Język C++, elementy nieobiektowe

⇒ język C wewnątrz C++

⇒ nowe nieobiektwowe elementy C++

⇒ wersje C i C++ ...

wersje języka C:

C89 - „ANSI C” (C++ jest oparty na C89 !!!)

C99

wersje języka C++:

twórca języka: Bjarne Stroustrup

ARM C++ - „The Annotated C++ Reference Manual”

kompilator bc3.1

C++ ISO/IEC14882:1998

C++ ISO/IEC14882:2003 - standard obecnie obowiązujący

http://www.open-std.org/jtc1/sc22/wg21/

C++0x - nieformalna nazwa ...

Język C++, typy

Typy proste i złożone ...

Typy proste:

- typy rzeczywiste:
  - float, double, long double
- typy całkowite:
  - char, int, long int = long, short int = short,
  - signed/unsigned „typ całkowity” - liczby ze znakiem/bez znaku; np. unsigned int
  - (int = signed int, char = signed/unsigned char (opcja kompilatora!))

Typy złożone:

- tablice
  - unsigned int tab1[10];
- struktury/ unie/ klasy
  - struct AAA { int konto1, konto2; char* nazwa; } a1;
- wskazania (?)
  - long liczba, *wsk = &liczba;
- referencje (??)
  - int i, &j1=i, &j2=i; // j1 i j2 to referencje do zmiennej „i”

Język C++, stałe/literały, inicjowanie

Typy proste:

typy rzeczywiste:

```
float f = 123.456e3; //123.456*10 ^3; stała typu double!
float f = 123.456e3F; //123.456*10 ^3; stała typu float
```

typy całkowite:

```
long x; x=123; /*przypisujemy stałą typu int*/ x=321L; /*stała typu long*/
  // gdzie ma zastosowanie „L” ??? odp: tam gdzie sizeof(int)==2
x=0x12AB; /*szesnastkowo*/ x=0234; /*ósemkowo*/
unsigned long int y =123UL; /* inicjujemy stałą typu unsigned long*/
char c; c=’a’; c=’\n’; c=’\xFF’; c=’33’; /*- ósemkowo*/ c=123;
```

Typy złożone:

tablice

```
unsigned int tab1[10] = {1, 2, 3, 4, 5, 6, 7}; // reszta zera ...
   // INICJOWANIE: typowi prostemu odpowiada pojedyncze wyrażenie,
      // tablicy/strukturze odpowiada lista wyrażeń w nawiasach „{ }”
char c1[10] = ”qwerty”;
char c2[10] ={‘q’, ‘w’, ‘e’, ‘r’, ‘t’, ‘y’, ‘\0’};
      // obie deklaracje równoważne !!! (tu jest uproszczenie inicjatora)
```

Język C++, inicjowanie c.d.

Typy złożone:

struktury

struct AAA { int konto1, konto2; char* nazwa; } a1 = {111, 222, ”Kowalski”};

wskazania

char *w=”123”; w=”ABC”; // ”ABC” to wsk. na pierwszy znak napisu

inicjowanie tablic wielowymiarowych:

zasady upraszczające:

1. można usuwać wewnętrzne nawiasy „{}”
2. można opuścić wyrażania końcowe w „{}” - wstawiane są ZERA!!!
3. uproszczenie ”ABC” ⇔ {‘A’, ‘B’, ‘C’, ‘\0’}
4. domniemanie pierwszego rozmiaru tablicy

int tab1[2][3] ={{1, 2, 0}, {3, 4, 0}};
int tab2[2][3] ={1, 2, 0, 3, 4, 0};
int tab3[2][3] ={{1, 2}, {3, 4}};
int tab4[][3] ={{1, 2}, {3, 4}};

Język C++, deklaracje

Deklarowanie złożonych struktur danych metodą „cebulkową” ...

Deklarujemy tablicę „tab” 7-elementową ...

```
tab[7]
```

której elementami są wskazania ...

```
*tab[7]
```

na 3-elementowe tablice ...

```
(*tab[7])[3]
    // musze użyć () bo „[]” ma wyższy priorytet niż „*” !!!
```

których elementami są wskazania na char

```
char *(*tab[7])[3]
    // to jest nasza deklaracja ...
```

Język C++, deklaracje

Interpretowanie złożonych struktur danych metodą „cebulkową” ...

Mamy deklarację zmiennej tab:

```
    char *(*tab[7])[3]
        // identyfikator tab jest otoczony przez „*” i „[]”
        // ale „[]” ma wyższy priorytet niż „*” zatem:
```

tab jest tablicą 7-elementową ...

```
    char *(*X)[3]
```

której elementami są wskazania ... (bo X ma „*”)

```
    char *X[3]
```

na 3-elementowe tablice ...

```
    char *X
```

wskazań na znaki.

Język C++, deklaracje

⇒ deklaracja zmiennej wygląda w języku C tak jak ta zmienna będzie wyglądać w wyrażeniu!

char tab[10]; c=tab[i];

⇒ formalnie, deklarator to:

```
deklarator:
   identyfikator
   (deklarator)
   deklarator [wyrażenie_stałe]
   * deklarator
   deklarator (def_parametrów_funkcji)
```

a deklaracja to:

```
deklaracja:
   oznaczenie_klasy_i_typu lista_deklaratorów
oznaczenie_klasy:
   auto| static| extern| register| typedef
oznaczenie_typu:
   char| int| ...| opis_struct| identyfikator_typu
```

Język C++, deklaracje

klasy pamięci:

⇒ auto - zmienne lokalne funkcji; trzymane na stosie; klasa domniemana

⇒ static - (dla zmiennych) prywatna zmienna globalna funkcji

⇒ extern - (dla zmiennych) zmienna zdefiniowana gdzie indziej

⇒ register

⇒ typedef - służy do definiowania typów;

UWAGA: pozwala uniknąć złożonych deklaracji

```
// zamiast:
char *(*tab[7])[3];
// można tak:
typedef char *t_tab3[3]; t_tab3 *tab[7];
```

inne przykłady deklaracji:

int (*(*wsk)[10])(int);

// wsk jest wsk na 10-elem tablice wsk na fun(int) zwracające int

Język C++, deklaracje

Deklaracja/definicja struktury:

```
struct AAA {
  int konto1, konto2; char* nazwa;
} a1 = {111, 222, ”Kowalski”};
  // zdefiniowaliśmy strukturę AAA oraz zmienna a1
struct AAA a2;
AAA a3;
   // deklaracje dodatkowych zmiennych a2 i a3
   // w C++ można opuścić słowo struct (C99?)
```

Klasy i unie definiuje się podobnie (zamiast **struct**, **class**, **union**);
różnica struct/class - domyślna widoczność pól w OOP
(struct - public, class - private)

Słownik: deklaracja - to „nie pełna” definicja;
definicja - coś, co w pełni definiuje zmienną/funkcję itp.
Przykłady: nagłówek/prototyp funkcji to deklaracja funkcji;
nagłówek z ciałem to definicja funkcji.

Język C++, wyrażenia

⇒ wyrażenia występują w instrukcjach:

```
fun();  // instrukcja-wyrażenie składająca się z wywołania funkcji
if( x+1>y ) { /* kod */} // instrukcja warunkowa
```

⇒ wyrażenie i l-wyrażenie; wyrażenie ma wartość, l-wyrażenie ma wartość i *referencję* (dlatego może wystąpić po lewej stronie operatora przypisania)

⇒ wyrażenia w C są zbudowane z identyfikatorów, literałów i operatorów

UWAGA: wywołanie funkcji, odczyt pola struktury i elementu tablicy robi się w języku C przy pomocy operatorów!!!

⇒ Sposobem interpretacji wyrażenia sterują priorytety i wiązania operatorów

Przykłady:

```
a  - b + c * d
```

jest interpretowane jako:

```
a  - b + (c * d)     // bo „*” ma wyższy priorytet od „+” i „-”
(a  - b) + (c * d)    // bo „+” i „-” wiążą od lewej do prawej
```

jeśli operator X wiąże od lewej do prawej to wyrażenie „a X b X c X” jest interpretowane jako: (((a X b) X c) X d)

Język C++, operatory

Operatory w kolejności malejących priorytetów; ostatnia kolumna: łączność/wiązanie

```
16   ()  []  ->  ::  .                left to right
15   !  ~  +  -  ++  --  &  *  (typ)  sizeof
     new  delete                       right to left
14   .*  ->*                           left to right
13   *  /  %                           left to right
12   +  -                              left to right
11   <<  >>                            left to right
10   <  <=  >  >=                      left to right
09   ==  !=                            left to right
08   &                                 left to right
07   ^                                 left to right
06   |                                 left to right
05   &&                                left to right
04   ||                                left to right
03   ?:                                right to left
02   =  *=  /=  %=  +=  -=  &=  ^=  |=
     <<=  >>=                          right to left
01   ,                                 left to right
```

Język C++, operatory nowe!

// operatory rzutowania (lepsze niż rzutowanie z języka C!)

const_cast	adds or removes the const or volatile modifier from a type
dynamic_cast	converts a pointer to a desired type
reinterpret_cast	replaces casts for conversions that are unsafe or implementation dependent.
static_cast	converts a pointer to a desired type

// RTTI:

typeid	gets run-time identification of types and expressions

Język C++, wyrażenia, operatory

Operatory z priorytetem 16 (tzw wyrażenia pierwotne):

fun(parametry) // wywołanie funkcji

tab[index] // dostęp do elementu tablicy

struktura.pole // dostęp do pola struktury

wskStrukt->pole // dostęp do pola struktury wskazywanej

Operatory z priorytetem 15:

(typ) wyrażenie // jawna konwersja (=rzutowanie);

// **UWAGA**: typ określa się jak w deklaracji tylko bez identyfikatora

```
long lo=50; // chce odczytac 4 bajty tego long-a
for (int i=0; i<4; i++)
  printf("%d\n",(int)((char*)&lo)[i]);
```

sizeof(typ), sizeof(wyrażenie)

rozmiar w bajtach typu lub wyrażenia

*** wyrażenie_wskazujace**

„wyłuskanie” danej na która wskazuje wyrażenie

UWAGA: tab[i] ⇔ *(tab+i) // to jest tzw arytmetyka na adresach!

& l-wyrażenie // pobranie adresu

Język C++, wyrażenia, operatory

Operatory inkrementacji/dekrementacji ++, --

występują w wersji pre i post-fiksowej

```
x++  // wartością tego wyrażenia jest wartość x
     // efektem „ubocznym” jest zwiększenie wartości x o 1
++x  // wartością tego wyrażenia jest wartość x+1
     // efektem „ubocznym” jest zwiększenie wartości x o 1
```

Operatory przypisania = += *= ...

```
x = 123  // operator przypisania
x+=123 // jest równoważne x=x+123,
                //ale wyrażenie „x” jest opracowywane tylko raz!
```

UWAGA: x musi być l-wyrażeniem !!!

Operatory warunkowy ?:

```
a ? b : c  // jeśli „a” to zwróć „b”, w przeciwnym razie zwróć „c”
```

Przykład:

```
#define max(a,b) (((a)>(b))?(a):(b))
  // tzw makrodefinicja preprocesora; działa ale wyrażenia są 2 razy obliczane!
a ? b : c ? d : e ⇔ a ? b : (c ? d : e) // bo wiązanie od prawej
```

Język C++, wyrażenia, operatory

Operatory logiczne i bitowe && oraz &

&& - „and”, & - „and” na bitach

Pytanie: jaka jest różnica między:

```
int fun1(); /*zwraca 0*/ int fun2(); /*zwraca 1*/
fun1() && fun2();
fun1() & fun2();
```

Nietypowe l-wyrażenia:

```
int* fun() { static i=123; printf(”fun”); return &i; }
// ...
*fun()+= 1; // ile razy się wywoła fun()? Tu na pewno raz!
*fun()= *fun() + 1; // zależy od kompilatora!!
*fun(),*fun(); // tu na pewno dwa razy!!
      // (to jest operator „,”)
```

Wniosek: efekty uboczne w wyrażeniach są NIEBEZPIECZNE !**!!**

Język C++, wyrażenia

Tablice i funkcje w wyrażeniach:

tablica ⇔ wskazanie na pierwszy element tablicy

wniosek: tab[i] = *(tab+i) // tab+i to tzw „arytmetyka wskazań”

funkcja ⇔ wskazanie na funkcję

Przykłady:

```
// eksperyment z tablicami i wsk:
char *c1=”ABCD”, tab[10], *c2=tab;
    // tab  jest tablicą i równocześnie wskazaniem na pierwszy elem tablicy
    // dlatego można napisać *c2=tab;
while( *c2++ = *c1++);  // przepisuje napis do tab
    // warunek w while równocześnie wykonuje całą robotę!
    // tu się używa faktu że string C kończy się ‘\0’!
    // na co wskazują c1 i c2 po zakończeniu działania tego kodu???
int i=0; while( tab[i] = c1[i]) i++; // wersja bez wsk
    // while( tab[i++] = c1[i]); // ZŁE rozwiązanie, dlaczego???

// eksperyment ze wsk na funkcje:
int fun1(int i); // zwraca i+1; fun zdefiniowane gdzie trzeba
int fun2(int i); // zwraca i+2;
int (*wfun[2])(int); wfun[0]=fun1; wfun[1]=fun2;
int i1=(*wfun[0])(100), i2=wfun[1](100); // poprawne wywołania fun !!!
    // jakie będą wartości i1 i i2 ??? czyżby OOP w C ???
```

Język C++, arytmetyka wsk, kiedy można przypisać

Arytmetyka wsk, przykład:

```
// eksperyment z arytmetyką wsk:
long tab[2] ={111, 222}, *w1=tab, *w2;
w2=w1+1;
printf("w1=%p, *w1=%ld, ", w1, *w1);
printf("w2=%p, *w2=%ld, ", w2, *w2);
   // w1=0064FDFC, *w1=111, w2=0064FE00, *w2=222,
```

Jeśli „T *w"; to „w++" oznacza zwiększenie wartości „w" o „sizeof(T)".
Można też porównywać wsk i odejmować ...

Zgodność „w sensie przypisania":

można wykonać Op1=Op2, jeśli Op1 jest modyfikowalna l-wartością oraz

1. Op1 i Op2 należą do typu arytmetycznego
2. Op1 i Op2 są strukturą (zgodną)
3. Op1 i Op2 są zgodnymi wskazaniami
4. Op1 jest (void*) a Op2 jest dowolnym wskazaniem
5. Op1 jest wskazaniem a Op2 jest stałą 0 (NULL)

Język C++, instrukcje

Instrukcje C++ właściwie identyczne jak w języku C

Przykłady:

```
// instrukcja-wyrażenie: sposób na uruchamianie procedur
fun()+1; // wartość wyrażenia jest gubiona

// instrukcja grupująca
{ /*lista instrukcji rozdzielonych „;” */ }

// instrukcja warunkowa
if( x<y ) {  // to jeden z możliwych styli zapisu kodu ...
   printf(”min x”); min=x;
} else {
   printf(”min y”); min=y; // kiedy można opuścić „{}” ?
}

// instrukcja „while”
int i=0; while( i++ < 10 ) { printf(”.”); }
   // ile kropek wypisze?
while(1); while(true); // pętle nieskończone
```

Język C++, instrukcje

```
// instrukcja „while” c.d.
int i=10;
while( 1 ) {
   if(i<0) break; // wyskakiwanie z pętli …
            // druga możliwość to „continue”
   i--;
}

// instrukcja „for” c.d.
for(int i=0; i<10; i++) { /*jakiś kod*/ }
for(;;) { /*petla nieskonczona*/ }
for(q=0,w=0; q+w<100; q++,w++) {
   // przykład zastosowania operatora „,”
}
```

Pozostałe instrukcje:

do/while, switch, break/continue, return, goto

Język C++, referencje

Przykład:

```
int i=123; int &j1=i; int &j2=i;
   // modyfikacja j1 pociąga modyfikację j2 oraz i
   // deklaracja zmiennej ref MUSI być połączona z inicjalizacją
```

Zmienne „j1” i „j2” to ref do zmiennej „i”.

Czym jest ref ? Odp: podobna do wsk, ale z „syntaktycznym uproszczeniem”
(nie wymaga „*” i „&” w wyrażeniach)
ref NIE jest „normalnym” typem (nie może wystąpić wszędzie tam gdzie wsk)
raczej nie należy zakładać że ref to ukryty wsk!!!

L-wartość to właśnie referencja!!! (termin „l-wartość” zapożyczony z C)

Typy wyrażeń uwzględniające pojęcie referencji:

```
123 // (int)
i+1 // gdzie „i” jest int-em; (int)
i // gdzie „i” jest int-em; (int&)
```

Ref i „const”:

```
int i=123; int &j1=i; int const &j2=i;
   // poprzez j1 mogę modyfikować a przez j2 NIE!!!
```

Język C++, funkcje

Przykład funkcji w języku C++ ...

Trzy sposoby przekazywania parametrów:

```
int fun(int p1, int* p2w, int &p3r) {
      // p3r -tutaj inicjalizacja niewymagana!
   printf("%d %d %d", p1, *p2w, p3r);
   *p2w=222; p3r=333;
   return 111;
}
// …
int i1,i2=22,i3=33; i1=fun(i1,&i2,i3);
   // fun modyfikuje zmienne i2 oraz i3
```

Najwygodniejsze wydaje się przekazywanie przez ref!!!

Ryzyko związane z ref:

```
void fun(int& i) { i=123; }
int i=321; fun(i);
float f=321.0F; fun(f); // kompilator mówi ok. ale ...
```

Język C++, funkcje

Przekazywanie parametrów do funkcji przez ref i const ref ...
(główne zastosowanie const ?)

```
void fun1(int& i) {
   i++;
}
void fun2(int const& i) {
   i++; // błąd zgłaszany przez kompilator
   ((int&)i)++; // ok
      // bez jawnej konswersji sie NIE skompiluje!!!
}
int main()
{
  int x=100;
  fun1(x); printf("x=%d\n",x);
  fun2(x); printf("x=%d\n",x);
}
```

Język C++, inne zastosowania const

```
int main()
{
   char c1, c2;
   char * const x = c1; // MUSI być zainicjalizowane
                // inaczej błąd kompilatora
   char const * y;
   y= &c2;
   *y='A'; // błąd zgłaszany przez kompilator
}
```

x jest stałym wsk na char

y jest wsk na stały znak

Język C++, funkcje

Zwracanie wartości funkcji przez ref ...

```
int& fun() {
   static int i=123; return i;
}
int main()
{
   fun()=100;
   printf("%d\n",fun());
   fun()++;
   printf("%d\n",fun());
}
```

Język C++, funkcje

Przekazywanie struktur danych przez parametr - tablice

```
int tab[11][22][33];
void fun(int _tab[11][22][33]) { // „11” można opuścić
  printf("sizeof(_tab)=%d\n", sizeof(_tab));
}
int main()
{
  fun(tab); printf("sizeof(tab)=%d\n", sizeof(tab));
}
/* wydruk:
sizeof(_tab)=4
sizeof(tab)=31944
*/
```

Wniosek: tablice przekazuje się przez wsk na pierwszy element

Język C++, funkcje

Przekazywanie struktur danych przez parametr - struktury

```
struct QQQ {int i,j; } qqq;
void fun(QQQ _qqq) {
  printf("sizeof(_qqq)=%d\n", sizeof(_qqq));
}
int main()
{
  fun(qqq); printf("sizeof(qqq)=%d\n", sizeof(qqq));
}
/* wydruk:
sizeof(_qqq)=8
sizeof(qqq)=8
*/
```

Wniosek: struktury są przekazywane w całości

Język C++, przeciążanie funkcji

```
void fun(int i) { printf("int, i=%d\n", i); }
void fun(float f) { printf("float, f=%f\n", f); }
int main()
{  fun(1);
   fun(1.1F);
   //fun(1.1); // błąd kompilatora (Ambiguity...)
         // 1.1 jest typu double!!
         // JEST możliwość konwersji z „double” do „int”
         // jak i do „float”
   fun('A');
}
/* wydruk:
int, i=1
float, f=1.100000
int, i=65
*/
```

Język C++, elementy nieobiektowe, co opuściliśmy ...

- $\Rightarrow$ funkcje inline
- $\Rightarrow$ domyślne wartości parametrów funkcji
- $\Rightarrow$ konwersje standardowe i jawne (jakie są ograniczenia)