści funkcji).

Przekazywanie wartości argumentów do funkcji

Podczas wykonywania programu

1. Tworzone są zmienne, których nazwy i typy danych odpowiadają nazwom i typom danych argumentów funkcji.
2. Wartości znajdujące się na pozycjach odpowiadających argumentom w zapisie wywołania funkcji (np. `suma += sin(a);`) stają się początkowymi wartościami tych zmiennych.
3. Wykonywany jest kod reprezentowany przez instrukcje w treści funkcji, odwołujący się do tych zmiennych.
4. Wykonanie tego kodu może skończyć się wygenerowaniem wyniku, który jest wykorzystywany w sposób określony przez kod wywołujący funkcję.

Przekazywanie wartości argumentów do funkcji – przykład

Całka trapezowa dla funkcji `f()` z poprzedniego przykładu.

```
double trapez(double a, double b, int N)
{
    double delta, suma, x_j;
    int j;

    suma = (f(a) + f(b)) / 2;
    delta = (b - a) / N;
    for (j = 1, x_j = a; j < N; j++) {
        x_j += delta;
        suma += f(x_j);
    }
    suma *= delta;
    return suma;
}
```

Funkcje biblioteczne (*ang. library function*)

„Gotowe” funkcje, które zostały napisane oraz skompilowane przez kogoś innego i znajdują się w specjalnych zbiorach funkcji, zwanych bibliotekami (*ang. library*).

Przykłady

`sin()`, `cos()`, `exp()`, `sqrt()` (pierwiastek kwadratowy).

Funkcje biblioteczne (*ang. library function*)

„Gotowe” funkcje, które zostały napisane oraz skompilowane przez kogoś innego i znajdują się w specjalnych zbiorach funkcji, zwanych bibliotekami (*ang. library*).

Przykłady

`sin()`, `cos()`, `exp()`, `sqrt()` (pierwiastek kwadratowy).

Pliki nagłówkowe (*ang. header files*)

Pliki zawierające, między innymi, nagłówki funkcji bibliotecznych, które mogą być wykorzystane w programie. Na podstawie nagłówków kompilator znajduje odpowiednie funkcje w bibliotekach.

Nagłówki funkcji „matematycznych” znajdują się w pliku nagłówkowym `cmath`.

Stałe (*ang. constant*)

`const`

Słowo kluczowe używane przy definiowaniu stałych. Po nim podaje się typ danych, nazwę oraz wartość stałej.

Przykład

```
const double pi = 3.14159265358979323846;
```

W C++ stałe mają własności analogiczne do zmiennych poza tym, że ich wartości nie zmieniają się.

Argument funkcji może być zadeklarowany jako stała i wtedy jego wartość nie może być mdyfikowana wewnątrz funkcji (otrzymuje on wartość przy wywoływaniu funkcji).

Referencje (*ang. reference*)

Referencja jest symbolem definiowanym podobnie, jak zmienna, ale w jej definicji, bezpośrednio przed nazwą, znajduję się znak & (*ang. ampersand*).

Przykład

```
double &ref = x;
```

Wartością referencji jest zmienna odpowiedniego typu, wskazana w jej definicji (np. `x` powyżej).

Referencję można traktować jako alternatywną nazwę (alias) zmiennej.

Referencje i argumenty funkcji

Argument funkcji może być zadeklarowany jako referencja i wtedy:

1. W wywołaniu funkcji na pozycji odpowiadającej temu argumentowi musi stać nazwa zmiennej odpowiedniego typu (nie może to być wyrażenie).
2. Zmienna ta będzie używana w treści funkcji bezpośrednio w miejscach, w których następują odwołania do reprezentującego ją argumentu.

Przykład

```
double oblicz(double r, int &n)
{
    return r / ++n;
}
```

Referencje i argumenty funkcji – przykład

Wywołanie funkcji `oblicz()`

```
k = 0;
a_k = 1;
suma = 1;
while (k < N) {
    a_k = oblicz(a_k, k);
    suma += a_k;
    if (a_k < epsilon)
        break;
}
```

Zmienna `k` jest modyfikowana przez funkcję `oblicz()` (występuje w niej pod nazwą `n`).

Tablice, elementy, indeksy

Tablica (*ang. array*)

Zestaw N zmiennych tego samego typu numerowanych liczbami w zakresie od 0 do $(N-1)$.

Tablice, elementy, indeksy

Tablica (*ang. array*)

Zestaw N zmiennych tego samego typu numerowanych liczbami w zakresie od 0 do $(N-1)$.

Element tablicy

Zmienna wchodząca w skład tablicy, mająca przypisaną określoną liczbę, która ją identyfikuje.

Tablice, elementy, indeksy

Tablica (*ang. array*)

Zestaw N zmiennych tego samego typu numerowanych liczbami w zakresie od 0 do $(N-1)$.

Element tablicy

Zmienna wchodząca w skład tablicy, mająca przypisaną określoną liczbę, która ją identyfikuje.

Indeks (*ang. index*) elmentu tablicy

Liczba identyfikująca element tablicy.

Tablice, elementy, indeksy

Tablica (*ang. array*)

Zestaw N zmiennych tego samego typu numerowanych liczbami w zakresie od 0 do $(N-1)$.

Element tablicy

Zmienna wchodząca w skład tablicy, mająca przypisaną określoną liczbę, która ją identyfikuje.

Indeks (*ang. index*) elmentu tablicy

Liczba identyfikująca element tablicy.

Długość tablicy

Liczba elementów tablicy (N).

Rozmieszczenie elementów w pamięci, nazwa tablicy

W C++ zakłada się, że elementy tablicy będą zawsze rozmieszczane w pamięci jeden obok drugiego, zgodnie z kolejnością indeksów (tzn. element o najmniejszym indeksie będzie znajdował się w pamięci w lokacji o najniższym adresie).

Rozmieszczenie elementów w pamięci, nazwa tablicy

W C++ zakłada się, że elementy tablicy będą zawsze rozmieszczane w pamięci jeden obok drugiego, zgodnie z kolejnością indeksów (tzn. element o najmniejszym indeksie będzie znajdował się w pamięci w lokacji o najniższym adresie).

Nazwa tablicy

Jest wykorzystywana do identyfikowania tablicy oraz wszystkich jej elementów. W programie reprezentuje adres tablicy, czyli adres jej pierwszego elementu.

`a[j]`

Element tablicy o nazwie `a`, któremu odpowiada indeks `j` (w ogólności `j` może być stałą lub dowolnie skomplikowanym wyrażeniem).

Deklaracja i definicja tablicy

Deklaracja tablicy

Określenie typu danych dla elementów tablicy, nazwy oraz (nie obowiązkowo) liczby elementów w nawiasie kwadratowym, np.

```
int tab[100];
int liczby[];
```

Deklaracja i definicja tablicy

Deklaracja tablicy

Określenie typu danych dla elementów tablicy, nazwy oraz (nie obowiązkowo) liczby elementów w nawiasie kwadratowym, np.

```
int tab[100];
int liczby[];
```

Definicja tablicy

Deklaracja tablicy, w której jest określona liczba elementów.

Deklaracja i definicja tablicy

Deklaracja tablicy

Określenie typu danych dla elementów tablicy, nazwy oraz (nie obowiązkowo) liczby elementów w nawiasie kwadratowym, np.

```
int tab[100];
int liczby[];
```

Definicja tablicy

Deklaracja tablicy, w której jest określona liczba elementów.

Po zdefiniowaniu tablicy każdy jej element może być wykorzystywany jako niezależna zmienna.

Tablica – przykład

```
int tab[10];
...
for (int j = 0; j < 10; j++)
    tab[j] = 0;
```

Tablica – przykład

```
int tab[10];
...
for (int j = 0; j < 10; j++)
    tab[j] = 0;
```

Uwaga!

1. Kompilator C++ nie sprawdza, czy używane w programach wartości indeksów tablic są właściwe.
2. Wartości te również nie są kontrolowane podczas wykonywania programu.
3. Ujemne wartości indeksów są dopuszczalne (oznaczają hipotetyczne elementy o adresach mniejszych od adresu pierwszego elementu).

Przykład zastosowania tablicy

Tablice dobrze sprawdzają się w zastosowaniach, w których mamy do czynienia ze stosunkowo dużą, ustaloną liczbą obiektów tego samego typu.

Przykład zastosowania tablicy

Tablice dobrze sprawdzają się w zastosowaniach, w których mamy do czynienia ze stosunkowo dużą, ustaloną liczbą obiektów tego samego typu.

Problem

W zbiorze $(N + M + 1)$ wylosowanych liczb mamy znaleźć taką, dla której w tym zbiorze jest N liczb nie większych i M nie mniejszych od niej. Dla $N = M$ jest to problem wyznaczania mediany („środkowego elementu") zbioru.

Przykład zastosowania tablicy

Tablice dobrze sprawdzają się w zastosowaniach, w których mamy do czynienia ze stosunkowo dużą, ustaloną liczbą obiektów tego samego typu.

Problem

W zbiorze $(N + M + 1)$ wylosowanych liczb mamy znaleźć taką, dla której w tym zbiorze jest N liczb nie większych i M nie mniejszych od niej. Dla $N = M$ jest to problem wyznaczania mediany („środkowego elementu") zbioru.

Metoda poszukiwania rozwiązania

Ustawimy liczby w danym zbiorze w kolejności rosnącej i wybierzemy tę, która będzie na pozycji $(N + 1)$.

Algorytm sortowania tablicy (przez wybieranie)

1. Zapisz każdą wylosowaną liczbę w innym elemencie tablicy $a[]$ o długości $L = N + M + 1$.
2. Powtarzaj dla indeksów i od 0 do $(L - 2)$:
   1. Wyznacz indeks j taki, że element tablicy o tym indeksie jest najmniejszy spośród elementów o indeksach od i do $(L - 1)$ włącznie.
   2. Zamień miejscami element o indeksie i z elementem o indeksie j.
3. Po zakończeniu powyższych czynności tablica jest posortowana.

Algorytm sortowania tablicy (przez wybieranie)

1. Zapisz każdą wylosowaną liczbę w innym elemencie tablicy $a[]$ o długości $L = N + M + 1$.
2. Powtarzaj dla indeksów i od 0 do $(L - 2)$:
   1. Wyznacz indeks j taki, że element tablicy o tym indeksie jest najmniejszy spośród elementów o indeksach od i do $(L - 1)$ włącznie.
   2. Zamień miejscami element o indeksie i z elementem o indeksie j.
3. Po zakończeniu powyższych czynności tablica jest posortowana.

Wyznaczanie najmniejszego elementu (krok 2.1)

1. Niech $j = i$.
2. Powtarzaj dla indeksów k od $(i + 1)$ do $(L - 1)$:
   - Jeśli $a[k] < a[j]$, to niech $j = k$.
3. j jest poszukiwanym indeksem.

Program sortujący tablicę

```
for (int i = 0; i < L-1; i++) {
    // Poszukiwanie indeksu j
    int j = i;
    for (int k = i + 1; k < L; k++)
        if (a[k] < a[j])
            j = k;

    // Zamiana miejscami a[i] z a[j]
    if (j > i) {
        double m = a[j];
        a[j] = a[i];
        a[i] = m;
    }
}
```

Program sortujący tablicę – c. d.

Wypełnianie tablicy

W powyższym kodzie źródłowym pominięto wypełnianie tablicy:

```
srandom(time(NULL));
for (int i = 0; i < L; i++)
    a[i] = 1.0 / (1.0 + random());
```

Funkcja `random()` generuje liczby pseudolosowe w zakresie od 0 do `RAND_MAX`. Do tablicy wstawiane są liczby pseudolosowe z przedziału $(0, 1]$.

Program sortujący tablicę – c. d.

Wypełnianie tablicy

W powyższym kodzie źródłowym pominięto wypełnianie tablicy:

```
srandom(time(NULL));
for (int i = 0; i < L; i++)
    a[i] = 1.0 / (1.0 + random());
```

Funkcja `random()` generuje liczby pseudolosowe w zakresie od 0 do `RAND_MAX`. Do tablicy wstawiane są liczby pseudolosowe z przedziału $(0, 1]$.

Wynik obliczeń

Po przeprowadzeniu sortowania tablicy wystarczy wydrukować $a[N]$ jako wynik.

Sito Eratostenesa

Poszukujemy liczb pierwszych nie większych od N

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

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

Sito Eratostenesa

Poszukujemy liczb pierwszych nie większych od N

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

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

Zbiór A w powyższym algorytmie można zastąpić tablicą.

Sito Eratostenesa z użyciem tablicy

Poszukujemy liczb pierwszych nie większych od N

Niech $a[]$ będzie tablicą o $(N+1)$ elementach typu `bool`.

1. Niech wszystkie elementy $a[j]$ dla $j > 1$ mają wartość `true`.
2. Niech $k = 2$
3. Wstaw `false` do wszystkich elementów $a[j]$, dla których j jest wielokrotnością k, począwszy od k^2.
4. Powtarzaj:
   - $k = k + 1$
   - Jeśli $k^2 > N$, przejdź do kroku 5.
   - Jeśli $a[k]$ ma wartość `true`, przejdź do kroku 3.
5. Wydrukuj indeksy j, dla których elementy $a[j]$ mają wartość `true`.

Sito Eratostenesa z użyciem tablicy – program

```
bool a[N+1];
int j;

for (j = 2; j <= N; j++)
    a[j] = true;

for (int k = 2; k*k <= N; k++)
    if (a[k]) {
        for (j = k*k; j <= N; j += k)
            a[j] = false;
    }

for (j = 2; j <= N; j++)
    if (a[j])
        cout << j << endl;
```

Sito Eratostenesa z użyciem tablicy – usprawnione

W poprzednim programie nie wykorzystujemy elementów `a[0]` i `a[1]`.

Sito Eratostenesa z użyciem tablicy – usprawnione

W poprzednim programie nie wykorzystujemy elementów `a[0]` i `a[1]`.

Poszukujemy liczb pierwszych nie większych od N

Niech $a[]$ będzie tablicą o $(N-1)$ elementach typu `bool`.

1. Niech wszystkie elementy $a[j]$ mają wartość `true`.
2. Niech $k = 2$
3. Wstaw `false` do wszystkich elementów $a[j]$, dla których $(j+2)$ jest wielokrotnością k, począwszy od k^2.
4. Powtarzaj:
   - $k = k + 1$
   - Jeśli $k^2 > N$, przejdź do kroku 5.
   - Jeśli $a[k-2]$ ma wartość `true`, przejdź do kroku 3.
5. Wydrukuj liczby j, dla których elementy $a[j-2]$ mają wartość `true`.

Sito Eratostenesa z użyciem tablicy – poprawiony program

```
bool a[N-1];
int j;

for (j = 0; j < N-1; j++)
    a[j] = true;

for (int k = 2; k*k <= N; k++)
    if (a[k-2]) {
        for (j = k*k - 2; j < N-1; j += k)
            a[j] = false;
    }

for (j = 0; j < N-1; j++)
    if (a[j])
        cout << (j + 2) << endl;
```

Sito Eratostenesa z użyciem tablicy i wzorów bitowych

Poszukujemy liczb pierwszych nie większych od N

Niech $a[]$ będzie tablicą o $(N+6)/8$ elementach typu `unsigned char`.

1. Niech wszystkie elementy $a[j]$ mają wartość 0 (wszystkie bity są zerami).
2. Niech $k = 2$
3. Wstaw 1 do wszystkich bitów w $a[]$ odpowiadających wielokrotnościom k, począwszy od k^2.
4. Powtarzaj:
   - $k = k + 1$
   - Jeśli $k^2 > N$, przejdź do kroku 5.
   - Jeśli bit w $a[]$ odpowiadający k jest zerem, przejdź do kroku 3.
5. Wydrukuj liczby j, dla których odpowiadające im bity w $a[]$ są zerami.

Sito Eratostenesa z użyciem tablicy i wzorów bitowych c. d.

Obserwacja

Wynik wyrażenia $(N + 7)/8$ jest równy wynikowi dzielenia N przez 8 z zaokrgleniem w górę.

Sito Eratostenesa z użyciem tablicy i wzorów bitowych c. d.

Obserwacja

Wynik wyrażenia $(N + 7)/8$ jest równy wynikowi dzielenia N przez 8 z zaokrgleniem w górę.

Ponieważ pierwszy bit w tablicy ma odpowiadać liczbie 2, a każdy element zawiera 8 bitów, liczba elementów $a[]$ musi być równa wynikowi dzielenia $(N - 1)$ przez 8 z zaokrągleniem w górę.

Sito Eratostenesa z użyciem tablicy i wzorów bitowych c. d.

Obserwacja

Wynik wyrażenia $(N + 7)/8$ jest równy wynikowi dzielenia N przez 8 z zaokrgleniem w górę.

Ponieważ pierwszy bit w tablicy ma odpowiadać liczbie 2, a każdy element zawiera 8 bitów, liczba elementów $a[]$ musi być równa wynikowi dzielenia $(N - 1)$ przez 8 z zaokrągleniem w górę.

Liczbę elementów $a[]$ można obliczyć tak: $(N - 1 + 7)/8 = (N + 6)/8$

Sito Eratostenesa z użyciem tablicy i wzorów bitowych c. d.

Bit w $a[]$ odpowiadający liczbie k

1. Pierwszy bit odpowiada liczbie 2.
2. Każdy element $a[]$ odpowiada 8 bitom.
3. Indeks: $i = (k-2)/8$ (dzielenie z pominięciem reszty).
4. Pozycja bitowa: $b = (k-2)\,\%\,8$.

Sito Eratostenesa z użyciem tablicy i wzorów bitowych c. d.

Bit w $a[]$ odpowiadający liczbie k

1. Pierwszy bit odpowiada liczbie 2.
2. Każdy element $a[]$ odpowiada 8 bitom.
3. Indeks: $i = (k-2)/8$ (dzielenie z pominięciem reszty).
4. Pozycja bitowa: $b = (k-2)\,\%\,8$.

Sprawdzanie wartości bitu

```
(a[(k-2)/8] & (1 << ((k-2) % 8)) == 0
```

Sito Eratostenesa z użyciem tablicy i wzorów bitowych c. d.

Bit w $a[]$ odpowiadający liczbie k

1. Pierwszy bit odpowiada liczbie 2.
2. Każdy element $a[]$ odpowiada 8 bitom.
3. Indeks: $i = (k-2)/8$ (dzielenie z pominięciem reszty).
4. Pozycja bitowa: $b = (k-2)\,\%\,8$.

Sprawdzanie wartości bitu

```
(a[(k-2)/8] & (1 << ((k-2) % 8)) == 0
```

Wstawianie jedynki

```
a[(k-2)/8] |= 1 << ((k-2) % 8)
```

Sito z użyciem tablicy i wzorów bitowych – program

```
unsigned char a[(N+6)/8];
int j;

for (j = 0; j < (N+6)/8; j++)
    a[j] = 0;

for (int k = 2; k*k <= N; k++)
    if ((a[(k-2)/8] & (1 << ((k-2) % 8))) == 0) {
        for (j = k*k; j <= N; j += k)
            a[(j-2)/8] |= 1 << ((j-2) % 8);
    }

for (j = 2; j <= N; j++)
    if ((a[(j-2)/8] & (1 << ((j-2) % 8))) == 0)
        cout << j << endl;
```

Czym są wskaźniki

Wskaźnik (*ang. pointer*)

Zmienna, której wartością jest adres innej zmiennej.

Czym są wskaźniki

Wskaźnik (*ang. pointer*)

Zmienna, której wartością jest adres innej zmiennej.

Deklaracja wskaźnika

Umieszcza się * przy nazwie zmiennej, np.:

```
int *ptr;  // Wartością jest adres zmiennej typu int
char *p;  // Wartością jest adres zmiennej typu char
```

Czym są wskaźniki

Wskaźnik (*ang. pointer*)

Zmienna, której wartością jest adres innej zmiennej.

Deklaracja wskaźnika

Umieszcza się * przy nazwie zmiennej, np.:

```
int *ptr;  // Wartością jest adres zmiennej typu int
char *p;  // Wartością jest adres zmiennej typu char
```

Operator & – obliczanie adresu zmiennej

`n` – zmienna typu `int`, `ptr` – wskaźnik zdefiniowany jak wyżej.

```
ptr = &n;  // Adres n staje się wartością wskaźnika ptr
```

Dostęp do zmiennej poprzez wskaźnik

Gdy wskaźnik `wsk` zawiera adres zmiennej `n`

Wtedy następujące operacje dają ten sam wynik (wstawienie liczby 2 do zmiennej `n`):

- `n = 2;`
- `*wsk = 2;`

Operację oznaczoną symbolem * powyżej nazywa się wyłuskaniem.

Dostęp do zmiennej poprzez wskaźnik

Gdy wskaźnik `wsk` zawiera adres zmiennej `n`

Wtedy następujące operacje dają ten sam wynik (wstawienie liczby 2 do zmiennej `n`):

- `n = 2;`
- `*wsk = 2;`

Operację oznaczoną symbolem * powyżej nazywa się wyłuskaniem.

Wskaźnik można traktować jak „okno” dające dostęp do zmiennej.

Dostęp do elementów tablicy poprzez wskaźnik

Elementy tablicy i wskaźnik tego samego typu

```
ptr = &a[j];  // Adres a[j] staje się wartością wskaźnika ptr
ptr + 1  // Adres a[j+1]
ptr - 1  // Adres a[j-1]
ptr + k  // Adres a[j+k]
```

Dostęp do elementów tablicy poprzez wskaźnik

Elementy tablicy i wskaźnik tego samego typu

```
ptr = &a[j];  // Adres a[j] staje się wartością wskaźnika ptr
ptr + 1  // Adres a[j+1]
ptr - 1  // Adres a[j-1]
ptr + k  // Adres a[j+k]
```

Dla `ptr == &a[j]` równoważne są następujące zapisy

```
a[j+k]   // Wartość elementu a[] o indeksie j+k
*(ptr+k)   // Wartość zmiennej pod adresem ptr przesuniętym o k
ptr[k]   // Wartość zmiennej pod adresem ptr przesuniętym o k
*(a+j+k)   // Wartość zmiennej pod adresem a przesuniętym o j+k
```

Dostęp do elementów tablicy poprzez wskaźnik

Elementy tablicy i wskaźnik tego samego typu

```
ptr = &a[j];  // Adres a[j] staje się wartością wskaźnika ptr
ptr + 1  // Adres a[j+1]
ptr - 1  // Adres a[j-1]
ptr + k  // Adres a[j+k]
```

Dla `ptr == &a[j]` równoważne są następujące zapisy

```
a[j+k]   // Wartość elementu a[] o indeksie j+k
*(ptr+k)   // Wartość zmiennej pod adresem ptr przesuniętym o k
ptr[k]   // Wartość zmiennej pod adresem ptr przesuniętym o k
*(a+j+k)   // Wartość zmiennej pod adresem a przesuniętym o j+k
```

Nazwę tablicy można traktować jako stałą wskaźnikową.

Operacje na wskaźnikach

Zwiększanie i zmniejszanie, += i -=

- Prawa strona musi być całkowitego typu.
- Oznacza przesunięcie adresu we wskaźniku o pewną liczbę pozycji.
- Dla wskaźnika typu `int`

```
ptr += k;  // Adres w ptr jest zwiększany o k * sizeof(int)
```

Operacje na wskaźnikach

Zwiększanie i zmniejszanie, += i -=

- Prawa strona musi być całkowitego typu.
- Oznacza przesunięcie adresu we wskaźniku o pewną liczbę pozycji.
- Dla wskaźnika typu `int`

```
ptr += k;  // Adres w ptr jest zwiększany o k * sizeof(int)
```

Inkrementacja i dekrementacja, ++ i --

- Przesunięcie „do przodu" lub „do tyłu" o jedną zmienną danego typu (np. element tablicy).
- Można łączyć z innymi wyrażeniami (jak dla „zwykłych" zmiennych).
- Dla wskaźnika typu `int`

```
ptr++;  // Adres w ptr jest zwiększany o sizeof(int)
```

Odejmowanie wskaźników

- Można odjąć jeden wskaźnik od drugiego (tego samego typu).
- Ma to sens, gdy wskaźniki zawierają adresy elementów tej samej tablicy.
- Wynikiem jest różnica między adresami wyrażona jako przesunięcie mierzone liczbą pozycji w tablicy (tzn. różnica indeksów odpowiadających elementom tablicy, których adresy zawierają te wskaźniki).

Odejmowanie wskaźników

- Można odjąć jeden wskaźnik od drugiego (tego samego typu).
- Ma to sens, gdy wskaźniki zawierają adresy elementów tej samej tablicy.
- Wynikiem jest różnica między adresami wyrażona jako przesunięcie mierzone liczbą pozycji w tablicy (tzn. różnica indeksów odpowiadających elementom tablicy, których adresy zawierają te wskaźniki).

Zmienna jest wskazywana przez wskaźnik

Gdy adres tej zmiennej jest wartością danego wskaźnika.

Problem z przekazywaniem tablic do funkcji

W C++ tablica nie może być argumentem funkcji

1. Argumenty funkcji to zmienne:
   - Których wartości początkowe pochodzą z innych części programu (przekazywanie przez wartość).
   - Które zostały zdefiniowane w innych częściach programu (przekazywanie przez referencję).
2. Tablice nie są zmiennymi!
   - Tablica jest zespołem zmiennych.
   - Nazwa tablicy jest stałą wskaźnikową.

Dostęp do tablic (z funkcji) poprzez wskaźniki

Wskaźniki mogą być argumentami funkcji

1. Wskaźnikowe argumenty funkcji można w treści tej funkcji traktować tak, jakby były nazwami tablic.
2. Wartość (początkowa) parametru wskaźnikowego oznacza lokację (w pamięci), którą funkcja ma traktować jako początek tablicy.
3. Funkcja musi „poznać” liczbę elementów tablicy niezależnie od adresu jej początku.
   - Informacja o liczbie elementów nie jest automatycznie przekazywana wraz z adresem początku tablicy.
   - Musi ona być przekazana do funkcji w inny sposób (np. jako dodatkowy argument funkcji).

Dostęp do tablic (z funkcji) poprzez wskaźniki

Wskaźniki mogą być argumentami funkcji

1. Wskaźnikowe argumenty funkcji można w treści tej funkcji traktować tak, jakby były nazwami tablic.
2. Wartość (początkowa) parametru wskaźnikowego oznacza lokację (w pamięci), którą funkcja ma traktować jako początek tablicy.
3. Funkcja musi „poznać" liczbę elementów tablicy niezależnie od adresu jej początku.
   - Informacja o liczbie elementów nie jest automatycznie przekazywana wraz z adresem początku tablicy.
   - Musi ona być przekazana do funkcji w inny sposób (np. jako dodatkowy argument funkcji).

```
void sort(double *a, int n);
```

Funkcja sortująca tablicę

```
void sort(double *a, int n)
{
    // Sortowanie tablicy.
    for (int i = 0; i < n-1; i++) {
        // Poszukiwanie indeksu j.
        int j = i;
        for (int k = i + 1; k < n; k++)
            if (a[k] < a[j])
                j = k;

        // Zamiana miejscami a[i] z a[j].
        if (j > i) {
            double m = a[j];
            a[j] = a[i];
            a[i] = m;
        }
    }
}
```

Funkcja sortująca tablicę

```
void sort(double *a, int n)
{
    // Sortowanie tablicy.
    for (int i = 0; i < n-1; i++) {
        // Poszukiwanie indeksu j.
        int j = i;
        for (int k = i + 1; k < n; k++)
            if (a[k] < a[j])
                j = k;

        // Zamiana miejscami a[i] z a[j].
        if (j > i) {
            double m = a[j];
            a[j] = a[i];
            a[i] = m;
        }
    }
}
```

Wskaźniki można definiować jako tablice o nieznanej długości:

```
void sort(double a[], int n);
```

Napisy i tablice

Literały tekstowe (napisy) w czasie kompilacji są zamieniane na tablice o elementach typu `char`. Podczas uruchamiania programu zwykle są one umieszczane w obszarze pamięci tylko do odczytu.

Napisy i tablice

Literały tekstowe (napisy) w czasie kompilacji są zamieniane na tablice o elementach typu `char`. Podczas uruchamiania programu zwykle są one umieszczane w obszarze pamięci tylko do odczytu.

1. Napis w kodzie źródłowym reprezentuje adres, pod którym znajduje się ciąg znaków w postaci tablicy.
2. Dla zbioru znaków ASCII każdy element tej tablicy reprezentuje 1 znak.
   - Znaki odpowiadają liczbom całkowitym, zgodnie ze zbiorem znaków wykorzystywanym przez kompilator.
   - W kodzie źródłowym mogą być zapisywane bezpośrednio (np. `'a'`) lub jako kody (liczby całkowite).
3. Ostatnim elementem tej tablicy zawsze jest znak o kodzie 0.

Napisy i wskaźniki

Napisy mogą być odczytywane z pomocą wskaźników:

```
char *s = "Ala ma kota";  // s zawiera adres początku napisu

cout << s << endl;  // Drukowanie napisu

// Drukowanie napisu
for (int i = 0; i < 11; i++)
    cout << s[i];  // Drukuj znak pod adresem s + i
cout << endl;

// Drukowanie napisu
do
    cout << *s;  // Drukuj znak pod adresem s
while (*s++);
cout << endl;
```

Tablice o elementach typu `char` i ciągi znaków

Każdą tablicę o elementach typu `char` można wykorzystać do drukowania ciągu znaków.

Tablice o elementach typu char i ciągi znaków

Każdą tablicę o elementach typu char można wykorzystać do drukowania ciągu znaków.

W tym celu adres posząątku tablicy przekazuje się do cout, jakby był on adresem napisu:

```
char tab[6];

tab[5] = '\0';  // Znak o kodzie 0.
tab[0] = 'A';
for (int i = 1; i < 5; i++)
    tab[i] = tab[i-1] + 1;
cout << tab << endl;
```

Drukowanie kończy się w momencie napotkania znaku '\0'.

Argumenty programu

Argumenty funkcji `main()`

`argc` – liczba argumentów programu + 1.

`*argv[]` – tablica wskaźników zawierających adresy argumentów programu.

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

Argumenty programu

Argumenty funkcji `main()`

`argc` – liczba argumentów programu + 1.

`*argv[]` – tablica wskaźników zawierających adresy argumentów programu.

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

1. Pierwszy (o indeksie 0) element `argv[]` to adres nazwy programu (użytej do uruchomienia go).
2. Element `argv[]` o indeksie `argc` ma wartość `NULL` (czyli 0).
3. Pozostałe elementy `argv[]` wskazują na ciągi znaków wprowadzone w linii poleceń (po nazwie programu).

Uruchamianie programu z argumentami

1. Argumenty programu rozdziela się spacjami (tzn. spacja stanowi separator dla argumentów programu).
2. Jeżeli argumentem programu ma być ciąg znaków zawierający spację, trzeba użyć cudzysłowu lub znaku \ (*ang. backslash*).

Uruchamianie programu z argumentami

1. Argumenty programu rozdziela się spacjami (tzn. spacja stanowi separator dla argumentów programu).
2. Jeżeli argumentem programu ma być ciąg znaków zawierający spację, trzeba użyć cudzysłowu lub znaku \ (*ang. backslash*).

Przykład

```
./program raz dwa trzy cztery
```

Uruchamianie programu z argumentami

1. Argumenty programu rozdziela się spacjami (tzn. spacja stanowi separator dla argumentów programu).
2. Jeżeli argumentem programu ma być ciąg znaków zawierający spację, trzeba użyć cudzysłowu lub znaku \ (*ang. backslash*).

Przykład

```
./program raz dwa trzy cztery
```

Drukowanie argumentów programu

```
for (int j = 1; j < argc; j++)
    cout << argv[j] << endl;
```

Wykorzystanie pamięci przez program

Zasady wykorzystania pamięci

1. Każdy program jest wykonywany jako jedno z wielu zadań (*ang. task*).
2. Zadania nie mogą przeszkadzać sobie nawzajem.
3. Każde zadanie dysponuje „własną” pamięcią.
   - Zawartość pamięci wykorzystywanej przez zadanie (na ogół) nie może być modyfikowana przez inne zadania.
   - Jądro systemu operacyjnego przydziela pamięć zadaniom według potrzeb.

Wykorzystanie pamięci przez program

Zasady wykorzystania pamięci

1. Każdy program jest wykonywany jako jedno z wielu zadań (*ang. task*).
2. Zadania nie mogą przeszkadzać sobie nawzajem.
3. Każde zadanie dysponuje „własną” pamięcią.
   - Zawartość pamięci wykorzystywanej przez zadanie (na ogół) nie może być modyfikowana przez inne zadania.
   - Jądro systemu operacyjnego przydziela pamięć zadaniom według potrzeb.

Pamięć statyczna

1. Kod (*ang. code*) – rozkazy dla procesora.
2. Dane tylko do odczytu (np. literały tekstowe).
3. Zmienne globalne i statyczne (wartość „przeżywa” zakończenie wykonywania funkcji).

Pamięć dynamiczna

Zmienne lokalne

1. Tworzone przed rozpoczęciem wykonywania bloku (np. treści funkcji) lub w trakcie wykonywania go.
2. Usuwane po zakończeniu wykonywania bloku.
3. Znajdują się w obszarze pamięci zwanym stosem (*ang. stack*), w którym pamięć jest przydzielana automatycznie (przez jądro systemu operacyjnego).

Pamięć dynamiczna

Zmienne lokalne

1. Tworzone przed rozpoczęciem wykonywania bloku (np. treści funkcji) lub w trakcie wykonywania go.
2. Usuwane po zakończeniu wykonywania bloku.
3. Znajdują się w obszarze pamięci zwanym stosem (*ang. stack*), w którym pamięć jest przydzielana automatycznie (przez jądro systemu operacyjnego).

Stos jest wykorzystywany także do innych celów (argumenty funkcji, adresy powrotne dla funkcji) i może mieć ograniczone rozmiary.

Pamięć dynamiczna

Zmienne lokalne

1. Tworzone przed rozpoczęciem wykonywania bloku (np. treści funkcji) lub w trakcie wykonywania go.
2. Usuwane po zakończeniu wykonywania bloku.
3. Znajdują się w obszarze pamięci zwanym stosem (*ang. stack*), w którym pamięć jest przydzielana automatycznie (przez jądro systemu operacyjnego).

Stos jest wykorzystywany także do innych celów (argumenty funkcji, adresy powrotne dla funkcji) i może mieć ograniczone rozmiary.

Pamięć rezerwowana na żądanie

Przydzielana programowi po zgłoszeniu przez niego jawnego zapotrzebowania na określoną ilość pamięci.

Rezerwowanie pamięci na żądanie

Rezerwowanie pamięci z pomocą `new`

- Rezerwowanie pamięci na zmienną:

```
ptr = new double;
```

- Rezerwowanie pamięci na tablicę:

```
ptr = new double[N];
```

Rezerwowanie pamięci na żądanie

Rezerwowanie pamięci z pomocą `new`

- Rezerwowanie pamięci na zmienną:

```
ptr = new double;
```

- Rezerwowanie pamięci na tablicę:

```
ptr = new double[N];
```

Zasady

1. Typ danych wskaźnika `ptr` musi odpowiadać typowi danych rezerwowanej zmiennej lub tablicy (`void *` „pasuje” do każdego typu danych).
2. W przypadku niepowodzenia wskaźnik `ptr` będzie mieć wartość `NULL` (czyli 0).

Zwalnianie pamięci przydzielonej na żądanie

Pamięć przydzielona na żądanie pozostaje do dyspozycji programu, aż zostanie przez niego jawnie zwolniona (nie ma znaczenia to, w którym miejscu programu została ona zarezerwowana).

Zwalnianie pamięci przydzielonej na żądanie

Pamięć przydzielona na żądanie pozostaje do dyspozycji programu, aż zostanie przez niego jawnie zwolniona (nie ma znaczenia to, w którym miejscu programu została ona zarezerwowana).

Zwalnianie pamięci z pomocą `delete`

- Zwalnianie pamięci zarezerwowanej na zmienną:

```
delete ptr;
```

- Zwalnianie pamięci zarezerwowanej na tablicę:

```
delete [] ptr;
```

Wskaźnik `ptr` musi zawierać adres obszaru pamięci przydzielonego na żądanie.

Korzystanie z pamięci przydzielonej na żądanie

Pamięć rezerwowana na żądanie i wskaźniki

1. Zmienne rezerwowane z pomocą `new` są dostępne tylko przez wskaźniki (jest to główne zastosowanie wskaźników w C++).
   - Trzeba przechowywać ich adresy, żeby można było zwolnić pamięć zajmowaną przez nie.
2. Tablice rezerwowane z pomocą `new` mają takie same własności, jak „zwykłe” tablice, ale nie mają nazw (są dostępne tylko za pośrednictwem wskaźników).

Korzystanie z pamięci przydzielonej na żądanie

Pamięć rezerwowana na żądanie i wskaźniki

1. Zmienne rezerwowane z pomocą `new` są dostępne tylko przez wskaźniki (jest to główne zastosowanie wskaźników w C++).
   - Trzeba przechowywać ich adresy, żeby można było zwolnić pamięć zajmowaną przez nie.
2. Tablice rezerwowane z pomocą `new` mają takie same własności, jak „zwykłe” tablice, ale nie mają nazw (są dostępne tylko za pośrednictwem wskaźników).

Zbyt częste rezerwowania i zwalnianie pamięci może powodować problemy

- Po zwolnieniu pamięć może być przydzielona innym zadaniom.
- Następnym razem możemy dostać pamięć z innego obszaru.
- Może to prowadzić do rozdrobnienia pamięci (*ang. memory fragmentation*).

Zmienne w pamięci statycznej

Zmienne globalne

1. Definiowane na zewnątrz definicji funkcji.
2. Tworzone przy uruchamianiu programu.
3. Ich zasięg (*ang. scope*) obejmuje cały program (deklaracje muszą być „widoczne" dla funkcji, które mają z nich korzystać).

Zmienne w pamięci statycznej

Zmienne globalne

1. Definiowane na zewnątrz definicji funkcji.
2. Tworzone przy uruchamianiu programu.
3. Ich zasięg (*ang. scope*) obejmuje cały program (deklaracje muszą być „widoczne" dla funkcji, które mają z nich korzystać).

Zmienne statyczne

1. Definiowane ze słowem kluczowym `static`.
2. Tworzone przy uruchamianiu programu.
3. Ich zasięg zależy od miejsca, w którym są zdefiniowane.
   - Zdefiniowane na zewnątrz funkcji są „widoczne" od pierwszej deklaracji do końca pliku zawierającego ich definicje.
   - Zdefiniowane wewnątrz bloku są „widoczne" tylko w tym bloku, ale ich wartości „przeżywają" zakończenie wykonywania go.

Dyrektywy preprocesora

Kod źródłowy programu w C++ jest wstępnie przetwarzany przed właściwą kompilacją przez program zwany preprocesorem (*ang. preprocessor*).

Dyrektywy preprocesora

Kod źródłowy programu w C++ jest wstępnie przetwarzany przed właściwą kompilacją przez program zwany preprocesorem (*ang. preprocessor*).

Dyrektywy preprocesora

Preprocesor wykonuje tzw. dyrektywy (*ang. directive*) określające pewne przekształcenia kodu źródłowego. Dyrektywy preprocesora zaczynają się znakiem '#'.

Dyrektywy preprocesora

Kod źródłowy programu w C++ jest wstępnie przetwarzany przed właściwą kompilacją przez program zwany preprocesorem (*ang. preprocessor*).

Dyrektywy preprocesora

Preprocesor wykonuje tzw. dyrektywy (*ang. directive*) określające pewne przekształcenia kodu źródłowego. Dyrektywy preprocesora zaczynają się znakiem '#'.

Dyrektywa `#include`

`#include` nakazuje włączenie do przetwarzanego pliku zawartości innego pliku o podanej ścieżce.

- Ścieżka w cudzysłowie – względem bieżącego katalogu.
- Ścieżka w nawiasie $< >$ jest interpretowana w specjalny sposób.

Makrodefinicje

Dyrektywa `#define`

`#define` pozwala na zdefiniowanie dowolnego symbolu, który jest „rozwijany” przed właściwą kompilacją, np.:

```
#define M_PI 3.14159265358979323846
```

Makrodefinicje

Dyrektywa `#define`

`#define` pozwala na zdefiniowanie dowolnego symbolu, który jest „rozwijany” przed właściwą kompilacją, np.:

```
#define M_PI 3.14159265358979323846
```

Makrodefinicja (*ang. macrodefinition, macro*)

Definicja symbolu z pomocą `#define`.

Makrodefinicje

Dyrektywa `#define`

`#define` pozwala na zdefiniowanie dowolnego symbolu, który jest „rozwijany” przed właściwą kompilacją, np.:

```
#define M_PI 3.14159265358979323846
```

Makrodefinicja (*ang. macrodefinition, macro*)

Definicja symbolu z pomocą `#define`.

Makrodefinicja musi mieścić się w 1 wierszu (można go „przedłużyć” używając znaków \ do „sklejania” wierszy).

Makrodefinicje z parametrami, kompilacja warunkowa

Makrodefinicja może zależeć od pewnej liczby parametrów, które są zastępowane odpowiednimi ciągami znaków przez preprocesor, np.:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Kompilacja warunkowa

Uzależnienie sposobu kompilacji programu od spełnienia (lub nie) określonych warunków. Do tego celu służą dyrektywy `#if`, `#ifdef`, `#ifndef`, `#else` oraz `#endif`, np.:

```
#ifdef CONFIG_KEXEC
    case LINUX_REBOOT_CMD_KEXEC:
        ret = kernel_kexec();
        break;
#endif
```