Języki programowania

Nowoczesne techniki programowania

Wykład 5

Witold Dyrka
witold.dyrka@pwr.wroc.pl

9/11/2012

Prawa autorskie

Slajdy do wykładu powstały
w oparciu o slajdy Bjarne Stroustrupa
do kursu Foundations of Engineering II (C++)
prowadzonego w Texas A&M University

http://www.stroustrup.com/Programming

Kolokwium

- I termin
  - piątek, 30/11/2012 (w terminie wykładu)
- II termin
  - środa, 5/12/2012, godz. 7.30 s.314/A-1

Program wykładów

1. Pierwszy program 5/10
2. Obiekty, typy i wartości. Wykonywanie obliczeń 12/10
3. Błędy. 19/10
4. Tworzenie oprogramowania 26/10
5. **Techniki pisania funkcji i klas 9/11**
6. Strumienie wejścia/wyjścia 16/11
7. Wskaźniki i tablice 23/11

Zagadnienia (1)

- O szczegółach technicznych języka
- Deklaracje
  - Definicje
  - Nagłówki i preprocesor
  - Zakres
- Funkcje
  - Deklaracje i definicje
  - Argumenty
- Przekazanie argumentu przez wartość, referencję i stałą referencję
- Przestrzenie nazw
  - Instrukcje **using**

Czy musimy zajmować się szczegółami technicznymi języka?

- Niestety tak!
  - Język programowania jest językiem obcym
  - Musimy poznać jego gramatykę i słownictwo
- Dlaczego?
  - Programy muszą być precyzyjnie i kompletnie określone
    - Jak pamiętamy, komputer jest bardzo głupią (choć bardzo szybką) maszyną
    - Nie zgadnie co naprawdę chcesz powiedzieć (i nie powinien próbować)
  - A więc to my musimy znać zasady
    - No, przynajmniej niektóre z nich (cały standard C++11 liczy 1338 stron:-)
- Jednak pamiętajmy, że
  - Uczymy się programowania
  - Tworzymy programy lub systemy
  - Język programowania jest tylko narzędziem

Szczegóły techniczne

- Nie roztrząsaj mało istotnych drobnych zagadnień składniowych i znaczeniowych
  - zwykle jest więcej niż jeden sposób by coś napisać
    - Tak jak w języku polskim czy angielskim
- Większość pojęć dotyczących projektowania i programowania jest uniwersalna
  - to czego nauczysz się używając C++ wykorzystasz w wielu innych językach
- Szczegóły techniczne języka są specyficzne dla niego
  - ale wiele szczegółów C++, którymi zajmujemy się teraz ma odpowiedniki w C, Javie, C# itp.

Deklaracje

- Deklaracja wprowadza nazwę do zakresu
- Deklaracja określa typ nazwanego obiektu
- Czasami deklaracja zawiera inicjator
- Każda nazwa musi być zadeklarowana zanim będzie użyta w programie C++
- Przykłady:

```
int a = 7;                 // deklaracja zmiennej całkowitoliczbowej a

const double cd = 8.7;     // deklaracja stałej zmiennoprzecinkowej
                           // podwójnej precyzji o nazwie cd

double sqrt(double);       // deklaracja funkcji o nazwie sqrt pobierającej
                           // zmiennoprzecinkowy argument
                           // i zwracającej zmiennoprzecinkowy wynik

vector<Token> v;           // deklaracja zmiennej v będącej wektorem zmiennych
                           // typu Token
```

Definicje

- Deklaracja, która w pełni określa deklarowany obiekt nazywa się definicją
  - Przykłady definicji

```
int a = 7;      // definicja zmiennej całkowitoliczbowej a o wartości początkowej 7
vector<double> v;     // definicja pustego wektora liczb zmiennoprzecinkowych
double sqrt(double a) { … };  // definicja funkcji: ciało funkcji w miejscu …
struct Point { int x; int y; };    // definicja struktury Point o składowych x i y
```

- Przykłady deklaracji, które nie są definicjami
  - Tylko informują jak używać obiektów

```
double sqrt(double);        // ciało klasy nieokreślone
struct Point;               // składowe struktury nieokreślone
extern int a;               // słowo kluczowe extern oznacza, że nie jest to definicja
                            // extern jest archaiczne, rzadko się przydaje
```

Deklaracje a definicje

- Nie możesz zdefiniować czegoś dwa razy
  - definicja mówi, czym coś jest:

```
int a;      // definicja
int a;      // błąd: podwójna definicja
double sqrt(double d) { … }          // definicja
double sqrt(double d) { … }          // błąd: podwójna definicja
```

- Deklarować możesz wielokrotnie
  - deklaracja mówi, jak coś może być użyte;

```
int a = 7;       // definicja jest także deklaracją
extern int a; // deklaracja, że gdzieś została zdefiniowana zmienna a typu int
double sqrt(double d) { … }          // definicja funkcji jest także deklaracją
double sqrt(double);        // deklaracja funkcji, czyli jej prototyp albo interfejs
```

Po co osobno deklarować?

- **Żeby odnieść się do czego, wystarczy sama deklaracja**
- **Często chcemy zdefiniować to gdzie indziej**
  - dalej w pliku
  - w innym pliku
    - najlepiej napisanym przez kogoś innego
- **Deklaracja określa interfejs, czyli sposób użycia**
  - Twojego kodu
  - bibliotek
    - tu jest klucz: nie możemy ani chcemy pisać sami wszystkiego
- **W większych programach**
  - deklaracje umieszcza się w plikach nagłówkowych,
    aby ułatwić dostęp do nich

Nagłówki

- Plik nagłówkowy („nagłówek”, ang. *header*) zawiera deklaracje komponentów programu
  - np. funkcji, typów, klasy, stałych, zmiennych
  - deklaracje określają interfejs, czyli sposób użycia
- Nagłówek daje dostęp do funkcji, typów itp., które chcesz użyć w programie
  - są one zdefiniowane w swoich plikach źródłowych
    - często jako część biblioteki
  - przeważnie nie interesuje Cię jak są napisane
- To rozwiązanie wspiera technikę programowania nazwaną abstrakcją
  - nie musisz znać szczegółów implementacji funkcji takiej jak **cout**, aby jej używać. Kiedy umieszczasz dyrektywę preprocesora:

    ```
    #include <iostream>
    ```

    deklaracje zawarte w bibliotece **iostream** są dołączane do programu

Nagłówki (2)

Nagłówek **token.h:**

```
// deklaracje:
class Token { … };
class Token_stream {
    Token get();
    …
};
…
```

Implementacja Tokenu **token.cpp:**

```
#include "token.h"
//definicje:
Token Token_stream::get()
{ /* … */ }
…
```

Program użytkownika **use.cpp:**

```
#include "token.h"
…
Token t = ts.get();
…
```

- Plik nagłówkowy (tutaj, **token.h**) określa interfejs pomiędzy kodem użytkownika (**use.cpp**) a kodem implementacji komponentu (**token.cpp**, przeważnie w bibliotece)
- Ta sama deklaracja **#include** w obu plikach **.cpp** (pliku z implementacją komponentu **Token** oraz pliku używającym komponent **Token**) ułatwia zachowanie spójności

Nagłówki (3)

Nagłówek **token.h:**

```
// deklaracje:
class Token { … };
class Token_stream {
    Token get();
    …
};
…
```

Implementa-cja Tokenu **token.cpp:**

```
#include "token.h"
//definicje:
Token Token_stream::get()
{ /* … */ }
…
```

Program użytkownika **use.cpp:**

```
#include "token.h"
…
Token t = ts.get();
…
```

- Uwaga! Zapis **#include "token.h"** oznacza, że kompilator szuka nagłówka biblioteki w tym samym katalogu co nasz plik, który dołącza bibliotekę.
- Zapis **#include <iostream>** oznacza, że kompilator szuka nagłówka biblioteki na ścieżce przeszukiwania kompilatora
    - Jeśli biblioteka jest częścią biblioteki standardowej, pomijamy rozszerzenie .h

Zakres

- Zakres jest fragmentem tekstu programu, np.
  - Zakres globalny
  - Zakres klasy (wewnątrz klasy lub struktury)
  - Zakres lokalny (blok kodu pomiędzy nawiasami { i })
  - Zakres instrukcji (np. wnętrze instrukcji for)
- Nazwa zadeklarowana w danym zakresie jest widoczna
  - wewnątrz tego zakresu
  - w zakresach zagnieżdżonych (podrzędnych) w tym zakresie
- Zakres zapewnia lokalność nazw
  - Zabezpiecza przed kolizją nazw zmiennych i funkcji używanych w różnych częściach programu
  - To ważne: prawdziwe programy składają się z tysięcy komponentów
  - Lokalność jest dobra – nazwy powinny być tak lokalne jak to tylko możliwe

Zakres – przykład

```
#include <cmath>                          // dołącz funkcje max and abs
// tu nie ma r, i, oraz v
class My_vector {
    vector<int> v;                        // v jest w zakresie klasy My_vector
public:
    int largest()                         // funkcja largest jest w zakresie klasy My_vector
    {
        int r = 0;                        // r jest w zakresie lokalnym funkcji largest()
        for (int i = 0; i<v.size(); ++i)  // i jest w zakresie instrukcji for
            r = max(r,abs(v[i]));
        // tutaj nie ma już i
        return r;
    }
    // tutaj nie ma już r
};
// tutaj nie ma już v
```

Zagnieżdżanie zakresów

```
int x = 100;        // zmienna globalna – unikaj, jeśli tylko możesz
int y = 200;        // inna zmienna globalna

int f()
{
    int x;          // zmienna lokalna (teraz mamy dwa x)
    x = 7;          // podstawienie lokalnej zmiennej x, a nie globalnej x
    {
        int x = y;       // kolejna lokalna x – zainicjowana globalną y
                         // (teraz mamy trzy x)
        x++;             // zwiększamy lokalną x w tym zakresie
    }
    return x;
}
```

// Uwaga! Unikaj takiego skomplikowanego zagnieżdżania zakresów i przesłaniania zmiennych – pisz proste programy!

Funkcje

- Ogólna postać:

  typ_zwracany nazwa (argumenty); *// deklaracja*
  typ_zwracany nazwa (argumenty) ciało_funkcji *// definicja*

  Na przykład:

  double f(int a, double d) { return a*d; }

- Argumenty funkcji nazywa się także parametrami
- By funkcja nie zwracała niczego, podaj **void** jako typ zwracany

  void zwieksz_moc(int poziom);

- **Ciało funkcji** to blok lub blok **try**, np.

  { /* kod programu */ } *// blok*

  try { /* kod programu */ } *// blok **try***
  catch(exception& e) { /* kod obsługi wyjątków */ } *//*

- Funkcje reprezentują/implementują przetwarzanie/obliczenia

Przekazywanie parametrów przez wartość

*// przekazywanie przez wartość – wysyłamy funkcji **f** kopię wartości argumentu*

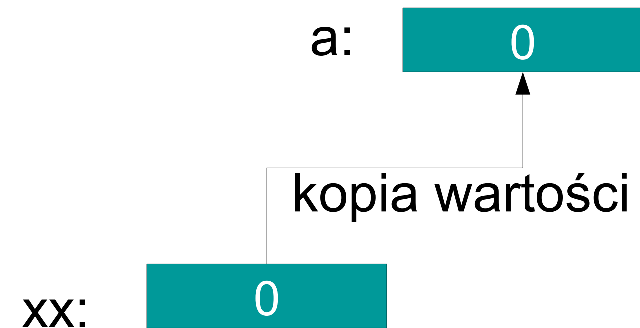
```
int f(int a) { a = a+1; return a; }


int main()

{

    int xx = 0;
    cout << f(xx) << endl; // wypisze 1
    cout << xx << endl;    // wypisze 0; f() nie zmieniła xx
    int yy = 7;
    cout << f(yy) << endl; // wypisze 8;  f() nie zmieniła yy
    cout << yy << endl;    // wypisze 7

}
```

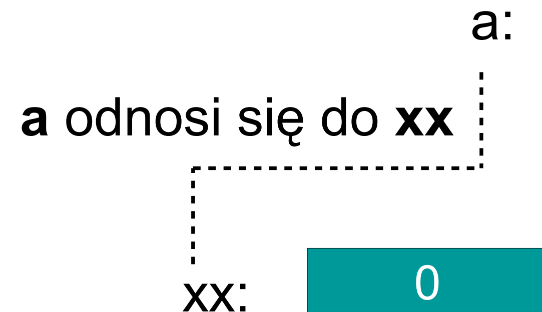


Przekazywanie parametrów przez referencję (czyli odniesienie)

*// przekazywanie przez referencję – przekazujemy funkcji **f** referencję do argumentu*

```
int f(int& a) { a = a+1; return a; }

int main()
{
    int xx = 0;
    cout << f(xx) << endl; // wypisze 1
    cout << xx << endl;    // wypisze 1; f() zmieniła wartość xx
    int yy = 7;
    cout << f(yy) << endl; // wypisze 8;  f() zmieniła wartość yy
    cout << yy << endl;    // wypisze 8
}
```



Przekazywanie przez referencję

- Niebezpieczne, prowadzi do trudnych do znalezienia błędów
  - kiedy zapomnimy, które argumenty mogą się zmienić

```
int zwieksz1(int a) { return a+1; }
void zwieksz2(int& a) { ++a; }
int x = 7;
x = zwieksz1(x); // to jest jasne – zwiekszylismy x
zwieksz2(x);     // to jest niejasny sposób na zrobienie tego samego
```

- Po co więc przekazywać parametry przez referencję?
  - Czasem jest to przydatne, np.
    - przy zmienianiu kilku wartości naraz
    - przy pracy z kontenerami (np. wektor)
  - bardzo przydatne jest przekazywanie przez stałą referencję

Przekazywanie parametrów przez stałą referencję

```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // błąd: cr jest stałe (const)
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;
    g(x,y,z);       // x==0; y==1; z==0
    g(1,2,3);       // błąd: argument referencyjny r musi odnosić się do zmiennej
    g(1,y,3);       // ok: ponieważ cr jest stałą (const) możemy przekazać literał
                    // literału nie można modyfikować – z definicji jest stałą
}
```

// ***stałe*** *referencje są bardzo przydatne do przekazywania dużych obiektów (oszczędzamy na czasie wykonywania kopii)*

Wskazówki jak przekazywać argumenty funkcji

- przez wartość – bardzo małe obiekty
  (np. typy wbudowane: **char, int, double**)
- przez stałą referencję – duże obiekty
- przez referencję – tylko jeśli musisz
  - lepiej zwrócić wynik i zmodyfikować obiekt niż modyfikować obiekt przekazując referencję

```
int zwieksz1(int a) { return a+1; }
void zwieksz2(int& a) { ++a; }
int x = 7;
x = zwieksz1(x); // to jest jasne – zwiekszylismy x
zwieksz2(x);     // to jest niejasny sposób na zrobienie tego samego
```

Referencje

- Referencja jest pojęciem ogólnym
  - służy nie tylko do przekazywania parametrów funkcji

```
int i = 7;
int& r = i;
r = 9;                  // i staje się 9
const int& cr = i;
// cr = 7;              // błąd: cr traktuje obiekt, do którego się odnosi, jak stałą
i = 8;
cout << cr << endl;     // wypisuje wartość i (tj. 8)
```

- Referencję można traktować jako przezwisko obiektu
- Nie da się przestawić raz zainicjowanej referencji na inny obiekt
  - zapis: **int a=1,b=2; int& r = a; r = b** byłby niejednoznaczny (sprawdź jak to działa?)

Kolizja nazw

- Piszemy program korzystając z klas opracowanych przez dwoje programistów – Jacka i Agatkę:

```
class Glob { /*...*/ };        // w nagłówku Jacka jacek.h
class Widget { /*...*/ };      // także w jacek.h
```

```
class Blob { /*...*/ };        // w nagłówku Agatki agatka.h
class Widget { /*...*/ };      // także w agatka.h
```

```
#include "jacek.h";        // to jest nasz kod, dołączamy nagłówki Jacka i Agatki
#include "agatka.h";       //

void my_func(Widget p)     // ups! – błąd: wielokrotna definicja klasy Widget
{
  // ...
}
```

Kolizja nazw (2)

- Kompilator nie poradzi sobie z wielokrotnie zdefiniowanymi komponentami
  - Takie konflikty nazw zdarzają się gdy korzystamy z wielu nagłówków
  - Możemy ich uniknąć stosując przestrzenie nazw:

```
namespace Jacek {            // w nagłówku Jacka jacek.h
  class Glob { /*...*/ };
  class Widget { /*...*/ };
}
```

```
#include "jacek.h";          // to jest nasz kod
#include "agatka.h";         // dołączamy nagłówki Jacka i Agatki

void my_func(Jacek::Widget p)     // ok! Widget Jacka nie pomyli się już
{                                 // z innymi Widgetami
  // ...
}
```

Przestrzeń nazw

- Przestrzeń nazw to nazwany zakres

```
namespace moj_zakres {
    int x = 0;
}
moj_zakres::x = 5;
```

- Używamy składni **::** aby określić, którą przestrzeń nazw mamy na myśli
  - Na przykład **cout** jest w przestrzeni nazw **std**, dlatego piszemy:

```
std::cout << "Proszę coś napisać… \n";
```

Deklaracje i dyrektywy **using**

- Ponieważ pisanie **std::cout** jest niewygodne można użyć deklaracji **using**

using std::cout; *// czytaj: kiedy piszę **cout**, chodzi mi o **std::cout***

cout << "Proszę coś napisać: \n" *// To zadziała, chodzi o **std:cout***

cin >> x; *// Ale to już nie, bo **cin** nie jest w zakresie*

- Dlatego wygodne może być zastosowanie dyrektywy **using**

using namespace std; *// czytaj: jeśli w bieżącym zakresie nie znajdziesz*
*// deklaracji nazwy, szukaj jej w przestrzeni nazw **std***

cout << "Proszę coś napisać: \n" *// To zadziała, chodzi o **std:cout***
cin >> x; *// To też, chodzi o **std:cin***

Zagadnienia (2)

- Klasy
  - Interfejs i implementacja
  - Konstruktory
  - Funkcje składowe
- Wyliczenia
- Przeładowywanie operatorów

Klasy

- Klasa bezpośrednio reprezentuje pojęcie w programie
  - Jeśli o „czymś” można myśleć jako o oddzielnym bycie, to prawdopodobnie może to być klasą lub obiektem klasy

    Np. wektor, macierz, strumień wejściowy, łańcuch znakowy, szybka transformata furiera, kontroler zaworu, ramię robota, sterownik urządzenia, obraz na ekranie, okno dialogowe, wykres, okno, odczyt temperatur, zegar
- Klasa jest typem zdefiniowanym przez użytkownika, który określa jak tworzyć i używać obiekty tego typu
- Klasy po raz pierwszy wprowadzono w języku Simula67

Składowe klasy i dostęp do nich

```
class X {               // ta klasa nazywa się X
   // dane składowe (przechowują informacje)
   // funkcje składowe (robią coś z danymi składowymi)
};
```

```
class X {
public:
   int m;      // dana składowa
   int mf(int v) { int old = m; m=v; return old; }   // funkcja składowa
};
```

```
X var;                      // zmienna var typu X
var.m = 7;                  // dostęp do danej składowej m zmiennej var
int x = var.mf(9);          // wywołanie funkcji składowej mf() dla zmiennej var
```

Interfejs i implementacja klasy

```
class X {          // klasa o nazwie X
public:            // publiczne składowe klasy – interfejs użytkownika
                   //     (dostępne dla wszystkich)
    // funkcje
    // typy
    // dane (najczęściej lepiej, żeby były prywatne)
private:           // priwatne składowe klasy – szczegóły implementacyjne
                   //     (dostępne tylko dla składowych klasy)
    // funkcje
    // typy
    // dane
 };
```

Struktury i klasy

- Składowe klasy są domyślnie prywatne, czyli zapis:

```
class X {
      int mf();
      // …
};
```

- oznacza tyle co:

```
class X {
private:
      int mf();
      // …
};
```

- Więc:

```
X x;                 // zmienna x typu X
int y = x.mf();      // błąd kompilacji: mf jest prywatna (niedostępna)
```

Struktury i klasy (2)

- Składowe struktury są domyślnie publiczne, czyli zapis:

```
struct X {
      int m;
      // …
};
```

- oznacza tyle co:

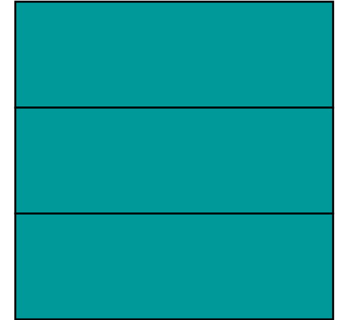
```
class X {
public:
      int m;
      // …
};
```

- Rodzi się pytanie – po co ukrywać składowe?

Po co komu prywatne składowe?

- Aby stworzyć przejrzysty interfejs klasy
  - dane i zagmatwane funkcje można ukryć
- **Aby chronić poprawność obiektu** (kontrolować niezmienniki)
  - tylko ograniczony zbiór funkcji ma dostęp do danych
- Aby ułatwić debugowanie
  - „zawężenie grona podejrzanych”
- Aby umożliwić zmianę reprezentacji (danych składowych)
  - wystarczy zmienić ograniczony zbiór funkcji
  - w przypadku składowej publicznej nie można nigdy wiedzieć, kto ją używa

Struktura – tylko dane
(jak w języku C)

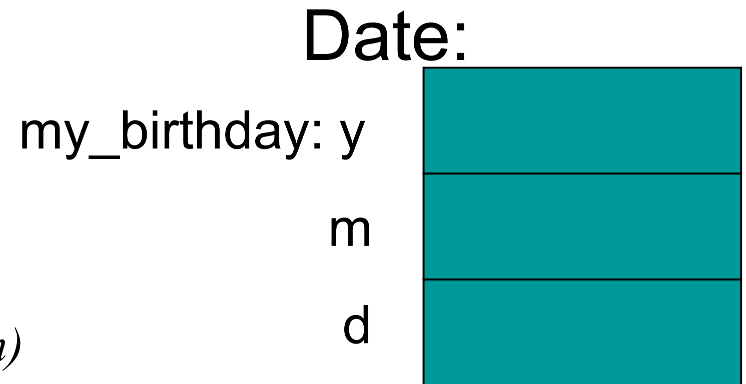



```
// najprostsza wersja Date (tylko dane)
struct Date {
    int y,m,d;      // rok, miesiąc, dzień
};

Date my_birthday;        // zmienna typu Date (obiekt)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950;    // Ups! (nie ma dnia 1950 w miesiącu 30)
                         // Błąd nie został zgłoszony, ale dalej w programie
                         // będziemy mieli problemy
```

Struktura – tylko dane
(jak w języku C) podejście 2




```
// prosta wersja Date (dodaliśmy kilka funkcji pomocnicznych)
struct Date {
    int y,m,d;        // rok, miesiąc, dzień
};

Date my_birthday;             // zmienna typu Date (obiekt)

// funkcje pomocnicze:

void init_day(Date& dd, int y, int m, int d);    // sprawdź poprawność daty i inicjalizuj
void add_day(Date&, int n);                      // zwiększ obiekt Date o n dni
// …

init_day(my_birthday, 12, 30, 1950);       // błąd czasu wykonania:
                                           // nie ma dnia 1950 w miesiącu 30
```

Świetnie, ale nie ma gwarancji, że użytkownik struktury użyje funkcji **init_day()** po utworzeniu zmiennej typu **Date**.

Struktura w stylu C++

Date:

my_birthday: y	1950
m	12
d	30

```
// prosta Date
//          gwarantuje inicjalizację przy użyciu konstruktora
//          zapewnia pewną wygodę notacyjną
struct Date {
    int y,m,d;                      // rok, miesiąc, dzień
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);        // zwiększa datę o n dni
};

// …
Date my_birthday;                   // błąd: my_birthday nie zainicjowane (I dobrze!)
Date my_birthday(12, 30, 1950);     // ups! Błąd czasu wykonania
Date my_day(1950, 12, 30);          // ok
my_day.add_day(2);                  // 1 stycznia 1951
my_day.m = 14;                      // grrrhh! (teraz my_day jest niepoprawną datą)
```

Świetnie, poprawna inicjacja jest zagwarantowana. Niestety, publiczny dostęp do danych składowych **Date** grozi "zepsuciem" daty

Struktura w stylu C++
(z kontrolą dostępu)

```
// prosta Date (z kontrolą dostępu)
struct Date {
private:
    int y,m,d;      // rok, miesiąc, dzień
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);      // zwiększa datę o n dni
};
```

Date:

my_birthday: y	1950
m	12
d	30

Klasa
(z kontrolą dostępu)

Date:

my_birthday: y	1950
m	12
d	30

```
// prosta Date (z kontrolą dostępu)
class Date {
    int y,m,d;        // rok, miesiąc, dzień
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);        // zwiększa datę o n dni
};


// …
Date my_birthday(1950, 12, 30);     // ok
my_birthday.m = 14;                 // błąd: składowa Date::m jest prywatna
cout << my_birthday.m   ;           // błąd: składowa Date::m jest prywatna
```

Funkcje dostępowe

Date:

my_birthday: y	1950
m	12
d	30

```
// prosta Date (z kontrolą dostępu)
class Date {
    int y,m,d;        // rok, miesiąc, dzień
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje

    void add_day(int n);      // zwiększa datę o n dni
    // funkcje dostępowe:
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};

// …
Date my_birthday(1950, 12, 30);          // ok
my_birthday.m = 14;                       // błąd: składowa Date::m jest prywatna
cout << my_birthday.m                     // błąd: składowa Date::m jest prywatna

cout << my_birthday.month() << endl;      // możemy odczytać miesiąc
```

Niezmienniki

- Pojęcie **poprawnej daty** jest szczególnym przypadkiem pojęcia **poprawnej wartości**
- Zasada: zagwarantować poprawne wartości dla projektowanego typu (**poprawny stan obiektu**)
  - inaczej musielibyśmy cały czas sprawdzać poprawność danych lub liczyć, że zrobi to użytkownik klasy
- Regułę opisującą poprawną wartość nazywa się **niezmiennikiem**
  - w przypadku daty jest to wyjątkowo trudne (28 luty, rok przestępny itd.)
- Jeśli nie można znaleźć dobrego niezmiennika – pracujesz na samych „czystych” danych
  - użyj struktury
  - pierwszą opcją jest jednak zawsze szukanie niezmienników!

Prosty konstruktor klasy

Date:

my_birthday: y	1950
m	12
d	30

```
// prosta Date (niektórzy wolą interfejs klasy na początku – dlaczego?)
class Date {
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);        // zwiększa datę o n dni
    int month();
    // ...
private:
    int y,m,d;      // rok, miesiąc, dzień
};

Date::Date(int yy, int mm, int dd)          // definicja konstruktora klasy
    :y(yy), m(mm), d(dd) { /* ... */ };     // inicjacja składowych

void Date::add_day(int n) { /* ... */ };    // definicja funkcji
```

Konstruktor z kontrolą poprawności

```
// prosta Date (co zrobimy z nieprawidłową datą)
class Date {
public:
    class Invalid { };                  // użyta jako rakieta sygnalizacyjna
                                        // przy obsłudze wyjątków
    Date(int y, int m, int d);          // sprawdza poprawność daty i inicjuje
    // …
private:
    int y,m,d;                          // rok, miesiąc, dzień
    bool check(int y, int m, int d);    // czy (y,m,d) jest poprawną datą?
};

Date:: Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd)                   // inicjuje dane składowe
{
    if (!check(y,m,d)) throw Invalid();     // sprawdza poprawność daty
}
```

Konstruktor z kontrolą poprawności

```
// prosta Date (co zrobimy z nieprawidłową datą)
class Date {
public:


    Date(int y, int m, int d);          // sprawdza poprawność daty i inicjuje
    // …
private:
    int y,m,d;                          // rok, miesiąc, dzień
    bool check(int y, int m, int d);    // czy (y,m,d) jest poprawną datą?
};

Date:: Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd)                       // inicjuje dane składowe
{
    if (!check(y,m,d)) throw 1;                 // sprawdza poprawność daty
}
```

Obiekty stałe

```
class Date {
public:
    // ...
    int day() { return d; }
    void add_day(int n);
    // ...
};

Date d(2000, 1, 20);            // zmienna d typu Date
const Date cd(2001, 2, 21);     // stała cd typu Date

cout << d.day();                // ok
cout << cd.day();               // błąd: cd jest const – ale przecież day()
                                // nie modyfikuje obiektu!
d.add_day(1);     // ok
cd.add_day(1);    // błąd: cd jest const – tutaj błąd „ma sens”
```

Składniki stałe klasy (**const**)

```
class Date {
public:
    // ...
    int day() const { return d; }      // stały składnik: nie może modyfikować
    void add_day(int n);               // nie-stały składnik: może modyfikować
    // ...
};

Date d(2000, 1, 20);            // zmienna d typu Date
const Date cd(2001, 2, 21);     // stała cd typu Date

cout << d.day() << " – " << cd.day() << endl;      // ok
d.add_day(1);     // ok
cd.add_day(1);    // błąd: cd jest const
                  // Tylko stałe funkcje składowe są dopuszczane
                  // do działania na stałych obiektach klasy
```

Składniki stałe klasy (2)

```
//
Date d(2004, 1, 7);              // zmienna
const Date d2(2004, 2, 28);      // stała
d2 = d;          // błąd: d2 jest const
d2.add(1);       // błąd: d2 jest const
d = d2;          // w porządku
d.add(1);        // w porządku
```

Składniki stałe klasy (3)

```
// Rozróżnij pomiędzy funkcjami, które mogą modyfikować obiekty
// I tymi, które nie mogą – zwanymi stałymi funkcjami składowymi
class Date {
public:
    // …
    int day() const;          // pobierz dzień (w zasadzie jego kopię)
    // …
    void add_day(int n);  // przesuń datę o n do przodu
    // …
};

const Date dx(2008, 11, 4);
int d = dx.day();             // w porządku
dx.add_day(4);                // błąd: nie można modyfikować stałych danych
```

Dobry interfejs klasy

- Minimalny
  - tak mały jak to możliwe
- Kompletny
  - i nie mniejszy
- Bezpieczny dla typów (ang. *type-safe*)
  - uwaga na kolejność argumentów
- Poprawnie określać stałe składniki klasy (ang. *const-correct*)

Minimalny zestaw składowych

- Najbardziej podstawowe operacje
  - Konstruktor domyślny (**domyślnie:** nie robi nic lub brak domyślnego konstruktora jeśli zadeklarowano inny konstruktor
  - Konstruktor kopiujący (**domyślnie:** kopiuje składowe)
  - Operator przypisania (**domyślnie:** kopiuje składowe)
  - Destruktor (**domyślnie:** nie robi nic)
- Na przykład

```
Date d;          // błąd: brak domyślnego konstruktora (zadeklarowaliśmy inny!)
Date d2 = d;     // ok: inicjacja kopią (kopiuje elementy – domyślne)
d = d2;          // ok: przypisanie kopii (kopiuje elementy – domyślne)
```

Interfejs klasy i funkcje pomocnicze

- Chcemy minimalny interfejs klasy (zbiór funkcji pomocnicznych), bo to:
  - Ułatwia zrozumienie klasy
  - Upraszcza debugowanie
  - Upraszcza pielęgnację
- Potrzebujemy funkcji pomocniczych, operujących na obiektach klasy, ale zdefiniowanych poza klasą, np.
  - == (równość) , **!**= (nierówność)
  - **next_weekday()**, **next_Sunday()**

Funkcje pomocnicze

```
Date next_Sunday(const Date& d)
{
    // ma dostęp do obiektu d poprzez d.day(), d.month(), oraz d.year()
    // tworzy nowy obiekt typu Date który zwraca
}

Date next_weekday(const Date& d) { /* ... */ }

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b) { return !(a==b); }
```

Zasady tworzenia operatorów

- Można definiować działania tylko istniejących operatorów
  - np. + - * / % [] () ^ ! & < <= > >=
- Trzeba zachować konwencjonalną liczbę operandów
  - np., nie ma unarnego (jednoargumentowego) <= (mniejsze-równe) albo binarnego (dwuargumentowego) ! (negacja)
- Każdy utworzony („przeładowany”) operator musi mieć przynajmniej jeden operand typu użytkownika
  - **int operator+(int,int);** *// błąd: nie można przeładować* ***wbudowanego+***
  - **const Wielomian operator+(const Wielomian&, const Wielomian &);** *// ok*
  - **const Wielomian operator+(const Wielomian&, int);** *// ok*
- Dobre rady:
  - nadawaj operatorom tylko konwencjonalne znaczenie

    \+ powinnien być dodawaniem, * - mnożeniem, [] - dostępem, a () wywołaniem
  - twórz je tylko jeśli to naprawdę potrzebne

Dzisiaj najważniejsze było...

- Deklaracja a definicja
- Zakres
- Przekazywanie argumentów funkcji
  - przez wartość
  - przez (stałą) referencję

- Interfejs i implementacja klasy
- Niezmienniki

A za tydzień...

- Strumienie wejścia/wyjścia