

Programowanie Obiektowe i C++

Marcin Benke

Instytut Informatyki UW

16.10.2006

Klasy pamięci

- Domyślną klasą pamięci jest `auto`; kompilator zadecyduje gdzie umieścić zmienną (w pamięci, na stosie w rejestrze...).
- Możemy zasugerować umieszczenie zmiennej w rejestrze specyfikatorem `register`:

```
register int i;
```
- `extern` oznacza, że zmienna ma przydzieloną pamięć “gdzie indziej” (zwykle w innym module)

Modyfikator `static`

- `static` w odniesieniu do zmiennej lokalnej nakazuje jej alokację statyczną; jej wartość będzie pamiętana pomiędzy wywołaniami funkcji.
- W odniesieniu do funkcji lub zmiennej globalnej, `static` oznacza “ta nazwa ma nie być widoczna poza tym plikiem”

```
void f() {
    static int i = 0 ;
    cout << ++i << endl;
}

int main() {
    for(x=0;x<10;x++)
        f(); // wydrukuje 1..10
}
```

Inkrementacja i dekrementacja

- Operator ++ oznacza zwiększenie zmiennej o 1, zaś -- zmniejszenie o 1.
- Wartością wyrażenia jest wartość zmiennej, ale “przed”, czy “po”?
- Jeśli operator został umieszczony przed zmienną, to najpierw wykona się operacja, czyli dostaniemy wartość “po”
- Jeśli operator został umieszczony za zmienną, to dostaniemy wartość “przed” (a operacja “potem”)

Stałe

- Stałą możemy zadeklarować używając modyfikatora `const`, np:

```
const int i = 10 ;
```
- Stałe obowiązują te same reguły zasięgu co zmienne (tj. możemy mieć stałe lokalne).
- Możemy pobrać adres stałej, ale nie możemy zmodyfikować komórki pamięci o tym adresie.
- Zwykle zadba o to kompilator, ale jeśli stosujemy “brudne sztuczki”, dowiemy się o tym dopiero w czasie wykonania...
- Specyfikatora `const` możemy też użyć do argumentu funkcji, np:

```
int f(const int i)
```

Modyfikator `volatile`

- `volatile` jest poniekąd przeciwieństwem `const`; informuje kompilator “nigdy nie wiadomo kiedy ta wartość się zmieni”
- Używamy go wobec wartości poza kontrolą programu, np. rejestrów sprzętowych.
- Ma on również zastosowanie w programach wielowątkowych.

Specyfikatory rozmiaru i znaku

- Do typów `int` oraz `double` można dodawać specyfikator rozmiaru `short` lub `long`. Oznacza to sugestię użycia reprezentacji o mniejszym lub większym od standardowego zakresie (ew. precyzji).
- Dokładne znaczenie jest zależne od implementacji.
- Do typów całkowitych można dodawać specyfikatory znaku `unsigned` lub `signed` oznaczającej użycie liczby bez znaku lub ze znakiem.
- `int` jest domyślnie ze znakiem
- `char`, `signed char` i `unsigned char` to trzy różne typy.

Domyślne wartości parametrów

- Deklarując argument funkcji można dopisać znak = i wyrażenie, np.

```
int f(int a, int b = 10)
```

- Deklaruje się w ten sposób jego domyślną wartość.
- Funkcję można wywoływać z tym argumentem lub bez niego, np

```
f(1, 2)
```

```
f(3)
```

- Jako brakujący argument przejęta będzie wartość wyrażenia z deklaracji.
- Można mieć kilka domyślnych argumentów, ale wszystkie muszą być na końcu.

Domyślne wartości parametrów

- Wyrażenie domyślne jest wyliczane przy każdym wywołaniu z brakującym argumentem — uwaga na efekty uboczne, np.

```
int f(int a, int b = c++) ;  
int g(int a, int b = h()) ;
```

- Wiązanie nazw odbywa się w miejscu deklaracji, zaś wartościowanie w miejscu wywołania:

```
int a = 1; int f(int);  
int g(int x = f(a));  
void h() {  
    a = 2;  
    { int a = 3 ;  
      g() ; // NB: g(f(2))  
    }  
}
```

Przeciążanie funkcji

- Możemy mieć funkcje o tej samej nazwie, acz różniące się ilością lub typami argumentów.
- Właściwa funkcja zostanie wybrana według argumentów w danym wywołaniu.
- Przykład:

```
int div(int, int);  
double div(double, double);  
div(1, 2)  
div(1.0, 2.0)
```

Przeciążanie funkcji

- Uwaga na mieszanie przeciążania i argumentów domyślnych

```
int f(int a) {  
    return 1 ;  
}
```

```
int f(int a, int b = 2) {  
    return b ;  
}  
// f(0) ... ?!
```

Jeszcze o operatorach

Poznaliśmy operatory arytmetyczne, warto wspomnieć jeszcze

- operatory logiczne,
- operatory bitowe,
- operator sekwencjonowania,
- operator warunkowy,
- operatory rzutowania,
- konstrukcję “operator-przypisanie”.

Operatory logiczne

- && — koniunkcja
- || — alternatywa
- ! — negacja

Operatory bitowe

- & — koniunkcja bitowa — $1 \& 2 = 0$
- | — alternatywa bitowa — $1 | 2 = 3$
- ^ — różnica symetryczna — $1 \wedge 3 = 2$
- ! — negacja bitowa
- << — przesunięcie w lewo — $1 \ll 7 = 128$
- >> — przesunięcie w prawo — $7 \gg 1 = 3$

Sekwencjonowanie wyrażeń, czyli operator Przecinek

Składnia:

wyrażenie1 , *wyrażenie2*

Semantyka: wyrażenia wyliczane są od lewej do prawej,
wartością wyrażenia jest wartość *wyrażenia2*.

Przykład:

```
for (i=f() , j=i+1; i<7; i++, j++) ...
```

Operator warunkowy

Składnia:

(wyrażenie1 ? wyrażenie2 : wyrażenie3)

Semantyka: Jeśli *wyrażenie1* jest prawdziwe, wartością wyrażenia jest *wyrażenie2*, wpp. *wyrażenie3*.

Przykład:

```
i = (i%2 ? 3*i + 1 : i/2) ;
```


Rzutowanie

Składnia:

(typ) wyrażenie albo *typ(wyrażenie)*

Semantyka: *wyrażenie* jest konwertowane do podanego typu.

Przykład:

```
int b = 200 ;  
unsigned long a = (unsigned long int)b;  
float c = float(200);
```

Niektóre rzutowania (zwłaszcza numeryczne) są dokonywane automatycznie, np.

```
float x = 1 + 2.5 ;
```

Konstrukcja operator-przypisanie

Często spotykamy instrukcje typu

```
x = x * 8 ;
```

W C++ (podobnie jak w C) możemy ją skrótowo zapisać

```
x *= 8 ;
```

Takie skróty są dostępne dla prawie wszystkich operatorów binarnych, możemy więc również napisać

```
x <<= 3 ;
```

Tablice i arytmetyka wskaźników

- Nazwa tablicy (bez [...]) oznacza adres jej początku
- Konstrukcja *wskaźnik* + *k* oznacza “*k* elementów od adresu *wskaźnik*. (dokładny adres zależy od typu *wskaźnika*).
- `p[k]` to to samo co `* (p+k)`
- `wskaźnik1 - wskaźnik2` — ile kroków pomiędzy.

```
int a[10];
int* ip = a;
int i ;
for(i=0;i<10;i++) ip[i] = i*10 ;
for(ip;ip<a+10;ip++) *ip = (ip - a) ;
```

Napisy

- Napisy to tablice znaków.
- Koniec napisu jest oznaczany znakiem o kodzie 0.
- Stałe napisowe ("...") to stałe tablice.
- Cudzysłów w napisie:
"Powiedział: \"żegnaj\" i odszedł"
- Stałe znakowe — pojedynczy znak w apostrofach, np.
'a' ''' '\0' '\n'

Przykład — obliczanie długości napisu

```
unsigned strlen(char *s)
{
    int i ;
    for(i=0;s[i];i++) ;

    return i ;
}
```

(NB funkcja `strlen` jest standardowo dostępna po dołączeniu nagłówka `<cstring>`, nie trzeba jej pisać na nowo).

Obliczanie długości napisu 2

```
unsigned strlen2(char *s)
{
    char *p;
    for(p=s; *p; p++) ;

    return p-s ;
}
```

Zmienne lokalne dla pętli

Zamiast

```
int i; for(i=0;...
```

możemy często napisać

```
for(int i=0;...
```

Uwaga: zasięg `i` ogranicza się do danej pętli, czyli np.

```
for(int i=0; s[i]; i++) ;  
return i ;
```

jest błędne.

W dawnych wersjach C/C++ reguły zasięgu były inne i nie było tego problemu.

Argumenty funkcji main

Funkcja `main` w rzeczywistości ma argumenty i może dawać wynik:

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

- `argv` to tablica argumentów programu
- `argc` to rozmiar tej tablicy
- `argv[0]` to ścieżka dostępu i nazwa programu
- `envp` to tablica zmiennych środowiskowych, zakończona 0.
- wynik to tzw. kod powrotu: 0 = sukces, co innego = kod błędu.

Struktury

```
struct Point {  
    int x ;  
    int y ;  
} ; // Musi kończyć się średnikiem  
  
Point p ; // p.x, p.y
```

Wskaźniki i struktury

```
struct Point {  
    int x ;  
    int y ;  
} ;
```

```
Point *p ;  
p->x // Skrót dla (*p).x
```

Dynamiczna alokacja pamięci — new i delete

```
int* pf;  
pf = new int ; // alokacja pamięci  
delete pf ; // zwolnienie  
pf = new int[10] // alokacja tablicy
```

Wyliczenia

```
enum ShapeType {  
    circle,  
    square,  
    rectangle  
}; // Musi kończyć się średnikiem
```