# Dzisiejszy wykład

- Deklaracje i definicje klas w C++
- Składowe, pola, metody
- Konstruktory
- Wskaźnik *this*
- Destruktor
- Przeciążanie funkcji i operatorów
- Funkcje otwarte

# Interfejs prostej klasy *Date*

- Oto **deklaracja** (nie **definicja**) prostej klasy

```
class DateType {
public:
   // constructor
   DateType();
   DateType(int newMonth, int newDay, int newYear);
   // accessor methods (get methods)
   int GetYear( ) const;   // returns Year
   int GetMonth( ) const;  // returns Month
   int GetDay( ) const;    // returns Day
private:
   int Year;
   int Month;
   int Day;
};
```

- Deklaracja typu czy zmiennej?
- Co oznacza public/private?
- Co oznacza const?
- Co to są pola?
- Co to są metody?
- Czego tu brakuje?

# Użycie prostej klasy *Date*

## Oto przykład użycia prostej klasy

```
class DateType {
public:
   // constructor
   DateType();
   DateType(int newMonth, int newDay, int newYear);
   // accessor methods (get methods)
   int GetYear( ) const;   // returns Year
   int GetMonth( ) const;  // returns Month
   int GetDay( ) const;    // returns Day
private:
   int Year;
   int Month;
   int Day;
};
```

- Jak są inicjalizowane zmienne?
- Użycie dwóch konstruktorów
- Czy mamy dostęp do pól?
- Czy możemy używać metod?

```
DateType today(3, 4, 2004);
DateType tomorrow, someDay;

//can I do this?
   cout << today.Month;
//how about
   cout << today.GetMonth();
```

# Implementacja prostej klasy *Date*

- Oto **definicja** (**implementacja**) prostej klasy

```
// DateType.cpp
#include "DateType.h"
/** Constructors **/
DateType::DateType() {
   Day = 1;
   Month = 1;
   Year = 1;
}
DateType::DateType(int newMonth, int newDay, int newYear) {
   Day = newDay;
   Month = newMonth;
   Year = newYear;
}
// returns Year
int DateType::GetYear( ) const { return Year; }
// returns Month
int DateType::GetMonth( ) const { return Month; }
// returns Day
int DateType:: GetDay( ) const { return Day; }
```

- Czego brakowało w deklaracji?
- Co to za zmienne Day, Month, Year?
- Czy DateType.h jest niezbędny do kompilacji?

# Enkapsulacja i ukrywanie informacji

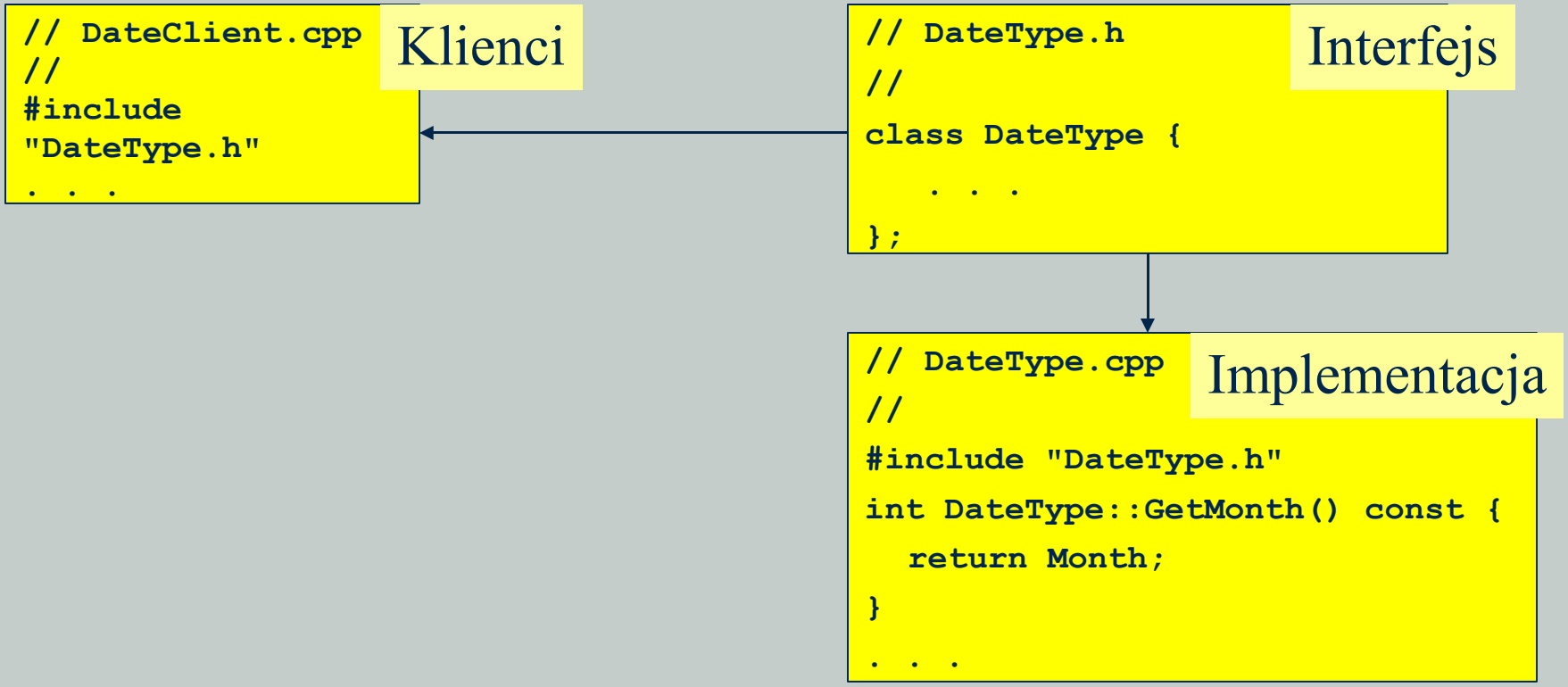

- Enkapsulacja
  - Klasa C++ dostarcza mechanizm grupowania danych i wykonywanych na nich operacji w jeden obiekt
- Ukrywanie informacji
  - Klasa C++ dostarcza mechanizm określania ograniczeń dostępu do pól i metod

# Organizacja implementacja

Aby umożliwić oddzielną kompilację, typowa organizacja implementacji składa się z dwóch plików:

| | |
|---|---|
| **`DateType.h`** | deklaracja klasy |
| **`DateType.cpp`** | definicje składowych |

Jeżeli użytkownik klasy DateType napisze program składający się z pojedynczego pliku **`DateClient.cpp`**, organizacja plików będzie następująca:



# Używanie składowych

Poza bezpośrednim użyciem składowych, klient używający klasy może zaimplementować funkcje wyższego poziomu które używają metod klasy, np.:

```
enum RelationType {Precedes, Same, Follows};
RelationType ComparedTo(DateType dateA, DateType dateB) {
  if (dateA.GetYear() < dateB.GetYear())
       return Precedes;
  if (dateA.GetYear() > dateB.GetYear())
       return Follows;
  if (dateA.GetMonth() < dateB.GetMonth())
       return Precedes;
  if (dateA.GetMonth() > dateB.GetMonth())
       return Follows;
  if (dateA.GetDay() < dateB.GetDay())
       return Precedes;
  if (dateA.GetDay() > dateB.GetDay())
       return Follows;
  return Same;
}
```

Klient

# Używanie składowych

Wówczas

```
DateType Tomorrow(1,18,2002), AnotherDay(10, 12, 1885);
if ( ComparedTo(Tomorrow, AnotherDay) == Same ) {
cout << "what do you think?" << endl;
}
```

Projektant klasy DateType mógłby również zaimplementować metodę porównującą dwie daty.

Jest to w istocie podejście bardziej naturalne i bardziej użyteczne, gdyż istnieje tylko jeden sposób na zadeklarowanie takiej funkcji.

# Dodatkowe metody klasy *DateType*

```
// add to DateType.h:
enum RelationType {Precedes, Same, Follows}; // file scoped
RelationType ComparedTo(DateType dateA); // to public section
```

```
// add implementation to DateType.cpp:
RelationType DateType::ComparedTo(DateType otherDate) {
  if (Year < otherDate.Year)
       return Precedes;
  if (Year > otherDate.Year)
       return Follows;
  if (Month < otherDate.Month)
       return Precedes;
  if (Month > otherDate.Month)
       return Follows;
  if (Day < otherDate.Day)
       return Precedes;
  if (Day > otherDate.Day)
       return Follows;
  return Same;
}
```

```
if ( Tomorrow.ComparedTo(AnotherDay) == Same )
  cout << "Think about it, Scarlett!" << endl;
```

# Używanie składowych

Kolejny przykład:

```
void PrintDate(DateType aDate, ostream& Out) {
  PrintMonth( aDate.GetMonth( ), Out );
  Out << ' ' << aDate.GetDay( )
      << ", " << setw(4) << aDate.GetYear( ) << endl;
}
void PrintMonth(int Month, ostream& Out) {
  switch (Month) {
      case 1: Out << "January"; return;
      case 2: Out << "February"; return;
      . . .
      case 12: Out << "December"; return;
      default: Out << "Juvember";
  }
}
```

Program:

```
DateType LeapDay(2, 29,2000);
PrintDate(LeapDay, cout);
```

wypisze:

```
February 29, 2000
```

# Klasyfikacja składowych

Metody implementują operacje na obiektach. Typy możliwych operacji mogą być klasyfikowane na różne sposoby. Oto powszechna klasyfikacja:

| | |
|---|---|
| **Konstruktor** | Operacja tworząca nowy egzemplarz klasy (nowy obiekt) |
| **Mutator** | Operacja zmieniająca stan jednego lub więcej pola klasy |
| **Obserwator** | Operacja odczytująca stan jednego lub więcej pól klasy, bez ich modyfikacji (Accessor, Getter) |
| **Iterator** | Operacja pozwalająca na kolejne przetwarzanie wszystkich elementów struktury danych |

W klasie *DateType*, *DateType()* jest konstruktorem, *GetYear()*, *GetMonth()* i *GetDay()* są obserwatorami. DateType nie posiada mutatorów ani iteratorów.

# Konstruktor domyślny

Klasa DateType ma dwa jawnie zdefiniowane konstruktory.

Zwykle definiuje się konstruktor domyślny, co gwarantuje inicjalizację każdego obiektu klasy:

```
DateType::DateType( ) {
   Month = Day = 1; // default date
   Year = 1980;
}
```

Konstruktor domyślny to konstruktor bez parametrów

Zasady dotyczące konstruktorów:

- nazwa składowej jest identyczna z nazwą klasy
- konstruktor nie ma specyfikacji wartości zwracanej, nawet **`void`**
- konstruktor domyślny jest wywoływany automatycznie podczas definicji egzemplarza klasy; jeżeli konstruktor przyjmuje parametry muszą one być podane po nazwie zmiennej w momencie deklaracji.

# Inne konstruktory

Klasa DateType posiada również konstruktor z parametrami. Pozwala to użytkownikowi na podanie reprezentowanej daty (ponieważ w klasie brak mutatorów, jest to jedyna możliwość).

```
DateType::DateType(int aMonth, int aDay, int aYear)
{
   if ( (aMonth >= 1 && aMonth <= 12)
        && (aDay >= 1) && (aYear >= 1) ) {
        Month = aMonth;
        Day = aDay;
        Year = aYear;
   }
   else {
        Month = Day = 1; // handling user error
        Year = 1980;
   }
}
```

**Kompilator określa który konstruktor wywołać w ten sam sposób, co dla funkcji przeciążonych.**

Jeżeli konstruktor przyjmuje parametry muszą one być podane po nazwie zmiennej w momencie deklaracji:

```
DateType aDate(10, 15, 2000);
DateType bDate(4, 0, 2005); // set to 1/1/1980
```

# Użycie domyślnego konstruktora

- Jeżeli nie dostarczy się jawnie żadnego konstruktora, kompilator wygeneruje automatycznie konstruktor domyślny. Konstruktor wygenerowany automatycznie
  - nie posiada parametrów
  - wywołuje konstruktor domyślny dla każdego pola, które jest klasą
  - nie inicjalizuje pól, które nie są klasami
- Dlatego:

**Projektując klasę, zawsze należy jawnie zaimplementować konstruktor domyślny**

# Wskaźnik *this*

- Rozważmy następujący fragment kodu:
  - Czy jest składniowo poprawny? Czy się skompiluje?
  - Czy jest semantycznie poprawny? Czy zadziała bez błędów czasu wykonania?
  - Czy robi coś sensownego?

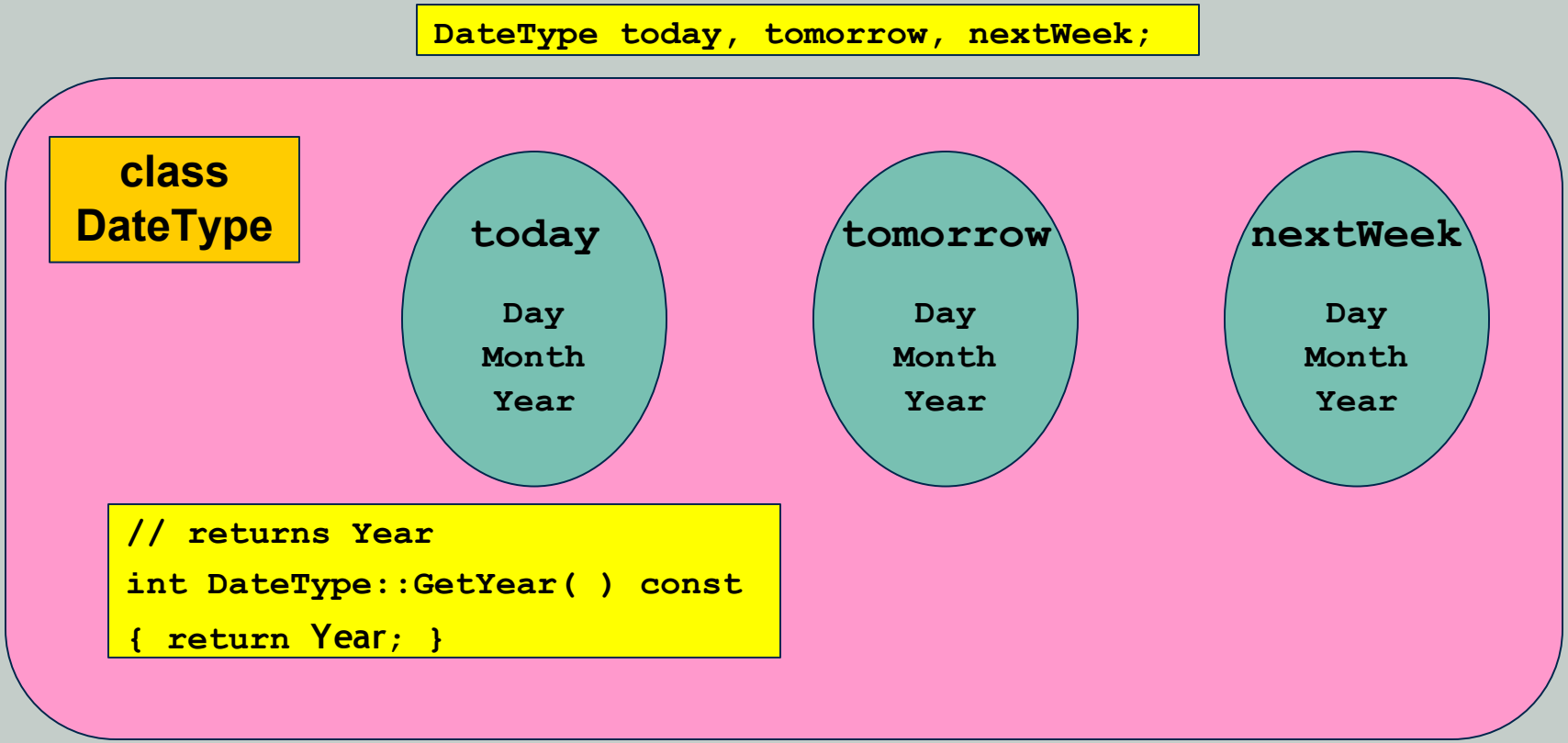
```
DateType::DateType(int Month, int Day, int Year) {
  Month = Month;
  Day = Day;
  Year = Year;
}
```

- Jaką wartością zostanie zainicjowane *today* w poniższym kodzie?

```
DateType today(3, 4, 2004);
```

# Wskaźnik *this*

- Aby zrozumieć *this* trzeba myśleć w kategoriach klas i obiektów



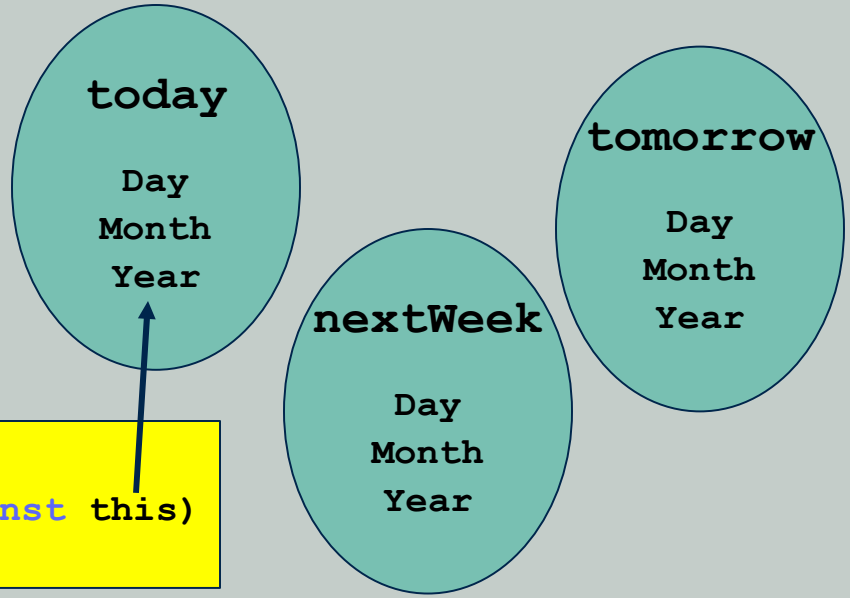

- Ktorych *Month*, *Day* i *Year* używamy w powyższej definicji?

# Wskaźnik *this*

- *this* jest przekazywany jako dodatkowy, niejawny argument do wszystkich metod i niejawnie używany do odwoływania się do pól klasy

```
// returns Year
int DateType::GetYear( ) const
{ return Year; }
```

```
int y=today.GetYear();
```




```
// returns Year
int DateType::GetYear(const DateType* const this)
{ return this->Year; }
```

- Tak wygląda definicja metody widziana przez kompilator
- *this* jest **stałym wskaźnikiem**, nie możemy go zmieniać wewnątrz metody,
- Ponieważ metoda jest typu *const* (jest obserwatorem), *this* jest również **wskaźnikiem do stałej**

# Destruktor

- Wywoływany automatycznie w chwili, kiedy zmienna jest usuwana z pamięci (m.in. kończy się jej zakres)
- W każdej klasie jest najwyżej jeden destruktor.
- Nazwa destruktora to nazwa klasy poprzedzona znakiem tyldy (~).
- Destruktor, podobnie jak konstruktor, nie ma typu wartości zwracanej (nawet *void*)
- Destruktor zwalnia zasoby używane przez obiekt (zaalokowaną pamięć, deskryptory plików, semafory etc.)

```
// stack.h
class stack {
   ...
   int* dane;
   ...
public:
   ...
   ~stack();
   ...
};
```

Interfejs

```
// stack.cpp
stack::~stack()
{
   free(dane);
};
```

Implementacja

```
{
   stack s;
   ...
   //tu wywoływany s.~stack()
}
```

Klient

# Klasa Stack

```
//stack.h
#define STACKSIZE 20

class stack
{
public:
   void push(int a);
   int pop();
   void clear();
   stack();
   ~stack();
private:
   int top;
   int dane[STACKSIZE];
};
```

Interfejs

```
//stack.cpp
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"

stack::stack()
{
   this->top=0;
};
stack::~stack(){};
void stack::clear()
{
   this->top=0;
};
void stack::push(int a)
{
   assert(this->top<STACKSIZE);
   this->dane[this->top++]=a;
};
int stack::pop()
{
   assert(this->top>0);
   return this->dane[--this->top];
};
```

Implementacja

# Klasa Stack - wersja ulepszona

- Do implementacji
  - Dynamiczna alokacja pamięci jak w drugiej wersji modułowej
  - Niepusty destruktor

```
//stack.h
class stack
{
public:
   void push(int a);
   int pop();
   void clear();
   stack();
   ~stack();
private:
   int top;
   int *dane;
   int size;
};
```

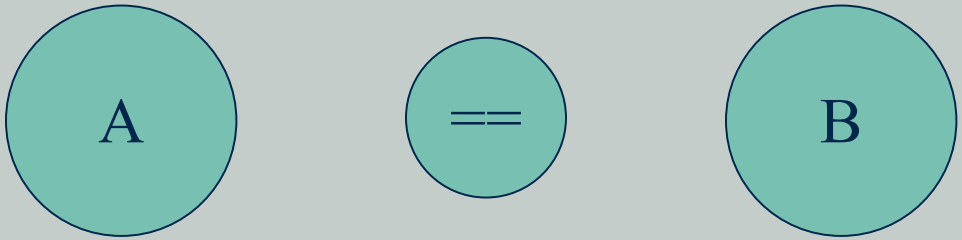
Interfejs

# Przeciążenie funkcji

- W C++ możliwe jest zadeklarowanie dwóch lub więcej funkcji o tej samej nazwie. Nazywa się to **przeciążeniem**.
- Kompilator określa, do której funkcji odnosi się każde wywołanie.
- Poszukiwanie najlepiej pasującej funkcji odbywa się na podstawie typów parametrów formalnych i aktualnych według kryteriów przedstawionych poniżej w podanej kolejności. Typ wartości zwracanej przez funkcję nie jest uwzględniany.
  - Dokładne dopasowanie (brak konwersji lub konwersje trywialne, np. tablicy do wskaźnika)
  - Dopasowanie z użyciem promocji (bool do int, char do int, float do double etc.)
  - Dopasowanie z użyciem konwersji standardowych (int do double, double do int etc.)
  - Dopasowanie z użyciem konwersji zdefiniowanych przez użytkownika
  - Dopasowanie z użyciem wielokropka (...)

# Przeciążenie operatorów

- Operatory C++ (np. ==, ++ etc.) mogą zostać przeciążone w celu umożliwienia operacji na typach zdefiniowanych przez użytkownika.




obiekt operator obiekt

oznacza

A.==(B)

czyli:

- Klasa typu A musi mieć zdefiniowaną jednoargumentową składową o nazwie ==

# Przeciążenie operatorów

- Składowe przeciążające operatory są definiowane z użyciem słowa kluczowego *operator*

```
// add to DateType.h:
bool operator==(Datetype otherDate) const ;
```

Interfejs

```
// add to DateType.cpp:
bool dateType::operator==(Datetype otherDate) const {
   return( (Day == otherdate.Day ) &&
   (Month == otherDate.Month ) &&
   (Year == otherDate.Year ));
}
```

Implementacja

```
DateType aDate(10, 15, 2000);
DateType bDate(10, 15, 2001);
if (aDate == bDate) { . . .
```

Klient

- Odpowiednio użyte przeciążenie operatorów pozwala na traktowanie obiektów typu zdefiniowanego przez użytkownika w sposób tak samo naturalny, jak typów wbudowanych.

# Domyślne argumenty funkcji

- W C++ możliwe jest podanie domyślnych wartości parametrów formalnych funkcji, które zostaną użyte w przypadku, kiedy pominie się je przy wywołaniu

```
// add to Datetype.h
DateType::DateType(int aMonth = 1, int aDay = 1, int aYear = 1980);
```

**Domyślne wartości parametrów podaje się w prototypie funkcji, a nie w implementacji.**

```
// add to DateType.cpp
DateType::DateType(int aMonth, int aDay, int
aYear){
  if ( (aMonth >= 1 && aMonth <= 12)
       && (aDay >= 1) && (aYear >= 1) ) {
       Month = aMonth;
       Day = aDay;
       Year = aYear;
  }
  else                  {
       Month = Day = 1; // default date
       Year = 1980;
       }
}
```

**Ponieważ wartości domyślne są podane dla wszystkich parametrów, można pominąć konstruktor domyślny. Pozostawienie konstruktora domyślnego spowoduje błąd przy próbie kompilacji.**

# Domyślne argumenty funkcji

- Jeżeli argument posiadający wartość domyślną zostanie pominięty przy wywołaniu funkcji, kompilator automatycznie wpisze wartość domyślną:

```
DateType dDate(2,29);     // Feb 29, 1980
DateType eDate(3);        // March 1, 1980
DateType fDate();         // Jan 1, 1980
```

- Jedynie ostatnie argumenty mogą zostać pominięte

```
DateType dDate(,29);      // error
```

- Wspólne użycie przeciążenia funkcji i parametrów domyślnych wymaga ostrożności

**Parametry domyślne mogą być użyte dla każdej funkcji, nie tylko dla konstruktorów i metod.**

# Używanie argumentów domyślnych

- Domyślne argumenty w prototypach funkcji
  - Można pomijać jedynie ostatnie argumenty
- Reguły stosowania domyslnych argumentów
  - Domyślne argumenty podaje się w pierwszej deklaracji/definicji funkcji (najczęściej jest to prototyp)
  - Domyślne wartości powinny być stałymi
  - Na liście parametrów w deklaracji funkcji argumenty domyślne muszą być ostatnimi argumentami
  - Przy wywołaniu funkcji posiadającej więcej niż jeden domyślny argument, argumenty po pierwszym pominiętym muszą być również pominięte
- Domyślne argumenty i konstruktory
  - Domyślne argumenty konstruktorów mogą zastąpić kilka osobnych konstruktorów
  - Domyślne argumenty konstruktorów zapewniają pełną inicjalizację obiektów
  - Konstruktory z wszystkimi parametrami domyślnymi zastępują konstruktor domyślny (bezparametrowy)

# Funkcje otwarte (inline)

- Najbardziej efektywne dla małych i średnich funkcji
- Rozwijane w miejscu wywołania
  - Brak narzutu na wywołanie funkcji
  - Kompilator generuje odpowiedni kod i odwzorowuje parametry
  - Oprócz tego generowana kopia implementacji funkcji (na wypadek, gdyby programista chciał pobrać jej adres)
- Dwie metody specyfikacji funkcji otwartej:
  - Podanie implementacji w deklaracji klasy
  - Użycie słowa kluczowego *inline* w definicji funkcji

# Przykłady funkcji otwartych

```
// DateType.h
class DateType {
public:
   DateType(int newMonth = 1, int newDay = 1,
   int newYear = 1980);
   int GetYear () const;
   int GetMonth ()const {return Month};
   int GetDay () const {return Day};
private:
   int Year, Month, Day;
};
```

Interfejs

```
// DateType.h
inline int DateType::GetYear() const { // explicit inline
   return Year;
}
```

Implementacja

# Funkcje otwarte

- Funkcje otwarte muszą być zdefiniowane w plikach nagłówkowych aby umożliwić kompilatorowi generację kopii funkcji w momencie ich użycia
- Metody zdefiniowane wewnątrz deklaracji klasy są niejawnie deklarowane jako otwarte
- Kompromis między wydajnością a ukrywaniem informacji
- Odwołanie się do pól zdefiniowanych poniżej ich definicji jest poprawne

```
// DateType.h
class DateType {
public:
   DateType(int newMonth = 1, int newDay = 1,
   int newYear = 1980);
   int GetYear () const;
   int GetMonth ()const {return Month};
   int GetDay () const {return Day};
private:
   int Year, Month, Day;
};
```

# Wady i zalety funkcji otwartych

- Pogwałcenie celów inżynierii oprogramowania
  - brak oddzielenia interfejsu od implementacji
  - brak ukrywania informacji
- Kod używający funkcji otwartych musi być ponownie skompilowany, gdy:
  - treść metody ulegnie zmianie
  - zostanie ona zastąpiona zwykłą metodą i odwrotnie
- *inline* to żądanie, nie polecenie
  - można rozwinąć ręcznie, ale to kosztowne
- Rozmiar pliku wykonywalnego może ulec powiększeniu
  - zwykle nie jest to problemem