Struktury danych

Źle – patrz referencje

Elementy składowe

Operator .
wybór składowej obiektu

```
string info(film_t film)
{
 stringstream ss;
 ss << "\"" << film.tytul << "\""
    << " rok:" << film.rok
    << " czas trwania:" << film.czas
    << " min. cena:" << film.cena << endl;
 return ss.str();
}

int main()
{
 film_t starWarsIV { "Star Wars: Episode IV – A New Hope", 1977, 121};
 cout << info(starWarsIV);
 starWarsIV.cena = 120;
 cout << info(starWarsIV);
 return 0;
}
```

```
typedef int czas_t;
using cena_t = int;

struct film_t
{
  string tytul;
  int rok;
  czas_t czas;
  cena_t cena;
};
```

nazwy w konwencji C

Struktura grupuje dane (różnych typów) pod wspólną nazwą we wspólnym obszarze pamięci

Uwaga! rozmieszczenie pól w pamięci zależy od ustawień kompilatora!

Pola bitowe

```
int main() {
  poleBitowe pB{};
  int *pInt = (int*)&pB;
  P(sizeof(poleBitowe));
  P(sizeof(int)); PB;
  pB.day = 31;    PB;
  pB.day = 33;    PB;
  *pInt = 1;      PB;
  *pInt = -1;     PB;
  pB.day = 31;    PB;
  pB.day = 33;    PB;
}
```

```
struct poleBitowe {
  unsigned int lsb   : 1;
  unsigned int day   : 5;
  unsigned int other : 25;
  unsigned int msb   : 1;
} ;
#define P(x) std::cout << #x << " = " << x << " "
#define PB std::cout << std::endl ; P(pB.lsb);
P(pB.day); P(pB.other); P(pB.msb) ; P(*pInt)
```

```
sizeof(poleBitowe) = 4 sizeof(int) = 4
pB.lsb = 0 pB.day = 0  pB.other = 0 pB.msb = 0 *pInt = 0
pB.lsb = 0 pB.day = 31 pB.other = 0 pB.msb = 0 *pInt = 62
pB.lsb = 0 pB.day = 1  pB.other = 0 pB.msb = 0 *pInt = 2
pB.lsb = 1 pB.day = 0  pB.other = 0 pB.msb = 0 *pInt = 1
pB.lsb = 1 pB.day = 31 pB.other = 33554431 pB.msb = 1 *pInt = -1
pB.lsb = 1 pB.day = 31 pB.other = 33554431 pB.msb = 1 *pInt = -1
pB.lsb = 1 pB.day = 1  pB.other = 33554431 pB.msb = 1 *pInt = -61
```

Politechnika Warszawska

Funkcja main()

Obowiązkowym typem wyniku jest **int**
Gwarantowane poprawne formy nagłówka:

```
int main()
{
//Program
//...
return 0; // poprawne zakończenie
}
```

```
int main(int ac, char* av[])
{
  //Program
  // ...
  return 0; // poprawne zakończenie
}
```

Na końcu można pominąć

int nie **void**

tablica argumentów:
av[0] - nazwa prog.
....
av[ac-1] - ostatni arg.
av[ac] == 0

Politechnika Warszawska

Funkcja `main()` (cd1)

- Dostępna zwykle postać:

```
int main(int ac, char* av[], char* ev[]){ /* ...*/ }
```

```
#include <iostream>
using namespace std;
int main(int ac, char *av[], char **ev)
{ int i;
  for(i = 0; ev[i]; ++i)
    if(ev[i][0] == 'H') // Filtr
      cout<< i<< ": "<< ev[i]<< endl;
  cout<<"Liczba zmiennych: "<<i<<endl;
}
```

zmienne środowiska;
koniec gdy: **ev[?] == 0**

```
Np.:
8: HOMEDRIVE=C:
9: HOMEPATH=\Documents and Settings\kgr
Liczba zmiennych: 40
```


Funkcja `main()` (cd2)

Ograniczenia

- **jedyna** w programie (bez przeciążania)
- bez kwalifikatorów **`inline`**, **`static`**
- nie można jej **wywoływać** (aktywowana przez środowisko); tym samym – bez rekursji
- nie można **pobierać jej adresu** (program taki jest niezgodny ze standardem – ułomny (ang. *ill-formed*); kompilatory nie muszą protestować).

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Przeciążanie funkcji i operatorów
Parametry funkcji
Funkcje otwarte (rozwijane)
Funkcje constexpr
Referencje
- *w parametrach*
- *w deklaracjach*

Referencje / wskaźniki

Przeciążanie funkcji, deklaracje `extern`

C: unikalne nazwy funkcji w danym zasięgu
1. globalne – w całym programie
2. statyczne – w zasięgu pliku źródłowego

C++: można używać tej samej nazwy dla zbioru funkcji z różnymi listami parametrów formalnych – przeciążanie (nazwy) funkcji

Funkcje różniące się tylko typem wartości zwracanej nie mogą być w tym samym zbiorze

Przeciążanie zawsze obowiązuje w ramach ustalonego zasięgu nazw

Przeciążanie funkcji (ang. function overloading) pozwala na definiowanie funkcji o tych samych nazwach, które różnią się liczbą lub typami parametrów

Przeciążanie funkcji, deklaracje `extern` cd.

- **Przykład: zbiór `max2` (bez szablonów)**

```
int   max2(int, int);
float max2(float, float);
float max2(int, float);
float max2(float, int);
float max2(float, int);// OK: ale powtórzenie
float max2(const float, int);
float max2(const float, const int);
char* max2(const char*, const char*);
const char* max2(char*, const char*);
const char* max2(const char*, char*);
char* max2(char*, char*);
const char* max2(char*, char*); // BŁĄD: wynik
char* max2(char[], char[]);     // OK: powtórzenie
```

Lepiej unikać

Przeciążanie funkcji, deklaracje `extern` cd.

Sygnatury (prototypów) funkcji

1. Konsolidacja programu wymaga unikalności „nazw" funkcji
2. Kompilator przekształca nazwę funkcji na unikalny łańcuch znaków (sygnaturę) z uwzględnieniem liczby i typów parametrów
3. Dla programisty generacja sygnatur jest niewidoczna
   (może być ignorowana z wyjątkiem programów mieszanych
   - wielojęzykowych)

Przeciążanie funkcji, deklaracje **extern** cd. 11

```
#include <cstdio>
extern"C" void exit(int);// Funkcja z <cstdlib>
extern"C" void abort();  // Funkcja z <cstdlib>

using namespace std;
int main(int ac, char *av[])
{ if(ac == 0) abort();  // Niemożliwe
  if(ac == 1)
  { fprintf(stderr, "Wywołanie\n%s plik_we\n", av[0]);
    exit(1);
  }
  // ...
  return 0;
}
```

```
...unresolved external symbol "void exit(int)" (?exit@@YAXH@Z)
...unresolved external symbol "void abort(void)" (?abort@@YAXXZ)
...fatal error LNK1120: 2 unresolved externals
```

Przeciążanie funkcji, deklaracje **extern** cd. 12

Przyczyna błędów

funkcje **exit(int), abort();** zostały poddane sygnaturowaniu wg C++, choć są z biblioteki ANSI C.

Zalecenia

- Używając funkcji bibliotecznych zawsze korzystać z plików nagłówkowych, np.

  #include <cstdlib>

- Jeśli funkcja nie pochodzi z biblioteki a spełnia konwencje języka C, to trzeba użyć deklaracji:

  extern "C" ***prototyp_funkcji*****;**

- Można stosować deklaracje zgrupowane:

  extern "C"{*lista_prototypów_funkcji*;} np.

  extern "C"{ void exit(int); void abort();}

Politechnika Warszawska

Przeciążanie operatorów

Operator: funkcja w specjalnej notacji

```
a + b  <==> operator+(a, b)
  *p   <==> operator*(p)
```

...

Przeciążanie operatorów: jak przeciążanie funkcji

Większość operatorów C++ jest już przeciążona dla typów wbudowanych, np.

1. "+" zastosowany do argumentów **`int`**
2. "+" zastosowany do argumentów **`double`**
3. "+" zastosowany do argumentów wskazanie i stała_całkowita

Przeciążanie operatorów (cd1)

Ograniczenia

1. Nie można definiować nowych operatorów; przeciążanie operatorów nie narusza struktury leksykalnej.
2. Interpretacja operatorów dla typów wbudowanych pozostaje bez zmian.
3. (Inna forma powyższego) W każdym przeciążeniu operatora musi wystąpić przynajmniej jeden argument typu nie wbudowanego (klasa lub wyliczenie).
4. Operatory przeciążone zachowują standardowy priorytet i łączność operatorów wbudowanych.
5. Przeciążać można tylko operatory jedno- lub dwu-argumentowe (z nielicznymi wyjątkami); jedyny operator 3-argumentowy **`a?b:c`** nie może być przeciążany.

Przeciążanie operatorów (cd2)

Uwagi

1. Dla każdego typu są gwarantowane predefiniowane operatory: '=' (przypisanie), '&' (wzięcie wskazania), ',' (operator sekwencji); zachowanie tych operatorów dla klasy lub typu wyliczeniowego można zmienić.
2. Niezmienniki obowiązujące dla typów wbudowanych (np. `--x;` ⇔ `x-=1;`) nie muszą być zachowane w przeciążeniach.
3. Operatory dla typów wbudowanych wymagające lub zwracające l-wartości nie muszą tego wymuszać w przeciążeniach (lepiej jednak zachować zgodność). Na przykład wyrażenie: `++++x;` może być błędne w wersji przeciążonej.

Przeciążanie operatorów (cd3)

Przykład

```
//typedef enum {E,N,W,S} Dir;
enum Dir {E,N,W,S};
Dir operator+(Dir d, int i)
// Zmiana kierunku o i*90 (lewoskrętnie)
{ int k = (i + d)%4;
  return static_cast<Dir>(k);
  // return (Dir)k; - konwencja C
  // return Dir(k);
  // return (Dir)(k);
}
```

Rzutowanie (zalecane)

Inne formy konwersji

Uwaga! Co się stanie dla:

```
int k = (d + i)%4;
```

Antyprzykład

Przeciążanie operatorów (cd4)

Przykład

silnie typowane (*scoped enum*)

Wymagane rzutowanie

```
enum class Dir { E, N, W, S };
Dir operator+(Dir d, int i)
// Zmiana kierunku o i*90 (lewoskrętnie)
{
  int k = (i + static_cast<int>(d)) % 4;
  return static_cast<Dir>(k);
  // return (Dir)k; - konwencja C
  // return Dir(k);
  // return (Dir)(k);
}
```


Mechanizmy pomocnicze

Przeciążanie funkcji i operatorów
Parametry funkcji
Funkcje otwarte (rozwijane)
Funkcje constexpr
Referencje
- *w parametrach*
- *w deklaracjach*

Referencje / wskaźniki

Prototypy i parametry funkcji

Zasada w C++: deklaruj / definiuj, używaj

- Funkcja może być wywołana **tylko po** wcześniejszym zadeklarowniu lub zdefiniowaniu.
- Deklaracja (prototyp) funkcji, specyfikuje typ wyniku, nazwę, typy parametrów i (nieobowiązkowo) nazwy parametrów. Konieczne ze względu na mechanizm przeciążania.

W C domniemanie `int fun();`

Przekazywanie parametrów

- przez wartość
- przez referencję

```
void swap(int a, int b) // Wartość
{ int temp = a;
  a = b; b = temp;
} // coś nie tak

void swap(int *a, int *b) // Wartość!
{ int temp = *a;
  *a = *b; *b = temp;
}

void swap(int &a, int &b) // Referencje/Odniesienia
{ int temp = a;
  a = b; b = temp;
}
```

W języku C argumenty przekazywane są tylko przez **wartość!**

Prototypy i parametry funkcji (cd1)


```
int f(void);/* W C i C++ to samo */
int g();
// W C++ znaczy int g(void);
/* W C funkcja z dowolną listą
   argumentów przekazywanych bez
   kontroli
*/
int f(void)/* Definicja */
{ h(); // C++: Błąd - brak deklaracji
       /* C:   domniemana funkcja int h(); */
  return 0;
}
```


Parametry funkcji

4 rodzaje parametryzacji funkcji:

1. Funkcje z ustaloną listą parametrów obowiązkowych
2. Funkcje z podlistą parametrów obowiązkowych i oznacznikiem parametrów nieobowiązkowych (dodatkowych)
3. Funkcje z podlistą parametrów obowiązkowych i podlistą parametrów domniemanych (predefiniowanych)
4. Funkcje z podlistą parametrów obowiązkowych, podlistą parametrów domniemanych i oznacznikiem parametrów nieobowiązkowych (dodatkowych)

Podlisty parametrów mogą być puste

Parametry funkcji (cd1)

Schemat ogólny prototypu

`TypWyniku nazwaFunkcji(po, pp, pd);`	
TypWyniku	Typ wartości zwracanej przez funkcję
po	parametry obowiązkowe
pp	parametry predefiniowane (domniemane)
pd	parametry dodatkowe, oznacznik '...'

Listy typów parametrów w deklaracji i definicji muszą być identyczne; typy Typ* i Typ[] są tożsame.

Parametry funkcji (cd2)

Rozłączność mechanizmów

- Deklarator / nagłówek funkcji może zawierać wszystkie 3 rodzaje parametrów (obowiązkowe, domyślne, dodatkowe)
- Jeżeli działa mechanizm parametrów dodatkowych, to były podane wszystkie argumenty odpowiadające parametrom domniemanym (i jeszcze jakieś dodatkowe).

Pojęcia obowiązkowy, domyślny należą do interpretacji wywołującego

- Z punktu widzenia funkcji parametry domyślne są także obowiązkowe: są obecne w każdej aktywacji, choć ich wartość nie musi być podana w miejscu wywołania lecz pochodzi z definicji wartości domniemanej.

Parametry funkcji (cd3)


```
int f(int, char*,char[]); //3 obowiązkowe
int g(char*,...); // 1 obowiązkowy i dodatkowe
int h(int, int=0, int=0);
// 1 parametr obowiązkowy, 2 domniemane;
// wywołania z 1,2 lub 3 argumentami
int q(int a, int b=1, ...);
// obowiazkowy, domniemany i dodatkowe
void doWszystkiego_doNiczego(...);
// dowolne argumenty bez kontroli
int r(int, int=0, int=0, int);
// Błąd: parametr obowiązkowy na końcu
int s(int, ... , int=0);
// Błąd: parametrów domyślnych
```


Parametry dodatkowe

- Argumenty dodatkowe (oznacznik '. . .') są przekazywane bez kontroli typów
- Programista musi zapewnić rozpoznawanie liczby i typów argumentów dodatkowych
- Przekazywanie informacji o argumentach dodatkowych:
  - poprzez argumenty obowiązkowe, np.

```
int printf(const char *format, ...);
```

  - poprzez konwencje rozpoznawane wewnątrz samej listy
  - poprzez zmienne globalne (nie stosować).

Parametry dodatkowe: <cstdarg>


```
#include <iostream>
#include <cstdio>
#include <cstdarg>
#define PRINT(x) std::cout <<(#x) << " = " << (x) << std::endl
// Makro do prezentacji wyników
int sum(int n, ...) // n: liczba arg. dodatkowych
{ int s=0;
  va_list ap;
  va_start(ap, n); // ap na pierwszy arg. z ...
  while(n-- >0)
    s += va_arg(ap, int);
  va_end(ap);
  return s;
}

int main()
{ PRINT(sum(5.0, 1,2,3,4,5));
  PRINT(sum(5,   1.0,2.0,3.0,4.0,5.0));
  system("pause");
}
```

Makra obsługi parametrów dodatkowych

```
sum(5.0, 1,2,3,4,5) = 15
sum(5, 1.0,2.0,3.0,4.0,5.0) =
2146435072
Press any key to continue . . .
```

`double` zamiast `int`

Parametry domyślne

TypW nf(lista_po, typ_1 pd_1 = v_1, /*itd*/, typ_n pd_n = v_n);

Funkcja zależna od ***n*** parametrów domniemanych może być wywoływana na **n+1**sposobów (pomijanie *0, 1, ..., n* argumentów od strony prawej). Pominięte argumenty są zastępowane wartościami predefiniowanymi $\mathbf{v_i}$.

```
double simpson(double f(double),
               double a=0.0, double b=1.0);

x = simpson(exp,1, 10);
// Całka funkcji exp w przedziale [1.0, 10.0]
y = simpson(sin,-1.0);
// równoważne: y=simpson(sin,-1.0, 1.0);
z = simpson(sqrt);
// równoważne: z=simpson(sqrt,0.0, 1.0);
```

Parametry domyślne => przeciążanie

Deklaracja n parametrów domyślnych => zbiór n+1 funkcji przeciążonych

```
double simpson(double f(double),
               double a=0.0, double b=1.0);
```

```
double simpson(double (*fp)(double),
       double a, double b);

double simpson(double f(double),double a)
{ return simpson(f, a, 1.0); }

double simpson(double f(double))
{ return simpson(f, 0.0, 1.0); }
```

generowane automatycznie przez kompilator

Parametry domyślne - wymagania (1)

- Wartość domniemana może być specyfikowana tylko raz i dotyczyć parametru, który po prawej stronie ma parametry domniemane albo jest ostatni w liście
- Kolejne prototypy tej samej funkcji obserwowane przez kompilator mogą co najwyżej uzupełniać od prawej do lewej listę parametrów domniemanych (powtórzenia prototypów funkcji są dopuszczalne)

```
int f(int x, int y);
int f(int x, int = 1);
int f(int a = 0, int); // OK
                       // niezalecane
//int f(int = 0, int = 1); // LEPIEJ TAK
int f(int a, int b); // Powtórzenie
```

```
int f(int x, int y);
int f(int a = 0, int); // Błąd!
int f(int x, int = 1);
```

Parametry domyślne – wymagania (2)

- Nazwy parametrów w deklaracjach funkcji są opcjonalne; w powtórzeniach można użyć innych nazw (lepiej nie!)
- W definicji funkcji można pominąć nazwę parametru, jeśli funkcja z tego parametru nie korzysta; sytuacja taka może powstać:
  - przejściowo, podczas modyfikacji lub uruchamiania programu
  - gdy parametr jest "martwy" i służy tylko do przeciążenia funkcji (np. przy przeciążaniu operatora "++" w wersji przyrostkowej).

```
Dir operator++(Dir& d, int ) // Operator przyrostkowy
{                            // Zmiana kierunku o +90
  Dir temp = d;
  d = d+1;         // Przeciążony operator+
  return temp;
} // Można pisać: Dir d = E; ... d++; ...
```


Parametry domyślne – wymagania (3)

Wyrażenia dla wartości domniemanych nie mogą zawierać innych parametrów ani zmiennych lokalnych

```
void fun(int x, int y=x)   // Błąd
{ int i;
  extern void g(int z = i);// Błąd
  // ...
}
```

Deklaracja funkcji nie może prowadzić do niejednoznaczności przy rozstrzyganiu wyboru funkcji ze zbioru przeciążonego

```
int h(int),h(int a,int b=0);// h(1)=>Konflikt
```

Parametry domyślne – wymagania (4)

Obliczanie wartości domyślnej następuje:

- w punkcie wywołania funkcji
- według związania nazw obowiązującego w punkcie deklaracji

```
int size=32; // Zmienna globalna
int h(int);
void f(int a = h(size));

// .....
int g()
{ size = 16;// zmienna globalna
  { int size = 64;// zmienna lokalna
    f();f(a = h(?))
    // .....
  }
}
```

Wiązanie w p. deklaracji

f(h(16))

Parametry domyślne i dodatkowe - przykład


```
#include <cstdio>
#include <cstdarg>
int f(int g(int)= 0, int n=0, ...)
{ if(g==0) return 0;
  va_list ap;
  int s=0;
  va_start(ap, n);
  while(n--) s+= g(va_arg(ap, int));
  va_end(ap);
  return s;
}
int by2(int n) { return n*2; }
int pos(int n) { return n; }

#define P(x) std::cout <<(#x) << " = " << (x) << std::endl
int main()
{ P(f());
  P(f(pos,3,1,2,3));
  P(f(by2,3,1,2,3));
}
```

```
f()=0
f(pos,3,1,2,3)=6
f(by2,3,1,2,3)=12
```

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Funkcje otwarte

Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Przeciążanie funkcji i operatorów
Parametry funkcji
Funkcje otwarte (rozwijane)
Funkcje constexpr
Referencje
- *w parametrach*
- *w deklaracjach*

Referencje / wskaźniki

Politechnika Warszawska

3

Funkcje otwarte (rozwijane)


```
inline TypW nazwaFun(parametry);
```

- **Specyfikator `inline`: traktuj każde wywołanie funkcji jak preferowane polecenie substytucji jej treści**
- Potencjalna korzyść - unika się protokołu wołania funkcji:
  - przygotuj rekord aktywacji na stosie
  - umieść argumenty wywołania w rekordzie aktywacji
  - zapamiętaj stan (niektórych) rejestrów i adres powrotu
  - wywołaj funkcję [.....]
  - pobierz wynik z rekordu aktywacji
  - wycofaj rekord aktywacji
- Kompilator może spełnić to żądanie – zależy to od strategii generacji kodu, zakresu optymalizacji itd. (podobnie jak w przypadku kwalifikatora `register` dla zmiennych lokalnych i parametrów).

```
//int fun(int,int)
// (prolog)
push ebp
mov ebp, esp
//int r;
sub esp , 4
//r = a+b;
mov eax, [ebp+8]
add eax, [ebp+12]
mov [ebp-4], eax
//return r;
mov eax, [ebp -4]
// (epilog)
mov esp,  ebp
pop ebp
ret
```


Funkcje rozwijane (cd1)

Funkcje rozwijane zachowują semantykę zwykłych funkcji (kontrola / konwersja typów argumentów) choć posiadają pewne cechy makro (substytucji treści)

```
#include <iostream>
inline int max(int a, int b) { return a>b?a:b; }

#define P(x) std::cout<<#x " = "<<(x)<<std::endl
inline int fun(int x, int y=0, ...)
{ if(y==1) return x+y+*(&y +1); // nieprzenośne
  else     return max(x,y)+x;
}
int main()
{ P(fun(10));       // fun(10) = 20
  P(fun(2.71, 3) ); // fun(2.71, 3) = 5
  P(fun(2,1,5));    // fun(2,1,5) = 8
  return 0;
}
```

Funkcje rozwijane (cd2)

- Podobieństwo do mechanizmu makrorozwinięcia jest tylko powierzchowne; przetwarzanie makro poprzedza kompilację - kompilator widzi wynik makrorozwinięcia
- Obliczanie (ew. konwersje) argumentów funkcji otwartych wynikają z semantyki parametrów
- Konwersje w makrorozwinięciach – z semantyki wyrażeń

```
#define MAX(a,b) ((a)>(b))?(a):(b)
// Typy parametrów a,b nieokreślone
inline int max(int a, int b) { return a > b ? a : b; }
// ...
z = max(A[i] * (B[j] + x), y);// Wyrażenie obliczane raz
z = MAX(A[i] * (B[j] + x), y);
// z = ((A[i]*(B[j]+x))>(y))?(A[i]*(B[j]+x)):(y);
p = MAX(f(i), g(j));// Ile wywołań f(.)?
q = MAX(++i, j++);       // Ile inkrementacji i?
```


Funkcje rozwijane – wymagania, zalecenia

- **Funkcja może być definiowana w każdej jednostce kompilacji!**
- Musi być zdefiniowana identycznie w każdej jednostce kompilacji korzystającej z niej (najlepiej w pliku nagłówkowym)
- Zalecenie: specyfikatora `inline` używać tylko przed definicją funkcji, a nie w deklaracji; dotyczy to również funkcji składowych klas
- Nie ma możliwości (składniowej) żądania rozwinięcia funkcji `inline` w jednym miejscu, a wywołania zwykłego w innym miejscu
- Funkcje `inline` bezpośrednio lub pośrednio rekurencyjne, zawierające skomplikowany kod lub struktury danych, nie będą rozwijane; kompilator nie musi generować ostrzeżeń w takich przypadkach
- Kompilatory optymalizujące same mogą podejmować decyzję o rozwijaniu pewnych (prostych) funkcji

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Funkcje constexpr

Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Przeciążanie funkcji i operatorów
Parametry funkcji
Funkcje otwarte (rozwijane)
Funkcje constexpr
Referencje
- *w parametrach*
- *w deklaracjach*

Referencje / wskaźniki

Politechnika Warszawska

4

Funkcje `constexpr`

Funkcje gwarantują, że **wartość zwracana** przez funkcję (przy odpowiednich argumentach) **będzie wyrażeniem stałym podczas kompilacji programu**

- funkcja musi zwracać wartość

W zależności od wersji standardu ma pewne ograniczenia

- deklarowanie zmiennych lokalnych, pojedyncza instrukcja *return*, itp.

```
constexpr int factorial_C11(int n) {
    return n <= 1 ? 1 : (n * factorial_C11(n - 1));
}
constexpr int factorial_C14(int n) {
    int res = 1;
    while (n > 1)
        res *= n--;
    return res;
}
int main() { // obliczane w trakcie kompilacji programu
   std::cout << factorial_C11(5) << '\n';
   std::cout << factorial_C14(5) << '\n';
}
```


Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Przeciążanie funkcji i operatorów

Funkcje otwarte (rozwijane)

Funkcje constexpr

Referencje

- *w parametrach*
- *w deklaracjach*

Referencje / wskaźniki

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Przeciążanie funkcji i operatorów
Parametry funkcji
Funkcje otwarte (rozwijane)
Funkcje constexpr
Referencje
- *w parametrach*
- *w deklaracjach*

Referencje / wskaźniki

Politechnika Warszawska

Wartości zmiennych


```
int j;
const int ci = 7;

j = j + 20;

// @1 = j + 20;
// j = @1;
// usuń @1
7 = i;    // Błąd: wymagana L-wartość
ci = 10; // Błąd
fun() = 4; // OK ?
```

L-wartość

R-wartość

kod zastępczy

L-wartość, ale zakaz modyfikacji

```
int x;          // C++

int& fun() {
  return x;
}
```

- **L-wartość** (*L-value*)
  Wartość związana z przechowywaną w pamięci daną. Można pobrać jej adres.
- **R-wartość** (*R-value*)
  Wartość tymczasowa służąca do chwilowego przechowania wyniku. Nie można pobrać jej adresu

Politechnika Warszawska

Referencje (odniesienia)


```
Typ x;          // Typ "zwykły"
Typ *p;         // Wskazanie
Typ &lr = ....; // Referencja (L-referencja)
Typ &&rr= ....; // Referencja (R-referencja) [C++11]
```

Kontekst parametryzacji funkcji

Wiązanie przez wartość: parametry są zmiennymi lokalnymi odpowiedniego typu; argumenty wywołania określają wartość początkową parametrów formalnych.

Funkcje czyste: funkcje bez efektów ubocznych – nie zmieniają obiektów nielokalnych, nie wykonują operacji WE/WY; wynik jest dostarczany poprzez zwróconą wartość, np.

```
int nwp(int a, int b);
```

Funkcje z deklarowanymi efektami dodatkowymi: umiejscowienie skutków ubocznych określone przez parametry (wskazania albo referencje).

Funkcje z dowolnymi efektami ubocznymi: działają bezpośrednio na zmiennych globalnych. **NIE UŻYWAĆ!**

Referencje w parametrach (cd1)


```
void swap(int *a, int *b) // Wskazania
{ int temp = *a;
  *a = *b; *b = temp;
}
```

operator wyłuskiwania

W języku C argumenty przekazywane są zawsze przez wartość!

```
void swap(int &a, int &b) // Referencje/Odniesienia
{ int temp = a;
  a = b; b = temp;
}

void test() {
 swap(&x, &y);  // C - pobranie(&) i przekazanie wskazania
                // na zmienną
 swap(x, y);    // C++ - przekazanie przez referencje
}
```

Czy funkcja zmieni argumenty?

Referencje w parametrach (cd2)

- Referencyjny parametr formalny desygnuje obiekt podany w argumencie wywołania (wiązanie przez referencję); operacje na parametrze są operacjami na tym obiekcie.
- Do parametrów referencyjnych nie trzeba stosować operatora "wyłuskania" '*'.
- Parametry referencyjne zachowują się jak zmienne nielokalne, dostarczone przez funkcję wywołującą.
- Funkcja może zwracać referencję; zwracana referencja musi dotyczyć obiektu o czasie życia **dłuższym** niż czas aktywacji funkcji:
  - statycznego (ma czas życia programu)
  - lokalnego w funkcji wywołującej i przekazanego przez referencję lub wskazanie
  - utworzonego dynamicznie

Referencje w parametrach (cd3)


```
int& ref(int& ir)
{ /* ... */ return ir; } // OK
// Użycie, np.: ref(x) = ref(x)+1;

Typ& expose(void)
{ Typ local;
  // ...
  return local;  // Błąd
}
Node& search(Tree& t, Key &k);
// search(dict, word).n++;
```

zmienna local przestaje istnieć

Zwracanie referencji - przydatne przy przeciążaniu operatorów i programowaniu funkcji które zwracają zmodyfikowany obiekt (w konwencjach języka C zwracają wskazanie na obiekt)

Referencje w parametrach (cd4)

Używając parametrów referencyjnych albo wskaźników trzeba konsekwentnie stosować kwalifikator **const**
Kompilator czerpie wiedzę z prototypów funkcji!

```
void fun(int &a, const int &b);
```

będzie zmieniane | nie będzie zmieniane

Zalecana specyfikacja parametrów funkcji		
Parametr → Dostęp ↓	`Small`	`Big`
Niemodyfikujący	`Small`	`const Big &` `const Big *`
Modyfikujący	`Small &` `Small *`	`Big &` `Big *`

Referencje w parametrach [C++11] (cd5)

L-referencje i R-referencje

Możliwość wiązania	Parametr funkcji → Argument wywołania ↓	L-referencje		R-referencje	
		`Typ&`	`Typ const &` `const Typ &`	`Typ&&`	`Typ const &&` `const Typ &&`
L-wartość	`Typ x;` `Typ& gen(.);`	+	+	+	+
L-wartość bez modyfikacji	`const Typ cx=...;` `const Typ& cgen(.);`		+		+
R-wartość	`Typ(.);` `Typ tfun(.);`		+	+	+
R-wartość bez modyfikacji	`const Typ ctfun(.);`		+		+

```
void f(int &a);
f(x+y); // Błędne wywołanie
```

Referencje w parametrach (cd6)

Skojarzenie z parametrem l-referencyjnym wyrażenia generującego r-wartość jest dopuszczalne tylko w przypadku l-referencji na obiekt const

```
void swap(int &a, int &b);
// Wywołanie: swap(x,x+y); jest błędne.

void print2(const string& s, const string &t)
{ cout<<s<<" --- "<<t<<endl;
}
// ...
print2(s1 + "(1)", s2);
// @1 = s1 + "(1)";
// print2(@1, s2);
// usuń @1
```

kod zastępczy

Prosty Quiz

Jaki jest wynik zwracany przez funkcję?

```
int ref_fun_add(int &a, int &b)
{
  a = 2;
  b = 5;
  return(a + b);
}
```

```
int i1 = 5, i2 = 6;
ref_fun_add(i1, i2) = ??
ref_fun_add(i1, i1) = ??
```

Referencje w deklaracjach

Zmienna referencyjna ≡ pseudonim zmiennej

```
Typ x,/*...*/ &r=x, /*...*/;
```

Inicjowanie zmiennej referencyjnej jest obowiązkowe; skojarzenia referencji z obiektem inicjującym nie można zmienić

- Wszelkie operacje nad zmienną referencyjną (r) dotyczą implicite obiektu użytego w inicjacji (x)
- Przydatność ograniczona – można skrócić zapis odwołań do głęboko zanurzonych podobiektów, np:

```
int &rx = Mat[0][0].Tab[0].x;
rx++; // zamiast Mat[0][0].Tab[0].x++;
```

- Inicjacja zmiennej referencyjnej nie dotyczy deklaracji ze specyfikacją `extern` lub jeśli deklarowana jest składowa klasy; inicjacja jest potrzebna w miejscu definicji zmiennej albo w konstruktorze.
- Przekazanie argumentu skojarzonego z referencją jest inicjacją zmiennej referencyjnej (parametr formalny staje się pseudonimem argumentu wywołania).

Referencje w deklaracjach - przykłady


```
int x, i = 1;
int* ip = &i; // ip wskazuje i;
int& ir = i;  // ir jest pseudonimem i
x = *ip + ir; // x = i + i;
ip = &ir;     // ip dalej wskazuje i;
ip = &x;      // ip wskazuje x; *ip==2;
ir = x;       // Równoważne i=x;
int& r2 = ir; // rr drugi pseudonimem i
//int& & r1 = 44; // BŁĄD: niedozwolona para & &
int&& rr1 = 44; // C++11  r-referencja
++rr1;          // rr1= 45
int&& rR = ir;        // BŁĄD: niedozwolone wiązanie
int&& rR = (int&&)ir; // Jawna konwersja typu
typedef int& refint;
refint& r3 = ir;      // C++11 (kolejny pseudonim)
const int&& cr1 = 44; // C++11
cr1++;                //BŁĄD: const
int*& ref2ptr2int = ip; // OK: pseudonim wskazania
int&* ptr2ref2int = &ir;// BŁĄD: wskazanie na referencję
int& T[2] = { i, x };   // BŁĄD: niedozwolone tab. referencji jak wyżej
typedef int Tab10[10];
Tab10 yy;
const Tab10& xx = yy;  // Poprzez xx możliwy tylko odczyt
yy[0] = 5;
yy[1] = xx[0] + 1;     // yy[1]==6
xx[2] = 2;             // BŁĄD: xx jest const
```

Referencje czy wskaźniki

Referencje

- Referencji nie można zmieniać
- Referencja ma taki sam adres jak zmienna (niezależnie od implementacji)
- Nie alokuje pamięci (teoretycznie)
- Nie ma referencji do referencji
- Nie można „zerować”
- Nie można utworzyć tablicy referencji
- Stała referencja może być związana z wartością tymczasową
- Dostęp do zmiennej przez nazwę

Wskaźniki

- Wskaźnik można zmieniać
- Wskaźnik przechowuje (w komórkach pamięci) adres zmiennej
- Wskaźnik to zmienna przechowująca adres
- Można używać wskazania na wskaźniki
- Wskaźnik można „wyzerować” (**nullptr**)
- Działają operatory ++
- Dostęp do zmiennej przez dereferencję(*) lub w przypadku struktur (->)