

podstawy i języki
programowania

pjp

Pascal

C++

Podstawy programowania

Część piąta

Podprogramy koncepcja, zastosowania

Autor

Roman Simiński

Kontakt

roman.siminski@us.edu.pl

www.programowanie.siminskionline.pl

Niniejsze opracowanie zawiera skrót treści wykładu, lektura tych materiałów nie zastąpi uważnego w nim uczestnictwa. Opracowanie to jest chronione prawem autorskim. Wykorzystywanie jakiegokolwiek fragmentu w celach innych niż nauka własna jest nielegalne. Dystrybuowanie tego opracowania lub jakiegokolwiek jego części oraz wykorzystywanie zarobkowe bez zgody autora jest zabronione.

W programach często występują powtarzające się fragmenty kodu

```
. . .  
cout << "Sredni dochod to: " << dochod;
```

```
cout << "Nacisnij Enter by kontynuowac...";  
cin.get();
```

```
cout << "Dochody uporządkowane rosnaco:";  
. . .
```

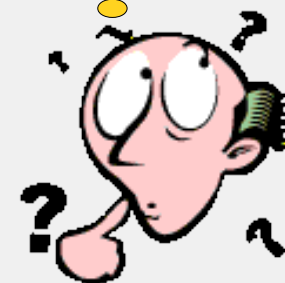
```
cout << "Nacisnij Enter by kontynuowac...";  
cin.get();
```

```
cout << "Dochod minimalny:" << min;  
cout << "Dochod maksymalny:" << maks;  
. . .
```

```
cout << "Nacisnij Enter by kontynuowac...";  
cin.get();
```

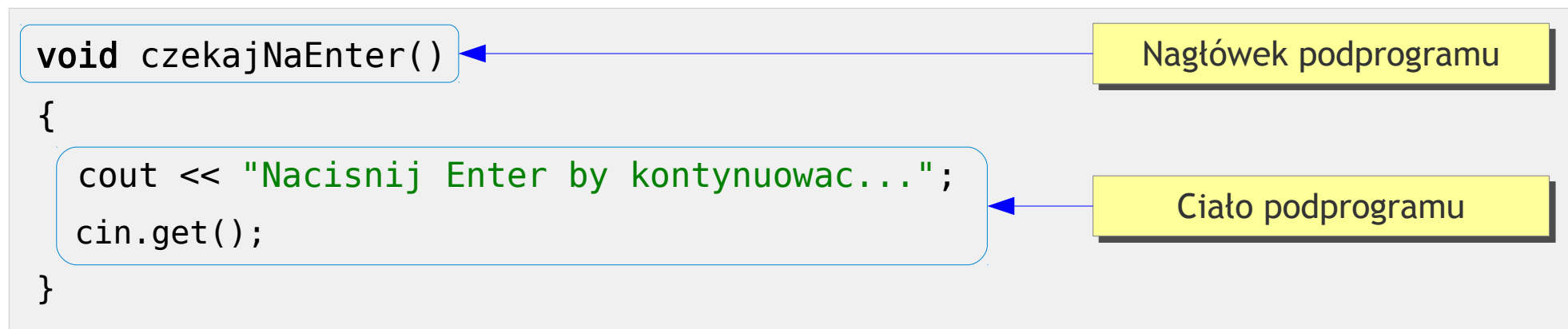
```
. . .
```

Tyle razy pisać
to samo...?

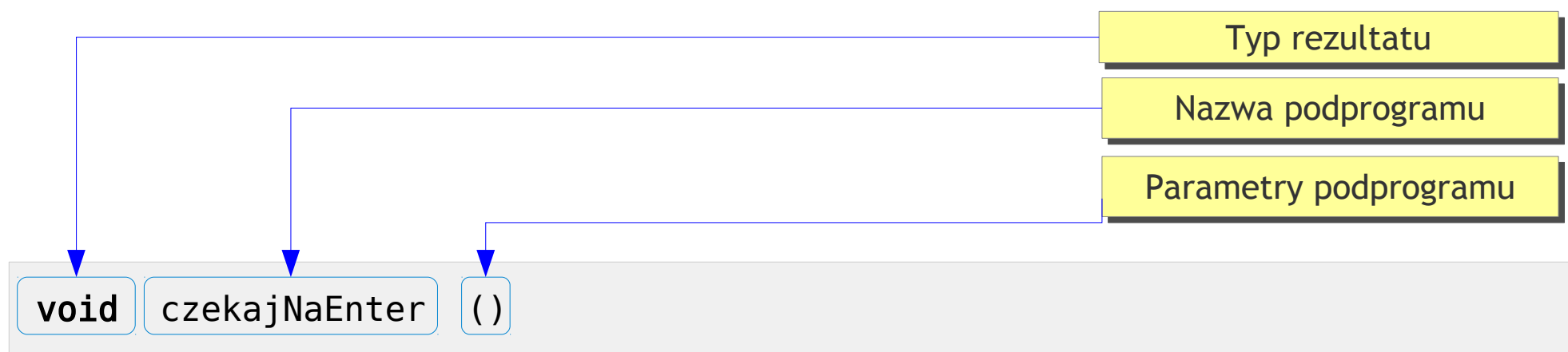


Powtarzające się fragmenty kodu jako „klocek” – podprogram

▶ Definicja podprogramu



▶ Nagłówek podprogramu



Słowo kluczowe **void** należy rozumieć jako *nic, brak wartości*

Aby podprogram zadziałał należy go wywołać

```
void czekajNaEnter()
```

```
{
    cout << "Nacisnij Enter by kontynuowac...";
    cin.get();
}
```

Definicja podprogramu

```
. . .
cout << "Sredni dochod to: " << dochod;
```

```
czekajNaEnter();
```

```
cout << "Dochody uporządkowane rosnaco:";
. . .
```

```
czekajNaEnter();
```

Wywołanie podprogramu

```
cout << "Dochod minimalny:" << min;
cout << "Dochod maksymalny:" << maks;
. . .
```

```
czekajNaEnter();
```

```
. . .
```

Wywołanie podprogramu

```
void czekajNaEnter()
```

```
{
    cout << "Nacisnij Enter by kontynuowac...";
    cin.get();
}
```

```
. . .
cout << "Sredni dochod to: " << dochod;
```

```
czekajNaEnter();
```

```
cout << "Dochody uporządkowane rosnaco:";
. . .
```

```
czekajNaEnter();
```

```
cout << "Dochod minimalny:" << min;
cout << "Dochod maksymalny:" << maks;
. . .
```

```
czekajNaEnter();
. . .
```

1

2

3

1

Wywołanie podprogramu

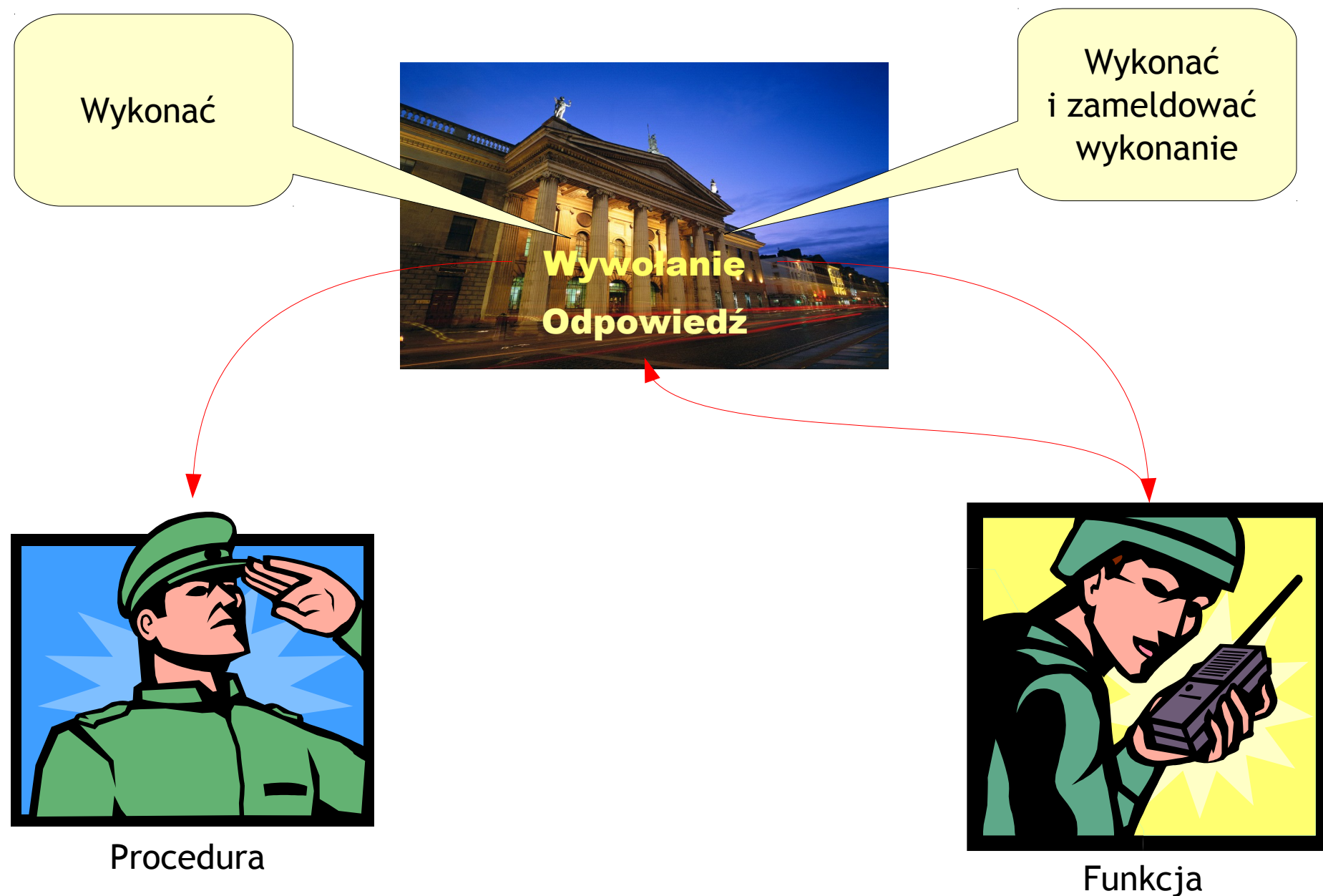
2

Wykonanie ciała podprogramu

3

Powrót podprogramu

Procedura a funkcja – rodzaje podprogramów



Rodzaje podprogramów

- ▶ Podprogramy dzielimy na *procedury* i *funkcje*.
- ▶ **Procedura**, to podprogram, który wykonuje *akcję* określoną *instrukcjami* zapisanymi w ciele podprogramu i **już!**

```
. . .  
czekajNaEnter();  
. . .
```

- ▶ **Funkcja**, to podprogram, który wykonuje *akcję* określoną *instrukcjami* zapisanymi w ciele podprogramu i **oddaje w miejscu wywołania pewien rezultat!**

```
. . .  
x = sinus( 0 );  
. . .
```

Rezultat funkcji, udostępniany w miejscu wywołania,
po powrocie z podprogramu

Rodzaje podprogramów

- ▶ W języku C/C++ *nie występuje* podział podprogramów na *procedury* i *funkcje*.
- ▶ Wszystkie podprogramy są *funkcjami*.
- ▶ Istnieje jednak możliwość wykorzystywania funkcji jak procedur, bądź deklarowania funkcji tak, by przypominały procedury.
- ▶ Słowo kluczowe *void*, będące nazwą typu, oznacza *brak, nieobecność jakiegolwiek wartości*.
- ▶ Jeżeli typem rezultatu będzie typ określany słowem kluczowym *void*, to oznacza, iż funkcja *nie udostępnia rezultatu* – staje się wtedy czymś podobnym do procedury z języka Pascal.

```
void nazwa_funkcj_działającej_jak_procedura()  
{  
    coś tam, coś tam...  
}
```


Procedury – podsumowanie informacji

- ▶ Procedura to programowe narzędzie realizujące określone czynności.
- ▶ Każda procedura ma swoją *nazwę*.
- ▶ Wpisanie *nazwy* procedury w kodzie programu oznacza jej *wywołanie*.
- ▶ Wywołanie procedury polega na:
 - zawieszeniu wykonania aktualnie realizowanego ciągu instrukcji,
 - wykonaniu instrukcji przypisanych do procedury o danej nazwie,
 - wznowieniu wykonania realizowanego ciągu instrukcji, począwszy od instrukcji następnej po wywołaniu procedury.

```
. . .  
wswietlKomunikat( "Uwaga, niepoprawne dane!" );  
  
. . .  
czekajNaEnter();  
  
. . .
```

Funkcje – podsumowanie informacji

- ▶ Funkcja to programowe narzędzie realizujące określone czynności, po wykonaniu których, funkcja udostępnia w miejscu wywołania pewien *rezultat*.
- ▶ Żargonowo mówi się, że funkcja *oddaje wartość w miejscu wywołania*.
- ▶ Funkcja różni się od procedury tym, że ta ostatnia tylko *wykonuje czynności* i *nie udostępnia rezultatu* w miejscu wywołania.
- ▶ Poza tą różnicą *procedury* i *funkcje* są podobne — wspólnie nazywa się je *podprogramami*.

```
. . .  
x = 2 * R + sinus( alfa ) ;  
  
. . .  
delta = wyznaczDelte( 5, 2, 8 );    // Delta równania:  $5x^2 + 2x + 8 = 0$   
  
. . .  
przyprostokatna = przeciwprostokatna * sinus( alfa );
```

Podprogramy mogą mieć parametry

- ▶ **Paramtery** (inaczej *argumenty*) to informacje przekazywane do wnętrza podprogramu.

Brak *parametrów*, dodatkowe informacje nie są potrzebne wewnątrz podprogramu

```
. . .  
czekajNaEnter ( ) ;  
. . .
```

- ▶ Parametry mogą, ale nie muszą występować. Dotyczy to zarówno procedur jak i funkcji.

```
. . .  
x = sinus( 0 ) ;  
. . .
```

Niektóre funkcje muszą dostawać *parametry*, ciało funkcji sinus musi wiedzieć, dla jakiego kąta ma zostać wyliczona jego wartość

Rola parametrów

- ▶ Instrukcje wykonywane wewnątrz podprogramów są zwykle *odseparowane* od reszty programu.
- ▶ Można powiedzieć, że te instrukcje są „uwięzione” wewnątrz „celi” jaką jest procedura lub funkcja.
- ▶ Aby instrukcje wewnętrzne podprogramu „wiedziały” o naszych chciejstwach, musimy im *przekazać informacje* oficjalnym kanałem wymiany informacji.



Rola parametrów

- ▶ Oficjalny kanał wymiany informacji z instrukcjami wewnętrznymi podprogramów to *parametry* (zwane też *argumentami*).



Parametry - kanał wymiany informacji

Rola parametrów

- ▶ Parametry mogą być ustalane na zewnątrz podprogramu i przekazywane *do jego wnętrza*, mają wtedy charakter *wejściowy*.



Rola parametrów

- Wartości parametrów mogą być ustalane *we wnętrzu* podprogramu, i stanowić odpowiedź podprogramu, mają wtedy charakter *wyjściowy*.



Rola parametrów

- ▶ Parametry mogą być ustalone służyć jednocześnie do komunikacji dwustronnej.



Pierwsza własna funkcja

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}

int main()
{
    double dlugosc_boku, pole;

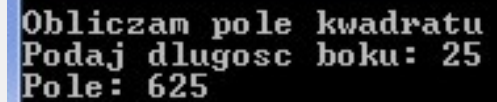
    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;

    pole = oblicz_pole_kwadratu( dlugosc_boku );

    cout << "Pole: " << pole;

    return EXIT_SUCCESS;
}
```

A screenshot of a terminal window showing the output of the program. The text is white on a black background. It reads: "Obliczam pole kwadratu", "Podaj dlugosc boku: 25", and "Pole: 625".

```
Obliczam pole kwadratu
Podaj dlugosc boku: 25
Pole: 625
```

Pierwsza własna funkcja – przed wywołaniem, wczytanie danych

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}
```

```
int main()
{
    double dlugosc_boku, pole;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";
```

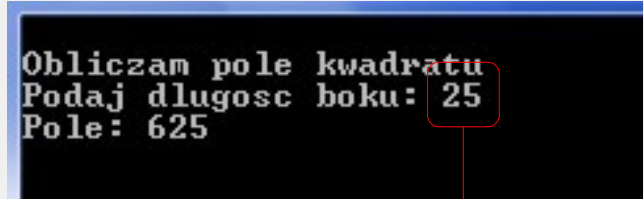
```
» cin >> dlugosc_boku;
```

```
pole = oblicz_pole_kwadratu( dlugosc_boku );
```

```
cout << "Pole: " << pole;
```

```
return EXIT_SUCCESS;
```

```
}
```



```
Obliczam pole kwadratu
Podaj dlugosc boku: 25
Pole: 625
```

dlugosc_boku

pole

	?
	?

Pierwsza własna funkcja – przed wywołaniem, parametr aktualny

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}

int main()
{
    double dlugosc_boku, pole;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;
```

dlugosc_boku
pole

	25
	?

```
» pole = oblicz_pole_kwadratu( dlugosc_boku );
```

```
cout << "Pole: " << pole;
```

```
return EXIT_SUCCESS;
```

```
}
```

To jest *parametr aktualny* wywołania podprogramu!
To jest bardzo ważne pojęcie – trzeba je
zrozumieć i zapamiętać!

Pierwsza własna funkcja – wywołanie funkcji, parametr formalny

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
» double oblicz_pole_kwadratu( double bok )
```

```
{
    return bok * bok;
}
```

```
int main()
{
```

```
    double dlugosc_boku, pole;
```

```
    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";
```

```
    cin >> dlugosc_boku;
```

```
    pole = oblicz_pole_kwadratu( dlugosc_boku );
```

```
    cout << "Pole: " << pole;
```

```
    return EXIT_SUCCESS;
```

```
}
```

bok	25	
dlugosc_boku	25	
pole	?	

To jest *parametr formalny* podprogramu!
To jest bardzo ważne pojęcie – trzeba je zrozumieć
i zapamiętać!

Parametr formalny to zmienna komunikacyjna

- ▶ Parametr formalny to *specjalna zmienna komunikacyjna*, zadeklarowana wewnątrz podprogramu.
- ▶ Jej wartość ustalana jest na *etapie wywołania podprogramu* na podstawie *parametru aktualnego wywołania*.
- ▶ Patrząc z punktu widzenia *wnętrza* podprogramu:

```
double oblicz_pole_kwadratu( double bok )  
{  
    return bok * bok;  
}
```

Co mi tu
znowu
dzisiaj dali...

Wartość, która jest kopią parametru
aktualnego wywołania

Parametr formalny podprogramu



Pierwsza własna funkcja – wywołanie funkcji, przekazanie parametrów

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
» double oblicz_pole_kwadratu( double bok )
```

```
{
    return bok * bok;
}
```

```
int main()
{
```

```
    double dlugosc_boku, pole;
```

```
    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";
```

```
    cin >> dlugosc_boku;
```

```
    pole = oblicz_pole_kwadratu( dlugosc_boku );
```

```
    cout << "Pole: " << pole;
```

```
    return EXIT_SUCCESS;
```

```
}
```

25

bok
dlugosc_boku
pole

bok	25
dlugosc_boku	25
pole	?

Przekazanie parametrów przez wartość – wartość parametru aktualnego jest kopiowana do parametru formalnego podprogramu.

Przekazywanie parametrów przez wartość

- ▶ Na etapie wywołania podprogramu wyznaczana jest *wartość parametru aktualnego* wywołania.
- ▶ Wyznaczona wartość jest *kopiowana* do *parametru formalnego* podprogramu.
- ▶ Operacje na *parametrach formalnych* wykonywane wewnątrz podprogramu nie przenoszą się na *parametry aktualne*.
- ▶ Parametry aktualne mogą być *literałami*, *stałymi* i *zmiennymi*.

Pierwsza własna funkcja – wykonanie funkcji

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
» return bok * bok ;
}

int main()
{
    double dlugosc_boku, pole;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;

    pole = oblicz_pole_kwadratu( dlugosc_boku );

    cout << "Pole: " << pole;

    return EXIT_SUCCESS;
}
```

625

bok
dlugosc_boku
pole

25	
25	
?	

Pierwsza własna funkcja – po powrocie funkcji

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}

int main()
{
    double dlugosc_boku, pole;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;

    pole = oblicz_pole_kwadratu( dlugosc_boku );

    » cout << "Pole: " << pole;

    return EXIT_SUCCESS;
}
```

dlugosc_boku
pole

	25
	625

Pierwsza własna funkcja – drobna optymalizacja

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}

int main()
{
    double dlugosc_boku;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;

    cout << "Pole: " << oblicz_pole_kwadratu( dlugosc_boku );

    return EXIT_SUCCESS;
}
```

Zmienna *pole* jest niepotrzebna, rezultat funkcji może być przekazany do strumienia wyjściowego bezpośrednio.

Przekazywanie parametrów przez wartość (język C i C++)

```
void inc( int i )  
{  
    ++i;  
}  
  
...  
  
int a = 5;  
  
inc( a );  
  
cout << "a = " << a;
```



Przekazywanie parametrów przez wartość (język C i C++)

```
void inc( int i )
{
    ++i;
}

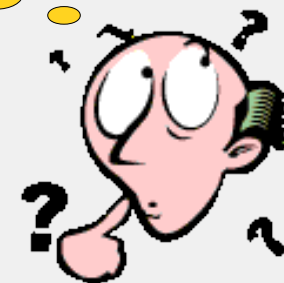
. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

Jaki jest stan
pamięci przed
wywołaniem?



Przed wywołaniem
inc(a)

a	5

Przekazywanie parametrów przez wartość (język C i C++)

```
void inc( int i )
{
    ++i;
}

. . .

int a = 5;

inc( a );

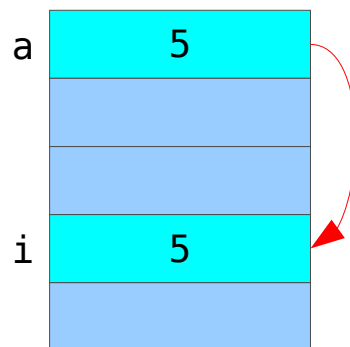
cout << "a = " << a;
```



Przed wywołaniem
inc(a)



Wywołanie
inc(a)



Przekazywanie parametrów przez wartość (język C i C++)

```
void inc( int i )
{
    ++i;
}

. . .

int a = 5;

inc( a );

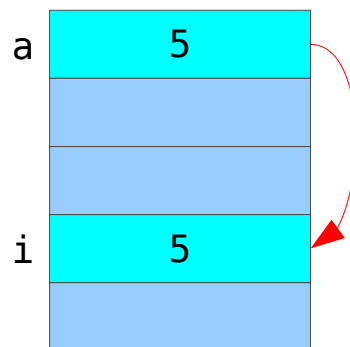
cout << "a = " << a;
```



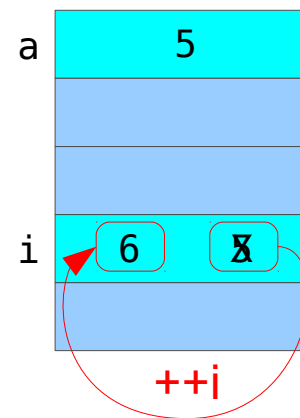
Przed wywołaniem
inc(a)



Wywołanie
inc(a)



Wykonanie
inc(a)



Przekazywanie parametrów przez wartość (język C i C++)

```
void inc( int i )
{
    ++i;
}

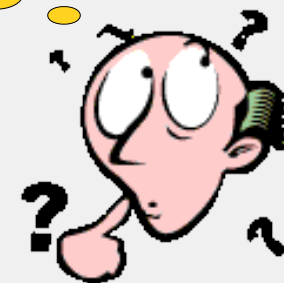
. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

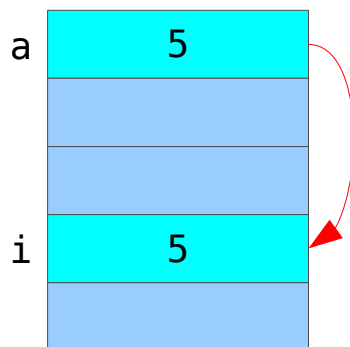
Jaki jest stan
pamięci po
wywołaniu?



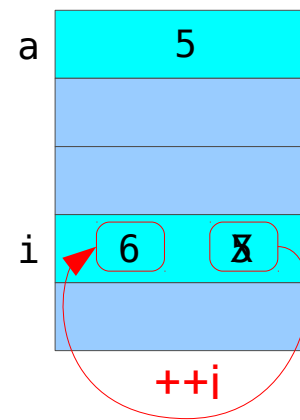
Przed wywołaniem
inc(a)



Wywołanie
inc(a)



Wykonanie
inc(a)



Po wykonaniu
inc(a)



Przekazywanie parametrów przez wartość (język C i C++)

```
void inc( int i )  
{  
    ++i;  
}  
  
...  
  
int a = 5;  
  
inc( a );  
  
cout << "a = " << a;
```



- ▶ Przy przekazywaniu parametrów przez wartość, wartość *parametru aktualnego* wywołania funkcji **kopiowana** jest do *parametru formalnego* funkcji.
- ▶ Od tego momentu parametr aktualny i formalny są od siebie niezależne.
- ▶ Żadna modyfikacja *parametru formalnego* funkcji **nie przenosi** się na *parametr aktualny* wywołania — wewnątrz funkcji nie jest w stanie zmodyfikować parametru formalnego funkcji.

Przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )  
{  
    ++i;  
}
```

. . .

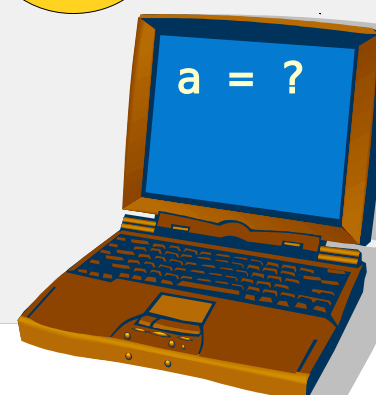
```
int a = 5;
```

```
inc( a );
```

```
cout << "a = " << a;
```

Parametr formalny *i* jest **referencją** do parametru aktualnego wywołania funkcji.

Co wyprowadzi program?



Przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )
{
    ++i;
}

. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

Jaki jest stan
pamięci przed
wywołaniem?



Przed wywołaniem
inc(a)

a	5

Przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )
{
    ++i;
}

. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

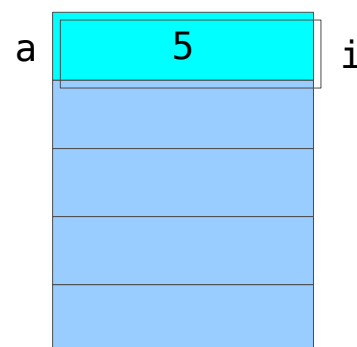
Co się dzieje w
trakcie
wywołania?



Przed wywołaniem
inc(a)



Wywołanie
inc(a)



Przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )
{
    ++i;
}

. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

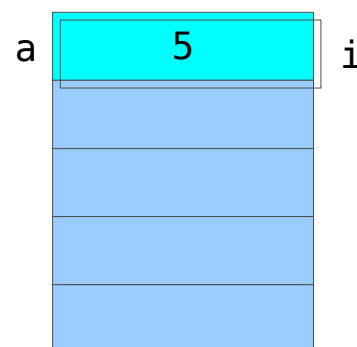
Co się dzieje w trakcie wykonania?



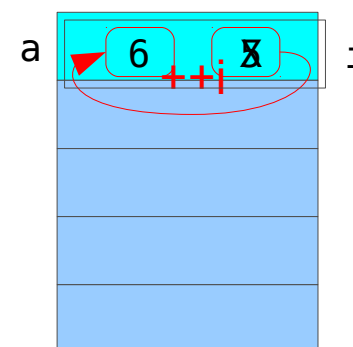
Przed wywołaniem
inc(a)



Wywołanie
inc(a)



Wykonanie
inc(a)



Przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )
{
    ++i;
}

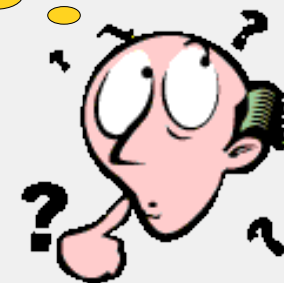
. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

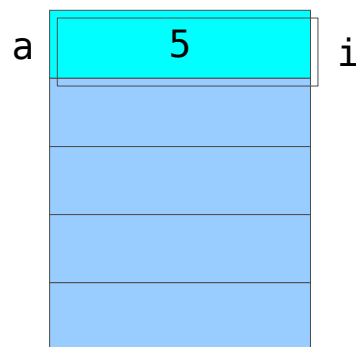
Jaki jest stan
pamięci po
wywołaniu?



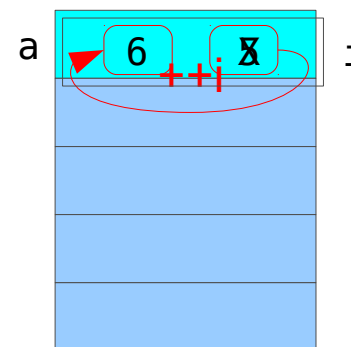
Przed wywołaniem
inc(a)



Wywołanie
inc(a)



Wykonanie
inc(a)



Po wykonaniu
inc(a)



Przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )  
{  
    ++i;  
}  
  
...  
  
int a = 5;  
  
inc( a );  
  
cout << "a = " << a;
```



- ▶ Przy przekazywaniu parametrów przez *referencję*, wywołania funkcji „nakłada” się na *parametr formalny funkcji*.
- ▶ Od tego momentu parametr *aktualny* i *formalny* odnoszą się do tej samej lokalizacji (adresu) w pamięci operacyjnej.
- ▶ Każda modyfikacja *parametru formalnego* funkcji **przenosi** się na *parametr aktualny* wywołania, wewnątrz funkcji może zmodyfikować parametr formalny funkcji.

Informacje wyjściowe z funkcji – rezultat czy parametr referencyjny?

Wczytywanie liczby — wczytana wartość jako rezultat funkcji:

```
double czytajDystans()  
{  
    double liczba;  
  
    do  
    {  
        cout << endl << "Podaj dystans: ";  
        cin >> liczba;  
  
        if( liczba <= 0 )  
            cout << "Dystans musi byc liczba dodatnia";  
    }  
    while( liczba <= 0 );  
  
    return liczba;  
}
```

```
double dystans;  
  
dystans = czytajDystans();
```

Informacje wyjściowe z funkcji – rezultat czy parametr referencyjny?

Wczytywanie liczby — wczytana wartość jako parametr referencyjny

```
void czytajDystans( double & liczba )
{
    do
    {
        cout << endl << "Podaj dystans: ";
        cin >> liczba;
        if( liczba <= 0 )
            cout << "Dystans musi byc liczba dodatnia";
    }
    while( liczba <= 0 );
}
```

```
double dystans;

czytajDystans( dystans );
```


Informacje wyjściowe z funkcji – rezultat czy parametr referencyjny?

Wczytywanie liczby – wczytana wartość jako rezultat funkcji:

```
double dystans;  
  
dystans = czytajDystans();
```

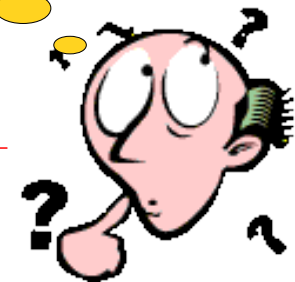
Wczytywanie liczby – wczytana wartość jako parametr referencyjny

```
double dystans;  
  
czytajDystans( dystans );
```

```
void czytajDystans( double & liczba )  
{  
    . . .  
}
```

```
void czytajDystans( double liczba )  
{  
    . . .  
}
```

Czy rzeczywiście
jest referencja?



Wiele danych wyjściowy – parametry referencyjne

```
void czytajPaliwoIDystans( double & p, double & d )
{
    do
    {
        cout << endl << "\nPodaj ilosc  paliwa: ";
        cin >> p;
        if( p <= 0 )
            cout << "Ilosc paliwa musi byc liczba dodatnia";
    }
    while( p <= 0 );

    do
    {
        cout << endl << "\nPodaj dystans: ";
        cin >> d;
        if( d <= 0 )
            cout << "Dystans musi byc liczba dodatnia";
    }
    while( d <= 0 );
}
```

```
double dystans, paliwo;
```

```
czytajPaliwoIDystans( paliwo, dystans );
```

Definicja funkcji po jej wywołaniu

```
int main()  
{  
    double dlugosc_boku, pole;  
  
    . . .  
  
    pole = oblicz_pole_kwadratu( dlugosc_boku );  
  
    . . .  
}
```

Wywołanie funkcji

```
double oblicz_pole_kwadratu( double bok )  
{  
    return bok * bok;  
}
```

Definicja funkcji

Czy
kompilatorowi
się to
spodoba?



Czy kompilatorowi się to spodoba?

```
int main()  
{  
    double dlugosc_boku, pole;  
  
    . . .  
  
    pole = oblicz_pole_kwadratu( dlugosc_boku );  
  
    . . .  
}
```

Wywołanie funkcji

```
double oblicz_pole_kwadratu( double bok )  
{  
    return bok * bok;  
}
```

Definicja funkcji

- ▶ *Tak, bez błędów i ostrzeżeń* — w starszych wersjach kompilatorów.
- ▶ *Tak, bez błędów lecz z ostrzeżeniem* — w nowszych wersjach kompilatorów, oraz tych pracujących w trybie zgodności z normą ANSI.
- ▶ *Nie, wystąpi błąd* — w kompilatorach pracujących w trybie C++.

Skąd te rozbieżności?

- ▶ *Definicja* funkcji występuje *po jej wywołaniu*.
- ▶ Kompilator na etapie wywołania jej jeszcze *nie zna*.
- ▶ Czyni w stosunku do niej założenia — że to funkcja, której *rezultatem jest wartość int*. To założenie może być słusznie albo nie.
- ▶ Aby kompilator mógł kontrolować poprawność wywołania funkcji, należy to wywołanie poprzedzić *definicją* lub *deklaracją* wywoływanej funkcji.
- ▶ Aby uniknąć niejednoznaczności, wprowadza się *prototypy funkcji*.
- ▶ *Deklaracja* przyjmuje postać *prototypu funkcji*.

Prototypy funkcji

Definicja funkcji:

```
double oblicz_pole_kwadratu( double bok )  
{  
    return bok * bok;  
}
```

Deklaracja — prototyp — funkcji:

```
double oblicz_pole_kwadratu( double bok );
```

Ogólna postać definicji funkcji:

```
typ_rezultatu nazwa_funkcji( lista_parametrów_formalnych )  
{  
    ciało_funkcji  
}
```

Ogólna postać prototypu funkcji:

```
typ_rezultatu nazwa_funkcji( lista_parametrów_formalnych );
```

Wykorzystanie prototypów funkcji – zadeklaruj funkcję potem wołaj

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok );

int main()
{
    double dlugosc_boku;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;

    cout << "Pole: " << oblicz_pole_kwadratu( dlugosc_boku );

    return EXIT_SUCCESS;
}

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}
```

Można jednak bez prototypu – zdefiniuj funkcje potem wołaj

```
#include <iostream>
#include <cstdlib>
using namespace std;

double oblicz_pole_kwadratu( double bok )
{
    return bok * bok;
}

int main()
{
    double dlugosc_boku;

    cout << endl << "Obliczam pole kwadratu";
    cout << endl << "Podaj dlugosc boku: ";

    cin >> dlugosc_boku;

    cout << "Pole: " << oblicz_pole_kwadratu( dlugosc_boku );

    return EXIT_SUCCESS;
}
```


Podsumowanie informacji o prototypach

- ▶ Starsze implementacje C dopuszczały wywoływanie funkcji wcześniej kompilatorowi nieznanym.
- ▶ W trakcie kompilowania wywołania nieznanej funkcji *przez domniemanie* przyjmowano, że jej *rezultatem jest wartość int* i *nic nie wiadomo* na temat jej parametrów. Nie pozwalało to kompilatorowi kontrolować poprawności wywołania funkcji.
- ▶ Aby kompilator mógł *kontrolować poprawność wywołania funkcji*, należy to wywołanie *poprzedzić definicją lub deklaracją* wywoływanej funkcji.
- ▶ Deklaracja przyjmuje postać prototypu funkcji.
- ▶ Deklaracja i definicja funkcji powinna być zgodna. Jeżeli w obrębie jednego pliku wystąpi niezgodność, kompilator zgłosi błąd kompilacji.

Podsumowanie informacji o podprogramach

Historycznie i technicznie pierwotna przyczyna wyodrębnienia podprogramów to eliminowanie powtarzających się fragmentów kodu.

Z czasem podprogramy stały się podstawowym środkiem podziału programu na mniejsze części, stając się podstawą dla *programowania proceduralnego*.

- ▶ Stosowanie podprogramów zwykle skraca program — zarówno *kod źródłowy* jak i *wynikowy*.
- ▶ Program staje się *czytelniejszy*.
- ▶ *Modyfikacje* programu stają się *łatwiejsze*.
- ▶ Łatwiejsze jest *lokalizowanie* i *eliminowanie* błędów.
- ▶ Program staje się podatniejszy na *modularyzację*.
- ▶ Łatwiej wyodrębnić zbiory spójnych podprogramów, stanowiące załączek potencjalnych *bibliotek*.

Przykład programu podzielonego na funkcje

```
#include <iostream>
#include <cmath>
using namespace std;

void komunikat_wstepny();
void oblicz();
float obwod_kola( float r );
float pole_kola( float r );

int main()
{
    komunikat_wstepny();
    oblicz();

    return EXIT_SUCCESS;
}

void komunikat_wstepny()
{
    cout << "\nObliczam obwod ...";
}
```

```
void oblicz()
{
    float r;

    cout << "\nPodaj promien R = ";
    cin >> r;

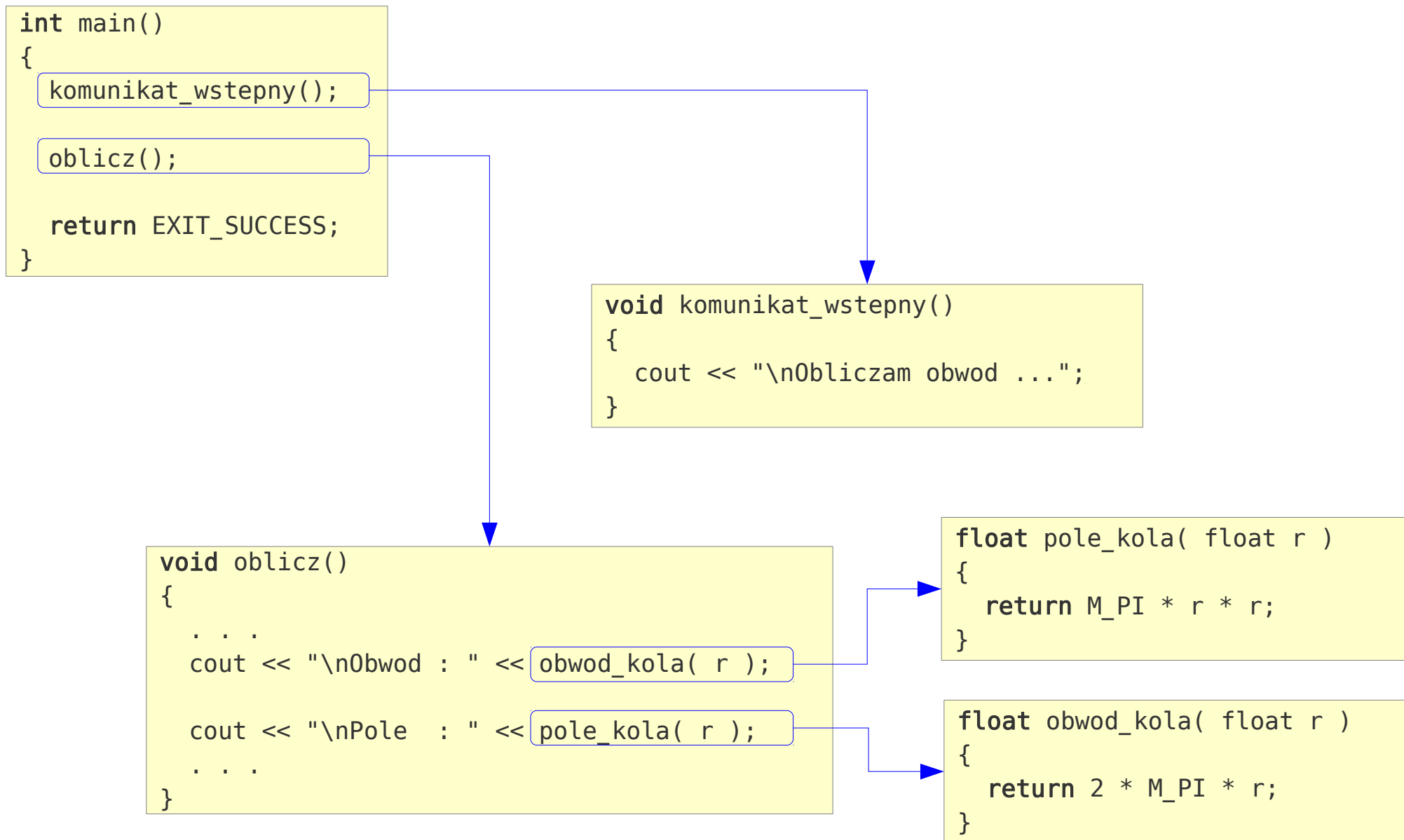
    cout << "\nObwod : " << obwod_kola( r );
    cout << "\nPole   : " << pole_kola( r );

    cout << "\nNacisnij Enter by ...";
    cin.ignore();
    cin.get();
}

float obwod_kola( float r )
{
    return 2 * M_PI * r;
}

float pole_kola( float r )
{
    return M_PI * r * r;
}
```

Programowanie zstępujące



Suplement I – przeciążanie funkcji

- ▶ *Przeciążanie funkcji* (ang. *function overloading*) – tworzenie większej liczby funkcji o takiej samej nazwie.
- ▶ Nazwa funkcji może być zatem użyta wielokrotnie do realizacji różnych czynności. Jest więc „przeciążona” dodatkowymi „obowiązkami”.
- ▶ Kompilator zadba o dobranie właściwej wersji funkcji przeciążonej w zależności od kontekstu jej wywołania.

```
int dodaj( int a, int b )           // 1-sza wersja funkcji przeciążonej add
{
    return a + b;
}

double dodaj( double a, double b ) // 2-ga wersja funkcji przeciążonej add
{
    return a + b;
}

cout << endl << "Dodawanie int      :" << dodaj( 1, 1 );
cout << endl << "Dodawanie double :" << dodaj( 1.0, 1.0 );
```

Uwaga, z przeciążaniem funkcji wiąże się szereg subtelnych niuansów, ich omówienie wykracza poza ramy tego wykładu. Należy zachować dużą ostrożność przy definiowaniu tego typu funkcji.

Suplement II – funkcje inline czyli funkcje wstawiane

- Funkcje *inline* nie są wywoływane w sposób klasyczny — ich kod jest umieszczany w miejscu wywołania i w rzeczywistości nie są one wywoływane.

```
inline int dodaj( int a, int b )  
{  
    return a + b;  
}
```

```
int main()  
{  
    int i = 2 , j = 2, k;  
    k = dodaj( i, j );  
    return EXIT_SUCCESS;  
}
```

```
int main()  
{  
    int i = 2 , j = 2, k;  
    {  
        k = i + j;  
    }  
    return EXIT_SUCCESS;  
}
```



Wywołanie podprogramu, opracowanie i przekazanie parametrów oraz powrót z podprogramu to dodatkowe instrukcje maszynowe — zatem instrukcje wpakowane do podprogramu wykonują się odrobinę wolniej. Jednak we większości typowych przypadków opóźnienie to jest nieznaczące.

Suplement II – funkcje inline czyli funkcje wstawiane, cd. ...

- Specyfikacja ze słowem kluczowym *inline* to tylko rekomendacja dla kompilatora — niektórych funkcji nie można w pełni rozwinąć i będą one wywoływane klasycznie (np. rekurencyjne).
- Funkcje wstawiane zastępują w języku C++ *makra*, stosowanie powszechnie w języku C.
- W porównaniu z makrami funkcje inline zapewniają kontrolę typów i wychwytywanie błędów na etapie kompilacji

Mechanizm funkcji zadeklarowanych jako *inline* przeznaczony jest do optymalizacji *małych, prostych i często wykorzystywanych funkcji*.

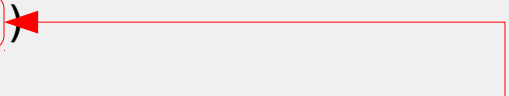
Kod wielokrotnie wykorzystujący pewną funkcję inline:

- *Może działać szybciej* — brak narzutu czasowego związanego z organizacją wywołania funkcji i powrotu z podprogramu;
- *Będzie dłuższy*, zawiera bowiem rozwinięcia ciała funkcji w miejscu jej każdorazowego wywołania.

Suplement III – parametry domyślne

- ▶ Bardzo często przy wywoływaniu funkcji, przy kolejnych wywołaniach pewne parametry się powtarzają.
- ▶ Powtarzające się parametry aktualne wywołania można ustawić jako *parametry domyślne*, można je pominąć przy wywołaniu. Jako parametr aktualny zostanie przyjęta wartość domyślna.

```
void outInt( int value, bool asDecimal = true )  
{  
    cout << (( asDecimal ) ? dec : hex ) << value;  
}
```



Parametr domyślny

Przykładowe wywołania:

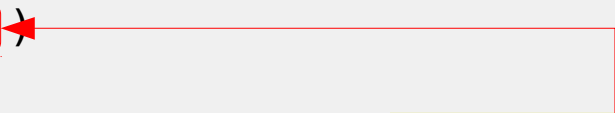
```
outInt( 22 );           // Parametr asDecimal otrzyma wartosc true  
  
outInt( 22, true );    // Parametr asDecimal otrzyma wartosc jak w wywołaniu  
outInt( 22, false );   // Parametr asDecimal otrzyma wartosc jak w wywołaniu
```


Suplement III – parametry domyślne, cd. ...

```
enum outFormat
{
    AS_DEC,
    AS_OCT,
    AS_HEX
};

void outInt( int value, int base = AS_DEC )
{
    switch( base )
    {
        case AS_DEC : cout << dec;
                       break;
        case AS_OCT : cout << oct;
                       break;
        case AS_HEX  : cout << hex;
                       break;
    }
    cout << value;
}

. . .
outInt( 10, AS_OCT );
outInt( 10, AS_HEX );
outInt( 10 );           // Parametr domyslny AS_DEC
```



Parametr domyślny

Suplement IV – zmienna liczba parametrów

- ▶ Czasem trudno oszacować, ile będzie parametrów wywołania. W językach C/C++ można definiować funkcje ze *zmienną liczbą parametrów*.
- ▶ Do obsługi zmiennej liczby parametrów służą makra zdefiniowane w *stdarg.h*.
- ▶ Parametry zmienne oznaczane są w nagłówku funkcji znakiem operatorem ...
- ▶ Scenariusz obsługi zmiennej liczby argumentów:
 - Definicja zmiennej identyfikującej parametry zmienne (typ *va_list*),
 - Ustalenie początku listy parametrów zmiennych (makro *va_start*),
 - Pobranie kolejnych parametrów zmiennych (makro *va_arg*),
 - Zakończenie pobierania parametrów zmiennych (makro *va_end*).

Uwaga, aby to zadziałało, funkcja musi posiadać przynajmniej jeden parametr „zwykły”, oraz funkcja musi „wiedzieć” jakie są typy kolejnych argumentów zmiennych.

Suplement IV – zmienna liczba parametrów, cd. ...

```
void varArgsFunction( jakisTyp normalnyParametr, ... )
{
    typParametru parametr;
    va_list argList;
    . . .
    va_start( argList, normalnyParametr );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    va_end( argList );
    . . .
}
```

Typ zmiennej używanej do wydobywania kolejnych parametrów funkcji.

Suplement IV – zmienna liczba parametrów, cd. ...

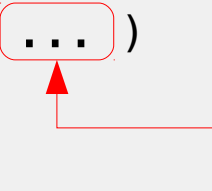
```
void varArgsFunction( jakisTyp normalnyParametr, ... )
{
    typParametru parametr;
    va_list argList;
    . . .
    va_start( argList, normalnyParametr );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    va_end( argList );
    . . .
}
```



Obowiązkowy „normalny” parametr.

Suplement IV – zmienna liczba parametrów, cd. ...

```
void varArgsFunction( jakisTyp normalnyParametr, ... )  
{  
    typParametru parametr;  
  
    va_list argList;  
    . . .  
    va_start( argList, normalnyParametr );  
    . . .  
    parametr = va_arg( argList, typParametru );  
    . . .  
    parametr = va_arg( argList, typParametru );  
    . . .  
    va_end( argList );  
    . . .  
}
```



Oznaczenie zmiennej części parametrów

Suplement IV – zmienna liczba parametrów, cd. ...

```
void varArgsFunction( jakisTyp normalnyParametr, ... )
{
    typParametru parametr;
    va_list argList;
    . . .
    va_start( argList, normalnyParametr );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    va_end( argList );
    . . .
}
```

Zakotwiczenie zmiennej *argList* o pierwszy parametr zmienny, czyli następny za ostatnim normalnym parametrem.

Suplement IV – zmienna liczba parametrów, cd. ...

```
void varArgsFunction( jakisTyp normalnyParametr, ... )
{
    typParametru parametr;
    va_list argList;
    . . .
    va_start( argList, normalnyParametr );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    va_end( argList );
    . . .
}
```

Wydobycie kolejnego parametru o określonym typie.



Suplement IV – zmienna liczba parametrów, cd. ...

```
void varArgsFunction( jakisTyp normalnyParametr, ... )
{
    typParametru parametr;
    va_list argList;
    . . .
    va_start( argList, normalnyParametr );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    va_end( argList );
    . . .
}
```

Wydobycie kolejnego parametru o określonym typie.

Suplement IV – zmienna liczba parametrów, cd. ...

```
void varArgsFunction( jakisTyp normalnyParametr, ... )
{
    typParametru parametr;
    va_list argList;
    . . .
    va_start( argList, normalnyParametr );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    parametr = va_arg( argList, typParametru );
    . . .
    va_end( argList );
    . . .
}
```

Zakończenie wydobywania parametrów.



Suplement IV – zmienna liczba parametrów, konkretny przykład

► Dodawanie dowolnej liczby danych typu *int*, wersja naiwna:

```
int addInts( int count, ... )
{
    int total = 0;
    va_list argList;

    va_start( argList, count );
    for( int i = 1; i <= count; i++ )
    {
        int value = va_arg( argList, int );
        total += value;
    }
    va_end( argList );
    return total;
}

cout << endl << addInts( 2, 1, 2 );
cout << endl << addInts( 3, 4, -1, 6 );
cout << endl << addInts( 0 );
cout << endl << addInts( 5, 1, 2, 3, 4, 5 );
```

Suplement IV – zmienna liczba parametrów, konkretny przykład

- Dodawanie dowolnej liczby danych typu *int*, wersja poprawiona:

```
int addInts( int count, ... )
{
    int total = 0;
    va_list argList;

    va_start( argList, count );
    for( ; count; count-- )
        total += va_arg( argList, int );
    va_end( argList );
    return total;
}

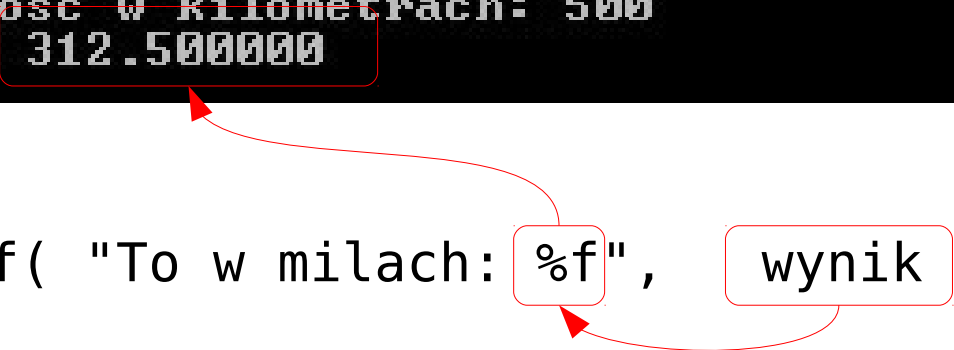
cout << endl << addInts( 2, 1, 2 );
cout << endl << addInts( 3, 4, -1, 6 );
cout << endl << addInts( 0 );
cout << endl << addInts( 5, 1, 2, 3, 4, 5 );
```

Suplement IV – zmienna liczba parametrów, printf jako przykład

- ▶ Funkcja *printf* pochodzi z biblioteki *stdio* z języka C.
- ▶ Funkcja *printf* wyprowadzane *sformatowane* dane do *stdout*.
- ▶ Pierwszy parametr funkcji, będący łańcuchem znaków, może zawierać *specyfikacje przekształceń*, rozpoczynające się znakiem %.
- ▶ W miejsce *specyfikatorów przekształceń* wstawiane są wartości kolejnych parametrów wywołania funkcji *printf*, sformatowane zgodnie z określonym formatem.

```
Przeliczanie odleglosci wyrazonej w kilometrach na mile  
Podaj odleglosc w kilometrach: 500  
To w milach: 312.500000
```

`printf( "To w milach: %f", wynik );`



Suplement IV – zmienna liczba parametrów, printf jako przykład

Sekwencje rozpoczynające się od znaku % stanowią specyfikacje przekształceń kolejnych parametrów funkcji *printf*:

- ▶ %d — wyprowadza liczbę całkowitą dziesiętną,
- ▶ %f — wyprowadza liczbę rzeczywistą,
- ▶ %c — wyprowadza znak,
- ▶ %s — wyprowadza napis.

Możliwości formatowania funkcji są bardzo szerokie, omówione zostaną osobno.

```
printf( "%s ma %d lat%c", "Ała", 18, '!' );
```

Suplement IV – zmienna liczba parametrów, printf jako przykład

```
Przeliczanie odleglosci wyrazonej w kilometrach na mile
Podaj odleglosc w kilometrach: 500
To w milach:      312.50_
```

```
printf( "To w milach: %10.2f",    wynik    );
```

```
Przeliczanie odleglosci wyrazonej w kilometrach na mile
Podaj odleglosc w kilometrach: 500
To w milach: 312.50    _
```

```
printf( "To w milach: %-10.2f",    wynik    );
```

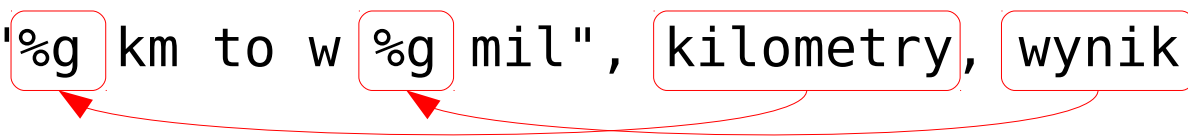
```
Przeliczanie odleglosci wyrazonej w kilometrach na mile
Podaj odleglosc w kilometrach: 500
To w milach: 312.50_
```

```
printf( "To w milach: %0.2f",    wynik    );
```

Suplement IV – zmienna liczba parametrów, printf jako przykład

```
Przeliczanie odleglosci wyrazonej w kilometrach na mile  
Podaj odleglosc w kilometrach: 500  
500 km to w 312.5 mil_
```

```
printf( "%g km to w %g mil", kilometry, wynik );
```



- ▶ %f — wyprowadza liczbę rzeczywistą,
- ▶ %g — wyprowadza liczbę rzeczywistą w najkrótszej postaci.

Czas zakończyć tego pdf-a!

Na chwilę kończymy z funkcjami ..., ale od teraz będą
one już na zawsze obecne w programowaniu!