



Uniwersytet  
Wrocławski

# C++: *funkcje*

Zbigniew Koza  
Wydział Fizyki i Astronomii

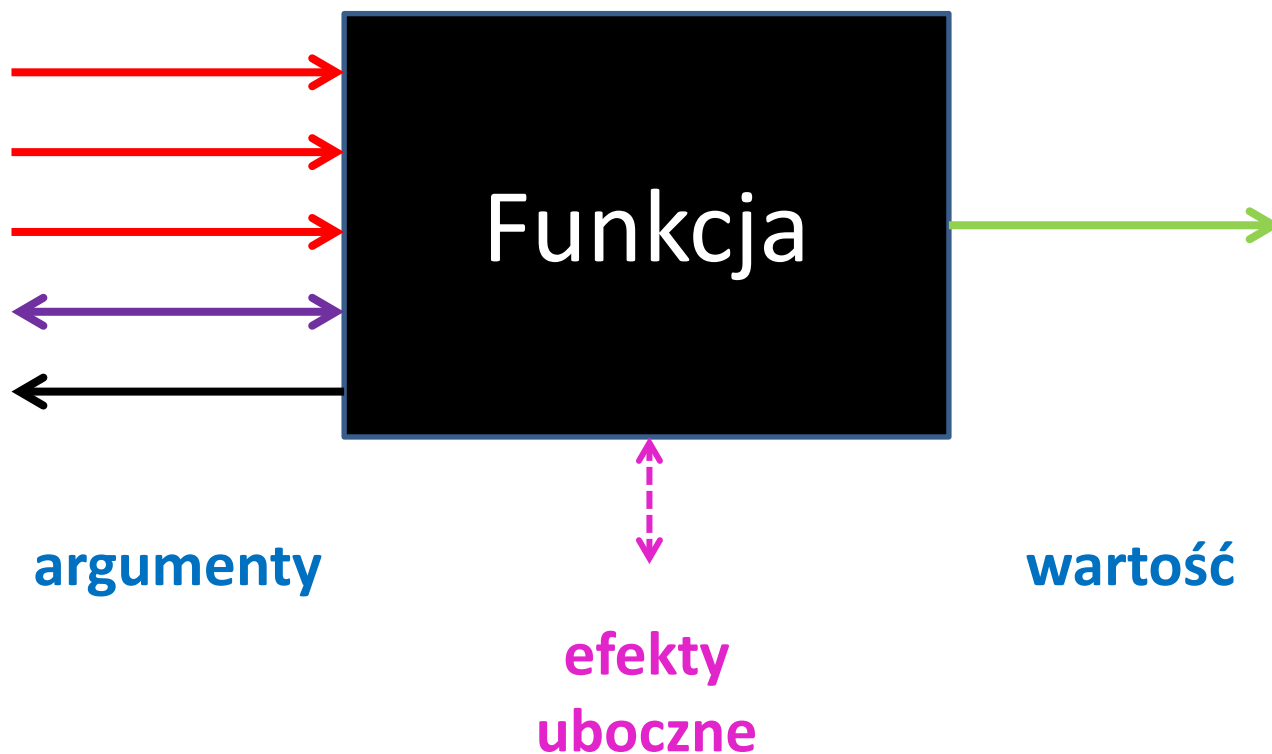
Wrocław

**CO TO SĄ FUNKCJE W C++?**

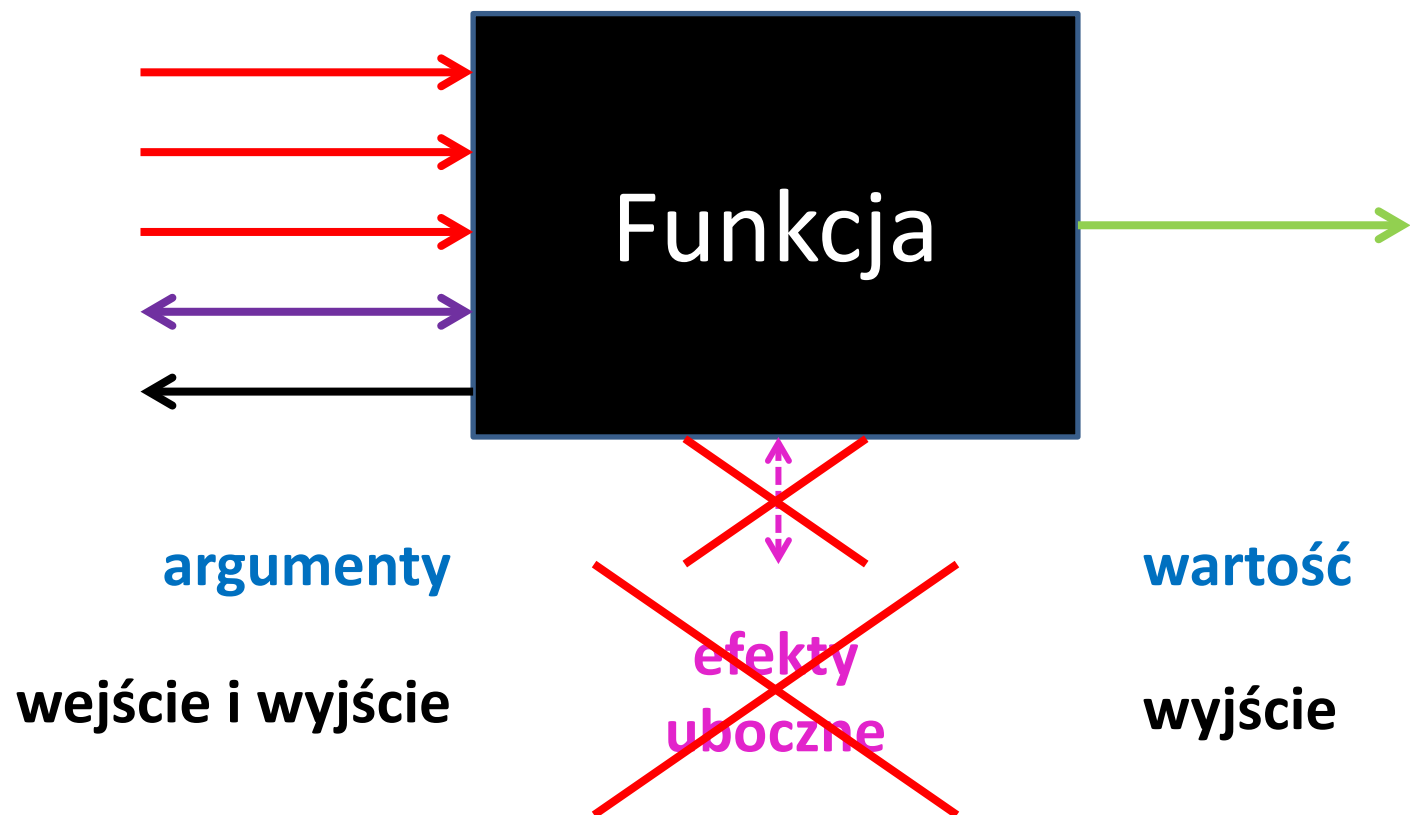
# Funkcja = podprogram

- Funkcja to nazwany fragment kodu, który wykonuje określone czynności na określonym zbiorze danych
- Funkcje umożliwiają modularyzację programów
- Modularyzacja umożliwia wielokrotne wykorzystywanie tego samego kodu
- A to z kolei ułatwia tworzenie, testowanie, uaktualnianie i utrzymywanie kodu

# Funkcja = „czarna skrzynka”



# Pożądana funkcja



# Argumenty i wartość

- Argumenty funkcji mogą służyć do:
  - Przekazania informacji do funkcji
  - Przekazania informacji z funkcji
  - Modyfikacji informacji przekazanej do funkcji
- Wartość przekazuje informację z funkcji
  - Wiele funkcji (zwłaszcza w bibliotekach języka C) przekazuje w wartości informację o swoim statusie w chwili zakończenia działania („kod błędu”)

```
int status = scanf ("%d", &i);
```

# Klasyfikacja funkcji

Kilka schematów klasyfikacji:

- Funkcje **swobodne** i **metody** klas
- Funkcje „zwyczajne”, **inline** i **wirtualne**
- Funkcje „zwyczajne” i **statyczne**
- Funkcje „zwyczajne”, **funktory** i **wyrażenia lambda**
- Funkcje „zwyczajne” i **rekurencyjne**
- Funkcje „zwyczajne” i **operatory**

**ARGUMENTY FUNKCJI**



# Argumenty funkcji

Funkcja może otrzymać swoje argumenty na wiele sposobów:

- **Przez wartość**
- **Przez wskaźnik** (\*, const\*, \*const, const\* const)
- **Przez referencję** (&, const&)
- **Przez referencję przenoszącą** (&&) [C++11]

```
int f(int n, int* p, int & r, int const& cr, int && mr)
int g(int* a, const int* b, int* const c,
      const int* const d);
```

# przez wartość

**int f(int n, int\* p, int & r, int const& cr, int && mr)**

- Argument przekazywany *przez wartość* jest **kopią oryginału**
- Jeśli argument jest obiektem, przekazanie go przez wartość wiąże się z wywołaniem *konstruktora klasy*
- Argument jest niszczone destruktorom klasy po zakończeniu działania funkcji

# przez wskaźnik

**int f(int n, int\* p, int & r, int const& cr, int && mr)**

- Do funkcji trafia wartość adresu zmiennej lub obiektu
- Sposób ten daje funkcji dostęp do oryginału obiektu, co umożliwia jej jego modyfikowanie
- Składnia jest „dziwna” (dużo gwiazdek etc.)
- Bez wywołania pary konstruktor/destruktor
- Bardzo popularna metoda w języku C

# przez referencję

**int f(int n, int\* p, int & r, int const& cr, int && mr)**

- Do funkcji trafia wartość adresu zmiennej lub obiektu, tak jak przy przekazaniu argumentu przez wskaźnik (brak pary konstruktor/destruktor)
- Funkcja operuje na oryginale argumentu
- Prosta składnia, jak przy przekazaniu przez wartość
- Bardzo popularna metoda w języku C++
- Używaj, jeśli chcesz modyfikować dany argument

# przez stałą referencję

```
int f(int n, int* p, int & r, int const& cr, int && mr)
```

- Technicznie jest to przekazanie argumentu przez referencję
- Plus **zobowiązanie**, że argument nie będzie przez funkcję modyfikowany
- Bardzo popularna metoda w języku C++
- Używaj, jeśli chcesz **odczytać** argument, który jest zbyt „ciężki” na przekazanie przez wartość

# przez referencję przenoszącą

**int f(int n, int\* p, int & r, int const& cr, int **&&** mr)**

- Technicznie jest to przekazanie argumentu przez referencję
- Plus informacja, że argument za chwilę straci swoją ważność (np. zostanie zniszczony)
- Wprowadzona w języku C++11
- Nie używaj, póki dobrze nie opanujesz C++
- Angielska nazwa: *rvalue reference*

# Referencja a stała referencja

```
void f(int & n) {...} // przez referencję
void g(int const& n) {...} // przez stałą referencję
int k = 8; // zmienna nazwana
f(k); // ok, można zbudować referencję do k
g(k); // ok, komentarz jw.
f(7); // błąd; nie istnieje referencja do literału 7
g(7); // ok; referencja do obiektu tymczasowego; konstruktor/destruktor
f(k + 1); // błąd; nie istnieje referencja do wyrażenia arytmetycznego
g(k + 1); // ok; referencja do obiektu tymczasowego; ctor/dtor
```

- Stałe referencje są bardziej uniwersalne
- Stałe referencje ułatwiają kontrolę poprawności programu (gwarancja stałości argumentu)

# Przez referencję vs. przez wartość

```
void f(X & n) {...} // przez referencję
void g(X const& n) {...} // przez stałą referencję
void h(X n) {...} // przez wartość

X x; // zmienna nazwana
f(x); // f ma dostęp do x i może zmienić jego stan
g(x); // g ma dostęp do x, ale nie może zmienić jego stanu.
h(x); // h ma dostęp do kopii x;
f(7); // błąd; nie istnieje referencja do literału 7
g(7); // g ma dostęp do tymczasowej kopii X(7) i nie może jej zmienić;
h(7); // h ma dostęp do obiektu tymczasowego X(7)
```

- Stałe referencje i wartość są bardziej uniwersalne, ale mogą być bardziej kosztowne od referencji



# Konstruktor/destruktor argumentu

```
void f(X & n) {...} // przez referencję
void g(X const& n) {...} // przez stałą referencję
void h(X n) {...} // przez wartość

X x; // zmienna nazwana, czyli posiadająca adres
f(x); // f ma bezpośredni dostęp do x przez jej adres
g(x); // g ma bezpośredni dostęp do x przez jej adres
h(x); // h działa na kopii x, utworzonej konstruktorem
f(7); // błąd; nie istnieje referencja do literału 7
g(7); // g działa na obiekcie tymczas. X(7), utworzonym konstruktorem
h(7); // h działa na obiekcie tymczas. X(7), utworzonym konstruktorem
```

na obiekcie  
nazwanym

na obiekcie **nienazwanym**

- Stałe referencje działają inaczej na obiektach nazwanych (jak ref.) i tymczasowych (jak wartość)

# Referencja & vs. &&

- Referencja & jest używana wyłącznie do obiektów, które mają „stały adres”, czyli pełnią funkcję (modyfikowalnych) **zmiennych**
- Referencja && jest używana przez kompilator do obsługi **obiektów tymczasowych**
- Referencja && umożliwia przeniesienie stanu obiektu zamiast tworzenia jego kopii

# **DEKLARACJA A DEFINICJA**

## Deklaracja

```
void f(int x);  
int g(int a, char** p);  
[typ nazwa(argumenty);]
```

- Można opuścić nazwy argumentów:

```
void f(int);  
int g(int, char**);
```

## Definicja

```
void f(int x)  
{  
    return x + 1;  
}
```

- Można opuścić nazwy nieużywanych argumentów:

```
void f(int x, int&)  
{  
    return x + 1;  
}
```

**WARTOŚĆ FUNKCJI**

# Wartość funkcji

Funkcja może zwrócić swoją wartość na wiele sposobów

- Przez wartość
- Przez wskaźnik (\*, const\*, \*const, const\*const)
- Przez referencję (&, const&)
- Przez referencję przenoszącą (&&) [C++11]
- Obowiązują dokładnie te same zasady, co przy argumentach funkcji

# Wartość funkcji

Przykłady:

- Przez wartość `double f(double x)`
- Przez wskaźnik `double* f(double x)`
- Przez referencję `double& f(double x)`
- Przez referencję przenoszącą `double&& f(double x)`

# void

- Słowo kluczowe **void** służy do wskazania, że funkcja nie zwraca wartości  
**void** f(double x)
- W C++ nie ma potrzeby używania **void** do wskazania, że funkcja nie pobiera argumentów  
int main()



**EFEKTY UBOCZNE**

# Efekt uboczny to...

- Dowolna interakcja funkcji ze swoim otoczeniem w inny sposób niż poprzez swoje argumenty lub wartość
- Przykłady: zmiana wartości zmiennej globalnej; wyświetlenie znaków na konsoli; wywołanie funkcji systemowej
- Bardzo utrudniają analizę poprawności kodu (ludziom) i jego optymalizację (kompilatorowi)

# Efekty uboczne...

- Mimo że występują dość często i są nie do uniknięcia, wystrzegaj się ich stosowania!

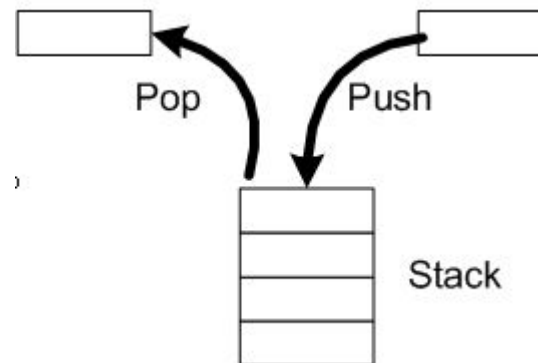
```
std::cout << "to jest efekt uboczny\n";  
file << "to nie jest efekt uboczny?\n";
```

- Nie wprowadzaj do systemu nowych efektów ubocznych
- Nie (nad)używaj własnych zmiennych globalnych

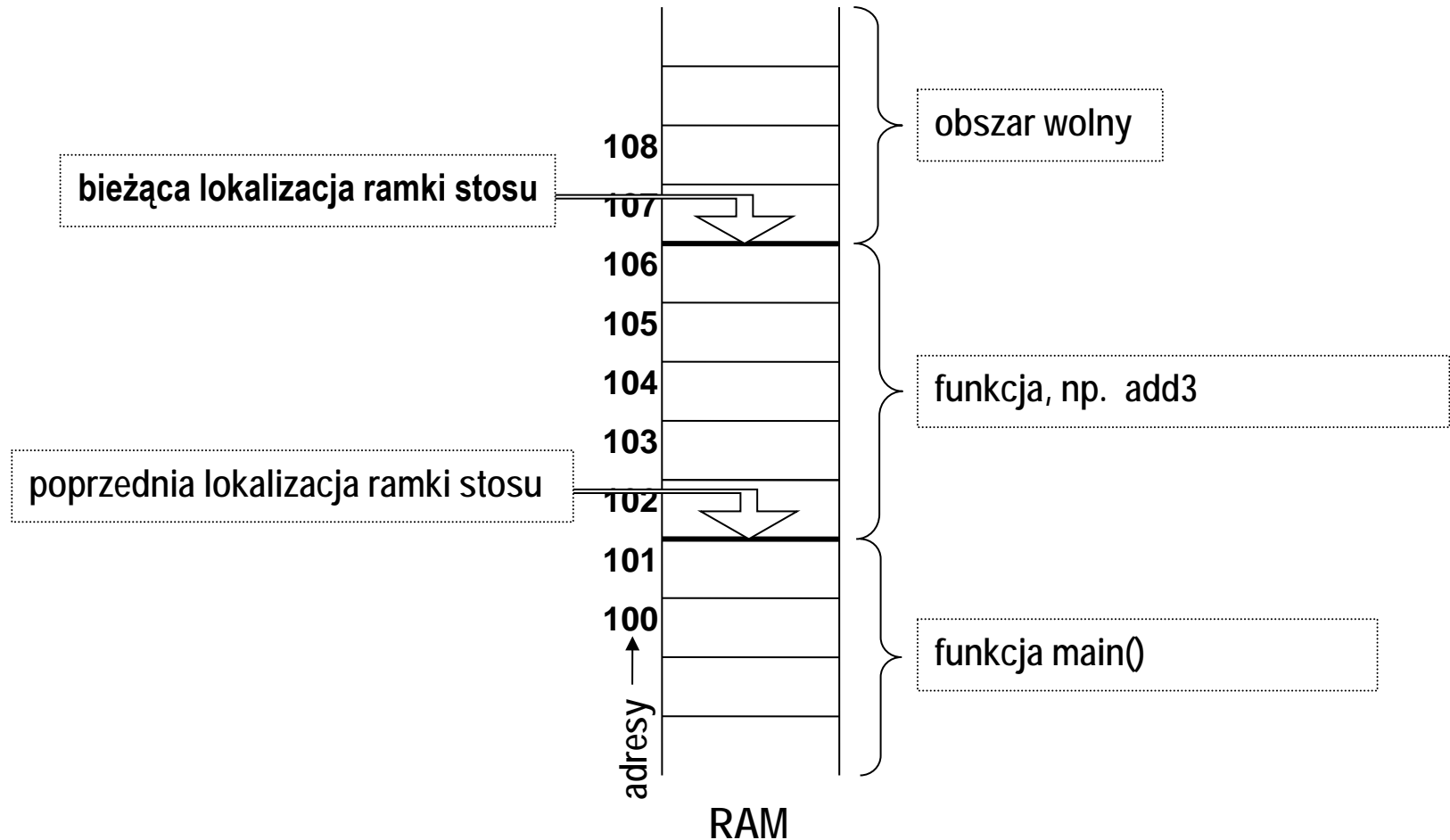
**STOS FUNKCJI**

# Stos funkcji (*call stack*)...

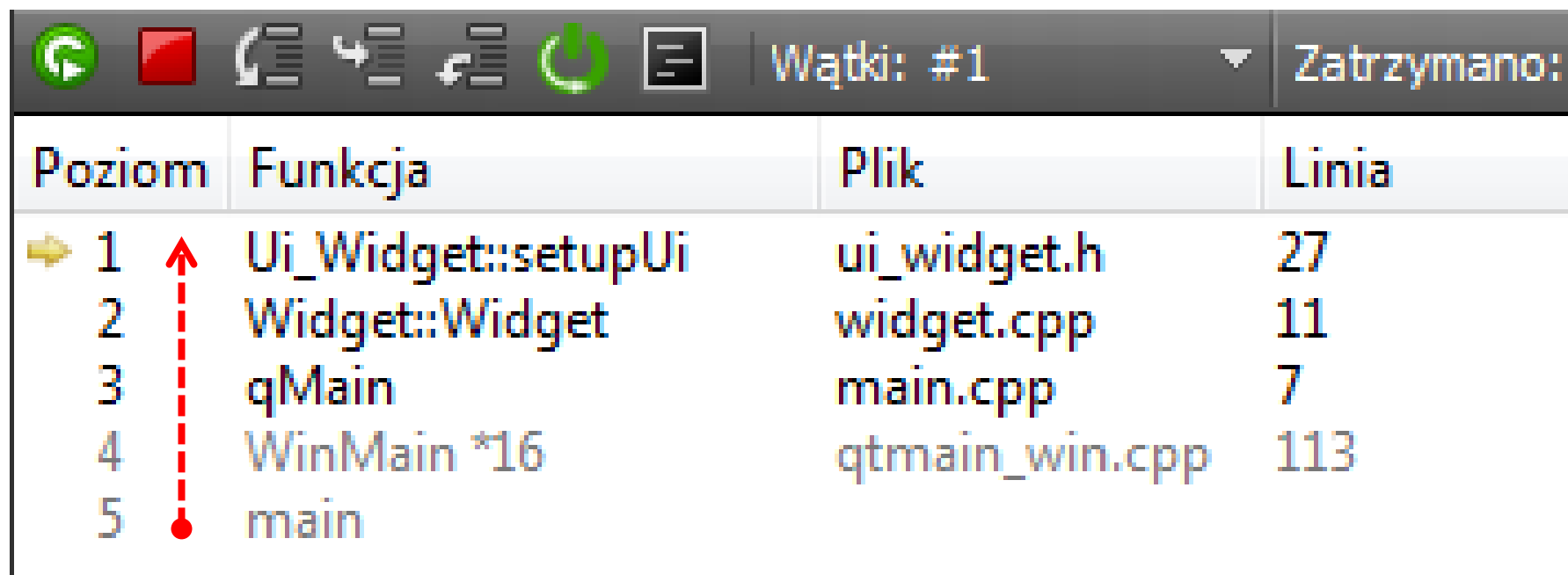
- Jest to specjalny obszar pamięci programu, w którym przechowywane są wszystkie dane potrzebne do wywołania i działania funkcji
- Stos działa na zasadzie kolejki **LIFO** (ang. *last in, first out*), czyli ostatni na wejściu jest pierwszy na wyjściu



# Z grubsza wygląda to tak...



albo tak (z perspektywy debugera):



Poziom	Funkcja	Plik	Linia
➡ 1	Ui_Widget::setupUi	ui_widget.h	27
2	Widget::Widget	widget.cpp	11
3	qMain	main.cpp	7
4	WinMain *16	qtmain_win.cpp	113
5	main		

# Przebieg ( $\pm$ ) wywołania funkcji

1. na stosie odkłada się adres powrotu z funkcji
2. rezerwuje się na nim miejsce na wartość funkcji
3. adres bieżącego szczytu stosu zapisywany jest w specjalnym rejestrze (*ramka stosu*)
4. na stosie umieszcza się argumenty funkcji
5. zmienia się adres wskaźnika instrukcji na adres kodu funkcji ( $\rightarrow$  skok do kodu funkcji)
6. funkcja na stosie umieszcza swoje zmienne lokalne, przesuwając wskaźnik szczytu stosu



# Jak przebiega ( $\pm$ ) kończenie funkcji

- funkcja zapisuje swoją wartość w miejscu zarezerwowanym w kroku 2
- następuje **zwijanie stosu** (wywoływanie destruktorów kolejnych zmiennych lokalnych) aż do osiągnięcia ramki stosu: „znikają” wszystkie zmienne lokalne i argumenty funkcji
- ze stosu odczytywane są wartości rejestrów, w tym adresu punktu wywołania funkcji
- następuje powrót do miejsca wywołania funkcji

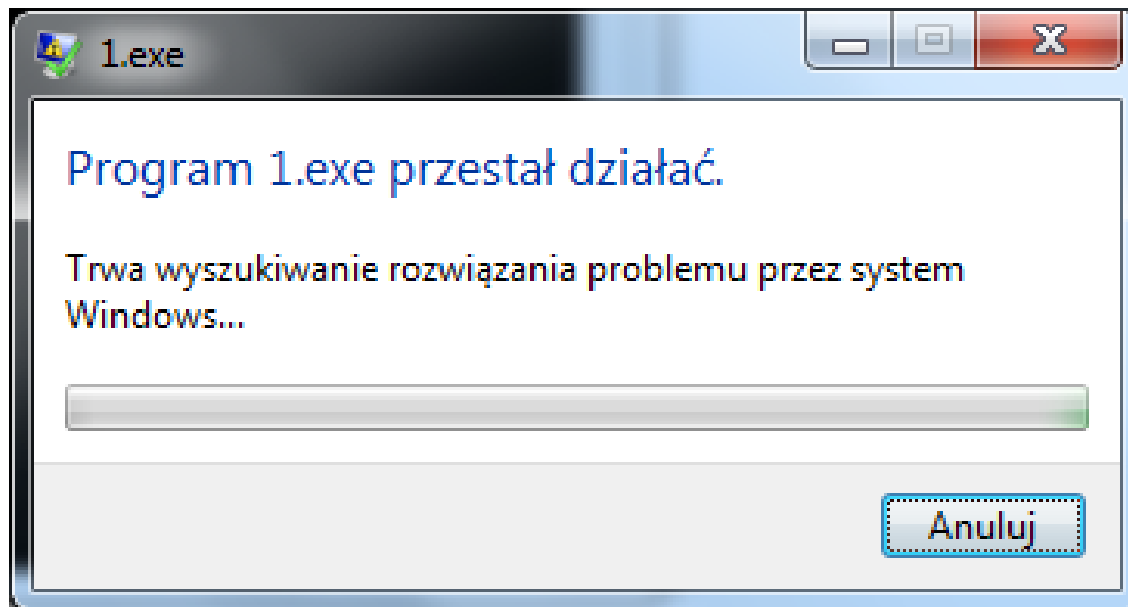
# Konsekwencje istnienia stosu funkcji

- Wszystkie zmienne definiowane w funkcji oraz jej argumenty są lokalne w funkcji i tracą ważność wraz z jej zakończeniem
- Rekurencyjne wywołanie funkcji przez samą siebie (***rekurencja***)  $\Rightarrow$  każde wywołanie ma własny obszar pamięci zmiennych lokalnych, wartości i argumentów funkcji
- Przekroczenie pojemności stosu prowadzi do padu programu

# Przykład

```
int main()  
{  
    const int N = 1000000;  
    int a[N];  
    a[0] = 0;  
}
```

← Zmienna na stosie



# Funkcja rekurencyjna

```
#include <iostream>
```

```
int silnia(int n)
{
    if (n == 0)
        return 1;
    return n * silnia(n - 1);
}
```

```
int main()
{
    std::cout << silnia(5) << "\n";
}
```

# Funkcje rekurencyjne...

- Swoje zmienne lokalne, argumenty i wartość przechowują na stosie funkcji
- Dzięki temu każde ich wywołanie ma osobny komplet zmiennych lokalnych
- Zbyt głęboka rekurencja powoduje pad programu (spróbuj jakąś funkcję wywołać rekurencyjnie kilkaset tysięcy razy)

# Zmienne statyczne

```
int f()  
{  
    static int licznik = 0;  
    return ++licznik;  
}
```

- Zmienne statyczne funkcji umieszczane są w innym obszarze pamięci niż stos funkcji: w obszarze zmiennych globalnych
- Dzięki temu funkcja przechowuje ich wartość pomiędzy swoimi kolejnymi wywołaniami.

# Przykład

```
#include <iostream>

void f()
{
    static bool first = true;
    if (first)
    {
        first = false;
        std::cout << "pierwsze wywołanie\n";
    }
}

int main()
{
    f(); f();
}

// pierwsze wywołanie
```

# Zmienne statyczne/ a zmienne automatyczne

- **Zmienne statyczne** inicjalizowane są **dokładnie raz** (**konstruktor!**) przy pierwszym wejściu programu do kodu inicjalizatora; **destrukторы** wywoływane są po zakończeniu funkcji main()
- **Zmienne automatyczne** inicjalizowane są **za każdym razem**, gdy program dochodzi do inicjalizatora (**konstruktor!**) i są niszczone (**destruktor!**) zawsze, gdy sterowanie opuszcza zakres, w którym zdefiniowano zmienną (ramka stosu przesuwana się w dół, zwijanie stosu)



# Przykład

```
for (int i = 0; i < 100000; i++)  
{  
    std::vector<int> v(2000);  
    static x x = 0;  
} ← — — — Koniec zakresu = częściowe zwinięcie stosu = destrukcja zmiennych lokalnych
```

- **Konstruktor** obiektu `v` wywoła się **100000** razy
- **Konstruktor** obiektu `x` wywoła się raz
- **Destruktor** obiektu `v` wywoła się **100000** razy
- **Destruktor** obiektu `x` wywoła się raz
- Zmiennych `x`, `v` nie można używać poza ich **zakresem** (ang. *scope*), czyli poza klamrami

**FUNKCJE SWOBODNE I SKŁADOWE**

# Funkcje swobodne a metody

- **Funkcje swobodne** to funkcje „w stylu języka C”, np.  
`double sin(double);`
- **Metody** to funkcje składowe klas
- Metody zawsze działają na jakieś **obiekty**
- Notacja z kropką: `v.size();`
- Notacja wskaźnikowa: `p->size();`

**this**

# Pseudowskażnik **this**

- **Metody klas** zawsze wywoływane są na rzecz obiektu zapisywanego przed kropką:

**v.size()**; // v jest argumentem size

- W powyższym przykładzie w ciele funkcji **size()** identyfikator **this** oznacza adres obiektu **v**; (**this == &v**)
- Wartość adresu obiektu (**this**) jest automatycznie odkładana na stosie funkcji wraz z argumentami metody (oszczędność notacji)

# Pseudowskażnik **this**

**p->size( ) ;**

- Teraz wewnątrz **size( )** pseudowskażnik **this** ma wartość **p**

# **this** a skrócona notacja

- **this** można używać tylko wewnątrz metod klas
- Wewnątrz metod klas zapis

**this->**funkcja(...) + **this->**skladowa

zwykle można uprościć do

**funkcja(...) + skladowa**

**FUNKCJE INLINE**



# Funkcje *inline* (otwarte, wklejane)

- Funkcje *inline* to funkcje, które nie posiadają własnej implementacji pod jednym, unikatowym adresem.
- Ich wywołaniu nie towarzyszy przeskok do osobnego kodu ich funkcji, bo takowego nie ma
- Zamiast tego kompilator wstawia ich kod (instrukcje) w każdym miejscu wywołania

# Funkcje *inline*

- Funkcje *inline* charakteryzują się krótszym czasem wywołania
- Częste używanie funkcji *inline* może doprowadzić do „napuchnięcia” kodu binarnego (ang. *code bloat*)
- Długi program zawsze jest powolny (*cache!*)
- Funkcje *inline* mogą więc przyspieszyć lub spowolnić działanie programu

# Przykład

```
#include<iostream>
inline
int f(int n)
{
    return n + 1;
}
int main()
{
    int n = 5;
    int k = f(n);
    std::cout << k;
}
```

# Przykład – c.d.

## Bez inline (Debug)

```
9      {...  
call  <__main>  
10     int n = 5;  
movl  $0x5,-0xc(%ebp)  
11     int k = f(n);  
mov   -0xc(%ebp),%eax  
mov   %eax,(%esp)  
call  <f(int)>  
mov   %eax,-0x10(%ebp)  
12     std::cout << k;
```

...

## Z inline (Release)

```
9      {...  
call  <__main>  
10     int n = 5;  
11     int k = f(n);  
12     std::cout << k;  
-----  
movl  $0x6,(%esp)  
mov   $0x489940,%ecx  
call  <std::ostream::operator<<(int)>  
sub   $0x4,%esp  
13     }
```

...

# Troszkę nakłamałem...

- Tak naprawdę na poprzednim slajdzie przedstawiłem kody asemblera wygenerowane w trybie Release i Debug.
- *Inlining* to jedna z metod **optymalizacji kodu** przez kompilator
- Zadeklarowanie funkcji jako *inline* gwarantuje jedynie to, że kompilator na pewno będzie w stanie dokonać tej optymalizacji; na to, jak i czy jej dokona nie mamy wpływu (!)

# `inline`

- Kompilator traktuje *inline* jako wskazówkę
  - W trybie Debug całkowicie wyłącza optymalizację typu *inlining*
  - W trybie Release kompilator sam decyduje, czy daną funkcję zoptymalizować w ten sposób – niezależnie od deklaracji *inline*
- Deklarator `inline` daje nam jednak pewność, że kompilator posiada techniczną możliwość dokonania *inliningu*.

# Kiedy stosować funkcje *inline*?

- Najlepiej tylko wtedy, gdy funkcje są bardzo krótkie (w sensie kodu maszynowego), wręcz trywialne, np.

```
size_t size() const { return _rozmiar; }
```

- Teraz

```
a = v.size();
```

kompilator skompiluje tak samo, jak

```
a = v._rozmiar;
```

jednak `v.size()` jest bardziej elastyczne

# Jak zadeklarować funkcje *inline*?

- Używając słowa kluczowego **inline**:

```
inline void f()  
{  
...  
}
```

- **Definiując funkcję składową w deklaracji jej klasy:**

```
class Wektor  
{  
...  
    size_t size() const { return _rozmiar; }  
};
```

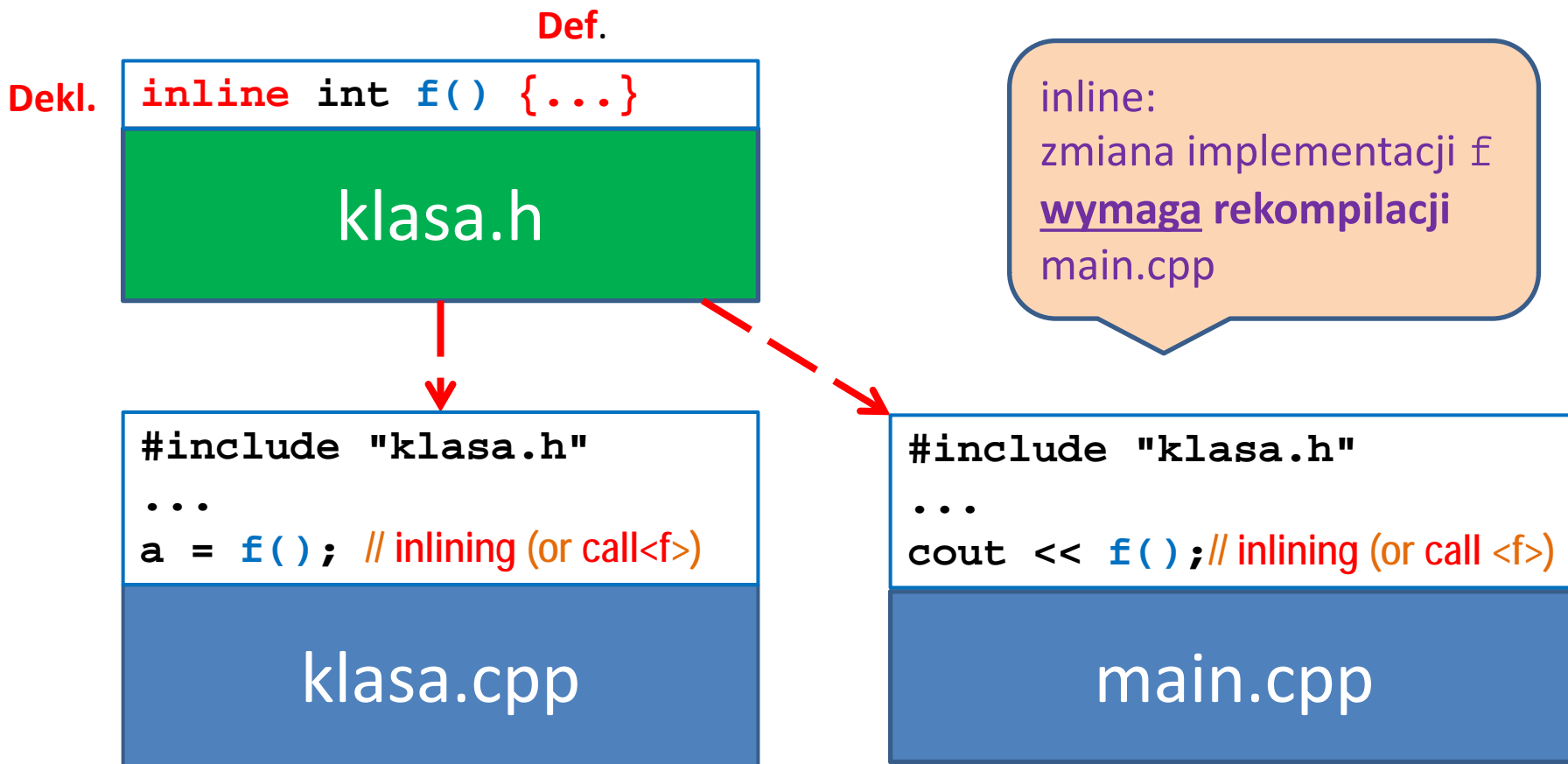
**size()** domyślnie otrzymuje atrybut inline



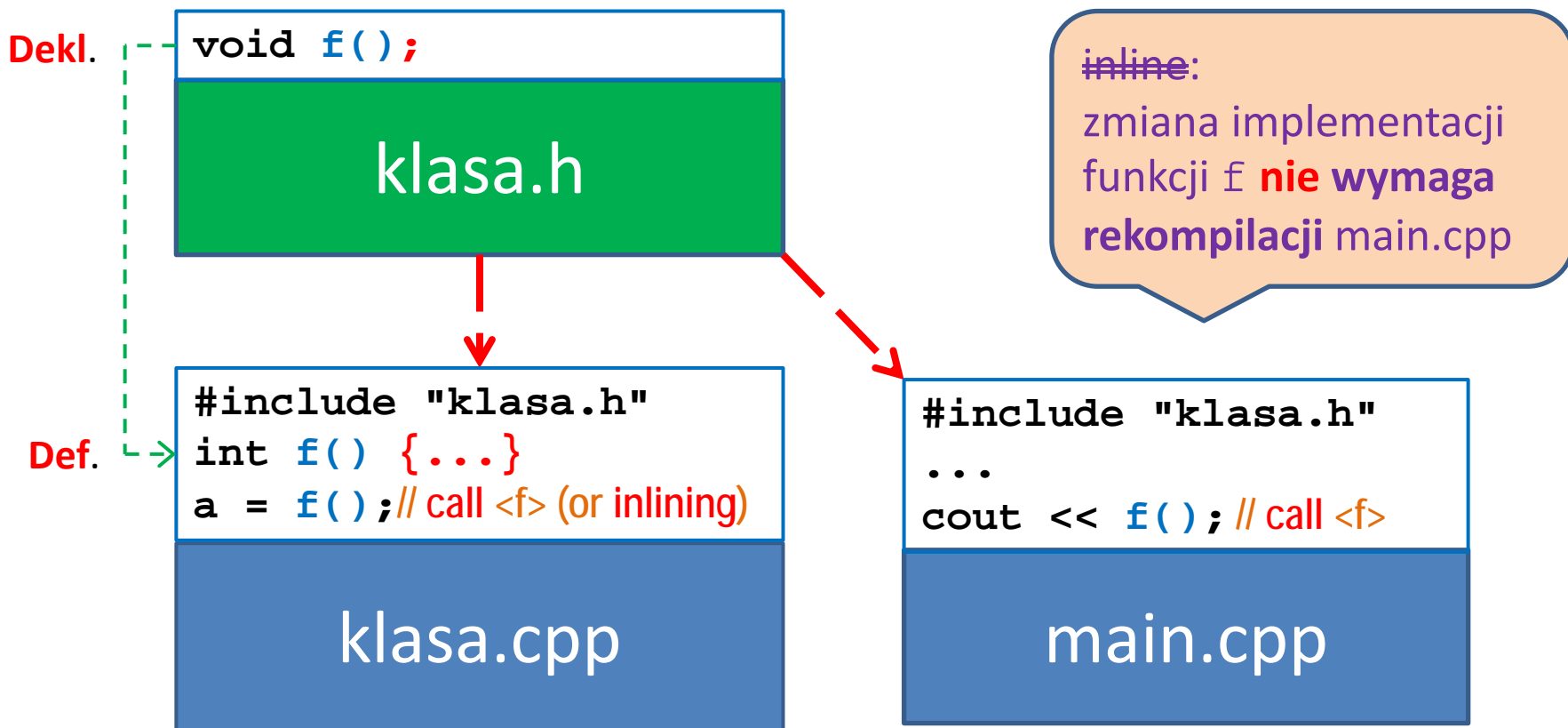
# Gdzie deklarować funkcje *inline*?

- Aby kompilator mógł zastosować *inlining*, musi mieć dostęp do kodu źródłowego funkcji
- Musi też mieć pewność, że wszystkie jednostki kompilacji używają tej samej funkcji inline
- Dlatego naturalnym miejscem na definiowanie funkcji *inline* są ***pliki nagłówkowe***
- Wszystkie pozostałe funkcje definiuje się (= podaje kod C++) w ***plikach źródłowych***

Kod funkcji *inline* **musi** być dostępny w każdej jednostce kompilacji (= \*.cpp), inaczej kompilator nie będzie mógł zastosować *inliningu*



Kod **zwykłej funkcji** **nie powinien** być dostępny w każdej jednostce kompilacji (= \*.cpp), bo inaczej pojęcie jednostki kompilacji straciłoby sens



# `inline` a biblioteki

- Biblioteki zawierające funkcje `inline` są ***bibliotekami czasu kompilacji*** (program użytkownika wkompilowuje funkcje biblioteki w swój kod)
  - Przykład: biblioteki generyczne, w tym `std::vector`
- Biblioteki niezawierające funkcji `inline` są ***zwykłymi bibliotekami*** (program użytkownika tylko dołącza do siebie skompilowany kod binarny biblioteki poprzez mechanizm "call")
  - Przykład: biblioteki własnościowe (dużo \$\$\$)

# **inline**: podsumowanie

- **inline** jest jedną z wielu cech języka C++, której jedynym celem jest ułatwienie optymalizacji kodu na poziomie kompilatora
- Posługiwanie się funkcjami **inline** nie jest trudne dla programisty: definiuj je w plikach nagłówkowych, a zwykłe funkcje – w plikach źródłowych, a w 99.999% przypadków wszystko będzie OK.
- W małych projektach są całkowicie zbędne (!); w dużych są nieocenione

**STATIC**

# static ma cztery (!) znaczenia...

- Dwa zostały odziedziczone z języka C

```
#include <iostream>

static int f()
{
    static int x = 0;
    x++;
    return x;
}

int main()
{
    for (int i = 0; i < 3; i++)
        std::cout << f() << "\n";
}
```

**static (function):** funkcja prywatna dla pliku.  
Nie używaj w C++

**static (data):** prywatna pamięć dla funkcji  
(por. poprzednie slajdy o stosie funkcji)  
dość użyteczny "wytrych".



```
zkoza@zbyszek: ./a.out
1
2
3
```

# static ma cztery (!) znaczenia...

- Dwa zostały dodane do języka C++

```
#include <iostream>

class X
{
    static int licznik;
public:
    X() { licznik++; }
    ~X(){ licznik--; }
    static int ile() { return licznik; }
};

int X::licznik = 0; // inicjalizator

int main()
{
    std::cout << X::ile() << "\n";
    X tab[4];
    std::cout << tab[0].ile() << "\n";
}
```

**static (data member):** pamięć wspólna dla wszystkich funkcji klasy

**static (method):** funkcja udostępniająca składowe statyczne

**inicjalizator** zawsze poza klasą!

użycie: **KLASA::funkcja(...)**

lub: **obiekt.funkcja(...)**



```
zkoza@zbyszek: ./a.out
0
4
```



# 4 znaczenia `static`

gdy `static` modyfikuje `dane`

gdy `static` modyfikuje `funkcję`

Jako **składowa klasy** (konstrukcje typowe dla **C++**)

- **alokacja**: w segmencie danych globalnych
- **konstrukcja**: jednorazowa
- **rola**: **pamięć klasy**
- **widoczność**: zgodnie z regułami C++ (public/protected/private)
- **składnia**: `X::dane` lub `x.dane` lub `p->dane`

- **rola 1**: **uchwyt do składowych** (zmiennych) **statycznych**
- **rola 2**: funkcja swobodna w przestrzeni nazw klasy
- **widoczność**: zgodnie z regułami C++ (public/protected/private)
- **składnia**: `X::f(...)` lub `x.f(...)` lub `p->f(...)`

Jako **zmienna lokalna** lub **funkcja swobodna** (jak w języku **C**)

- **alokacja**: w segmencie danych globalnych
- **konstrukcja**: jednorazowa
- **rola**: **pamięć funkcji**
- **widoczność**: wewnątrz jednej funkcji
- **składnia**: `x`

- **rola**: **funkcja lokalna** w jednym pliku (**jednostce kompilacji**);  
w C++11: *deprecated* ( $\Rightarrow$  namespace)
- **widoczność**: w danej jednostce kompilacji
- **składnia**: `f(...)`

# Statyczne funkcje składowe klas...

- Są to funkcje swobodne umieszczone w przestrzeni nazw danej klasy
- Nie są wywoływane na rzecz obiektów
- Nie mają dostępu do pseudoskażnika `this`
- Składnia wywołania:

```
obiekt.f( ) ; // obiekt nie jest używany  
klasa::f( ) ;
```

# Przykład użycia (Qt)

```
long a = 63;  
QString s = QString::number(a, 16); // s == "3f"
```

Można też użyć obiektu i notacji z kropką,  
ale taki zapis może być nieco mylący (co tu robi s?):

```
QString t = s.number(a, 16; // t == "3f"
```

# Statyczne składowe klas (dane)

- Są to **zmienne** statyczne (czyli umieszczane w segmencie zmiennych globalnych) **wspólne dla wszystkich obiektów danej klasy**.
- Można np. zliczać liczbę aktywnych obiektów danej klasy

# Statyczne składowe klas – przykład

```
std::string s = "Ala ma kota";  
s = s.substr(4, s.npos);    // s = "ma kota"
```

- Tu **npos** oznacza statyczną stałą reprezentującą „nieskończoność” w operacjach na napisach
- Alternatywny zapis:

```
s = s.substr(4, std::string::npos);
```

- Można zamiennie używać notacji

```
obiekt.składowa
```

```
klasa::składowa
```

# Statyczne funkcje swobodne...

- Poza nazwą nie mają niczego wspólnego z metodami statycznymi klas ☹
- Funkcja statyczna jest lokalna w jednostce kompilacji; w praktyce – jest **lokalna** („**prywatna**”) w danym pliku źródłowym.
- Bardzo rzadko używane
- W C++11 zastępowane przez umieszczenie funkcji w tzw. nienazwanej przestrzeni nazw

# Przykłady w dokumentacji

- Qt

**[static]** **QString** QString::number(long *n*, int *base* = 10)

Returns a string equivalent of the number *n* according to the specified *base*.

The base is 10 by default and must be between 2 and 36. For bases other than 10, *n* is treated as an unsigned integer.

The formatting always uses **QLocale::C**, i.e., English/UnitedStates. To get a localized string representation of a **number**, use **QLocale::toString()** with the appropriate locale.

```
long a = 63;
QString s = QString::number(a, 16);           // s == "3f"
QString t = QString::number(a, 16).toUpper(); // t == "3F"
```

See also **setNum()**.

# Przykłady w dokumentacji

- [cplusplus.com](http://cplusplus.com)

public static member constant

std::string::npos

<string>

static const size\_t npos = -1;

**Maximum value for size\_t**

npos is a static member constant value with the greatest possible value for an element of type size\_t.

This value, when used as the value for a *len* (or *sublen*) parameter in `string`'s member functions, means *"until the end of the string"*.

As a return value, it is usually used to indicate no matches.

This constant is defined with a value of -1, which because `size_t` is an unsigned integral type, it is the largest possible representable value for this type.



**ARGUMENTY DOMYŚLNE**

# Argumenty domyślne...

- to sposób na eliminację powtarzania tego samego kodu.

```
int f(int n, int base = 16) { ... }
```

```
...
```

```
f(10, 20);
```

```
f(10); // równoważne: f(10, 16)
```



f udaje funkcję jednowartościową

# Argumenty domyślne...

- Wartości domyślne deklaruje w ***pliku nagłówkowym*** a nie źródłowym!
- Uwaga: zmiana wartości parametru domyślnego funkcji wymaga rekompilacji wszystkich plików \*.cpp, które z tej funkcji korzystają.

**OPERATORY**

# Operators...

- Służą uproszczeniu zapisu, np.

```
std::cout << 7;      x = y + z[3];
```

- W C++ można definiować znaczenie większości operatorów w wyrażeniach zawierających obiekty klas użytkownika.
- Składnia operatorowa jest równoważna funkcyjnej, jeśli za nazwy funkcji przyjmie się zapis `operatorX`, gdzie `X` = `+`, `-`, `*`, `/`, `%`, `=`, `<<`, `>>`, `*`, `->`, `new`, `delete`, etc.

# Alternatywna notacja

```
c = a + b[0] ;
```

to inaczej

```
operator=(c, operator+(a, operator[](b,0)));
```

# składnia

```
class X {  
public:  
    int operator[] (int n) const { return p[n]; }  
    int operator() (int n, int k) const { return std::pow(p[n], k); }  
    operator double() const { return p[0]; }  
    double* p;  
    ...  
};  
X x;  
x[9];      // x.p[9];  
x(2, 3);   // std::pow(x.p[2], 3) ;  
double(x); // x.p[0];
```

funkcję można wywoływać  
na obiektach stałych

# Popularne operatory

- `X::operator= (X const &)` // kopiowanie
- `X::operator= (X &&)` // C++11; przesuwanie
- `X::operator()` // C++11:  $\Rightarrow$  funkcje lambda
- `operator<<, operator>>` // serializacja
- `X::operator[]` // indeksowanie

Przeciążanie innych operatorów to [Level Five](#).



# Metody generowane automatycznie

<i>lp.</i>	<i>nazwa</i>	<i>sygnatura dla klasy X</i>	<i>uwagi</i>
1	konstruktor bezargumentowy	<code>X::X( )</code>	
2	konstruktor kopiujący	<code>X::X( const&amp; )</code>	
3	konstruktor przesuwający	<code>X::X( &amp;&amp; ) noexcept</code>	C++11
4	operator= kopiujący	<code>X&amp; X::operator=( const&amp; )</code>	
5	operator= przesuwający	<code>X&amp; X::operator=( &amp;&amp; ) noexcept</code>	C++11
6	destruktor	<code>~X::X( )</code>	

- Reguły określające, kiedy każda z tych funkcji jest generowana, są dość złożone
- Dość częsty powód komunikatów diagnostycznych kompilatora

**FUNKCJE WIRTUALNE**

# Funkcje wirtualne...

- Omawialiśmy na osobnych zajęciach
- Ich idea polega na tym, że to nie kompilator, a program decyduje, która z wielu funkcji o tej samej sygnaturze zostanie wywołana na rzecz danego obiektu.
- Zwie się to ***łączeniem dynamicznym***: statyczna (= podczas kompilacji) analiza kodu binarnego nie pozwala przewidzieć, która z kilku funkcji zostanie w danym miejscu użyta

**POLIMORFIZM**

# Polimorfizm oznacza...

- Łączenie dynamiczne funkcji wirtualnych (nie wiadomo a priori, która z nich będzie użyta)
- Możliwość definiowania wielu funkcji o identycznych nazwach
  - Muszą się różnić liczbą argumentów, sposobem przekazywania argumentów lub modyfikatorem `const`

**CONST**

# Metody stałe (const)

```
int vector::size() const {return _size;}
```

- deklaratorem **const** to nasze zobowiązanie, że funkcja składowa klasy nie będzie modyfikować obiektów, na rzecz których jest wywoływana

```
void f(vector const& v)
{
    int n = v.size(); // ok, size() nie zmieni v
}
```

- Tylko amatorzy nie używają **const**!  
(bez nich trudno użyć **const&** )

**MENAŽERIA**



# explicit

```
void f (std::string const& s) {...}  
std::string s ("Ala"); // konstruktor  
f(s);                // ok  
f("0la");           // pożądana konwersja
```

Ale:

```
void g (vector<int> const& v) {...}  
vector<int> w(5); // konstruktor  
g(w);            // ok  
g(5);            // niepożądana konwersja
```

# explicit

- Każdy konstruktor jednoargumentowy

```
X::X(Y const& y)
```

jest jednocześnie operatorem *niejawnej*  
konwersji z Y do X

```
g(w) ; // ok
```

```
g(5) ; // niepożądana konwersja 5 ⇒ vector<int>(5)
```

# explicit

```
class X
{
    explicit X(Y const& y) {...}
    ...
};
```

- `explicit` wyłącza niejawną konwersję

```
    g(w);           // ok
    g(5);           // teraz błąd!
    g(vector<int>(5)); // ok, jawna konwersja
```
- eliminuje subtelne błędy
- tylko do konstruktorów z jednym argumentem

# =delete (C++11)

```
class X {  
    // ...  
    X& operator=(X const&) = delete;  
    X(X const&) = delete;  
};
```

- modyfikator **=delete** zakazuje kompilatorowi generowania kodu oznaczonej tak funkcji
- Zastosowanie: uniemożliwienie kopiowania strumieni

# `=default` (C++11)

```
class X {  
    // ...  
    X& operator=(X const&) = default;  
    X(X const&) = default;  
};
```

- modyfikator `=default` nakazuje kompilatorowi wygenerować domyślny kod danej funkcji
- informacja dla innych programistów, że jesteśmy świadomi tego, co robimy

= 0

```
class X {  
    // ...  
    virtual void f() = 0; // pure virtual  
};
```

- modyfikator **=0** zakazuje możliwości tworzenia obiektów klasy, w której w ten sposób zadeklarowano *czystą funkcję wirtualną*
- Użyteczna klasa musi być wyprowadzona z X przez *dziedziczenie*, a wszystkie jej czyste funkcje wirtualne – przeciążone

# override (C++11)

```
class Baza
{
    virtual int A(float);
    virtual int B() const;
    virtual int C();
    int D();
};

class Moja: public Baza
{
    virtual int A(int) override; // error: Moja::A(int) nie przeciąża Baza::A(float)
    virtual int B() override; // error: Moja::B() nie przeciąża Baza::B() const
    virtual int C() override; // ok! Moja::C() przeciąża Baza::C()
    int D() override; // error: Moja::D() nie jest wirtualna
};
```

- **override** ułatwia unikać subtelnych błędów

# noexcept

```
void f() noexcept; // f() nie zgłasza wyjątków
```

```
void g() noexcept(false); // g() może zgłosić wyjątek
```

- Deklarator **noexcept** (C++11) ułatwia kompilatorowi optymalizację kodu



**INNE...**

Funktory,  
wyrażenia lambda,  
wskaźniki na funkcje,  
sparametryzowane makra...

- O tych cudёнkach porozmawiamy przy innej okazji