

Podstawy Informatyki

Katedra Telekomunikacji, EiT

dr inż. Jarosław Bułat (c)

kwant@agh.edu.pl

Plan prezentacji

- » Przeciążenie operatorów
- » Funkcja rekurencyjna
- » Lista jednokierunkowa
- » Drzewa binarne

class Car{
car = car1 + car2; //???
przeciążenie operatorów

Przeciążenie operatorów

- » Operator może mieć różną implementację w zależności od typu użytych argumentów
 - nie wszystkie języki: C++/C#/Python/..., C/Java/PHP/...
 - poprawia czytelność kodu
 - umożliwia np. rozszerzenie biblioteki standardowej na poziomie języka
 - przykład **polimorfizmu**
 - tzw. Lukier składniowy “Syntactic sugar”

polimorfizm (*wielopostaciowość*) Pozwala programiście używać zmiennych, funkcji, wartości na kilka różnych sposobów:

jeden interfejs dla wielu różnych rzeczy

C++

+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbclx01/cplr318.htm

» Najczęściej używane: +, -, +=, -=, <<, >>, ==, =

Przykład

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

class Complex {
public:
    float re_, im_;
    Complex(float, float);
};

int main() {
    Complex c1(1, 3); // 1+j3
    Complex c2(3, -7); // 3-j7

    Complex c = c1 + c2;
    cout << c << endl;
}
```

- » Definiuję klasę complex operującą na liczbach zespolonych
- » Deklaruję i inicjalizuję dwa obiekty tej klasy

Przykład

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

class Complex {
public:
    float re_, im_;
    Complex(float, float);
};

int main() {
    Complex c1(1, 3); // 1+j3
    Complex c2(3, -7); // 3-j7

    Complex c = c1 + c2;
    cout << c << endl;
}
```

- » Definiuję klasę **complex** operującą na liczbach zespolonych
- » Deklaruję i inicjalizuję **dwa obiekty tej klasy**
- » Chciałbym używać operatorów arytmetycznych pomiędzy tymi obiektami

Przykład

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

class Complex {
public:
    float re_, im_;
    Complex(float, float);
};

int main() {
    Complex c1(1, 3); // 1+j3
    Complex c2(3, -7); // 3-j7

    Complex c = c1 + c2;
    cout << c << endl;
}
```

- » Definiuję klasę **complex** operującą na liczbach zespolonych
- » Deklaruję i inicjalizuję **dwa obiekty tej klasy**
- » Chciałbym używać operatorów arytmetycznych pomiędzy tymi obiektami
- » Chciałbym używać standardowych bibliotek

Przykład

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

```
class Complex {
public:
    float re_, im_;
    Complex(float, float);
};
```

```
int main() {
    Complex c1(1, 3); // 1+j3
    Complex c2(3, -7); // 3-j7
```

```
    Complex c = c1 + c2;
    cout << c << endl;
}
```

error: no match for '**operator+**'
(operand types are 'Complex' and
'Complex')

Complex c = c1 + c2;

error: no match for '**operator<<**'
(operand types are 'std::ostream
{aka std::basic_ostream<char>}'
and 'Complex')

cout << c << endl;

przeciążenie + operator+()

Przeciążenie “+”

» Definiuję nowy operator +

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
- » Wygląda podobnie jak definicja metody, posiada:

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
- » Wygląda podobnie jak definicja metody, posiada:
 - typ

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
- » Wygląda podobnie jak definicja metody, posiada:
 - typ
 - nazwę

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
- » Wygląda podobnie jak definicja metody, posiada:
 - typ
 - nazwę
 - zasięg

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
 - » Wygląda podobnie jak definicja metody, posiada:
 - typ
 - nazwę
 - zasięg
 - listę argumentów
- `const Complex &right`

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
- » Wygląda podobnie jak definicja metody
- » operator+ jest wywołany na obiekcie c1

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » Definiuję nowy operator +
- » Wygląda podobnie jak definicja metody
- » operator+ jest wywołany na obiekcie c1
- » Można całkiem “legalnie” jawnie wywołać **operator+**

Przeciążenie “+”

```
class Complex {};  
  
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » **Kolejność operacji**
- » Tworzę **nowy obiekt** typu Complex, będę w nim przechowywał wynik działania

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » **Kolejność operacji**
- » Tworzę nowy obiekt typu Complex, będę w nim przechowywał wynik działania
- » Wykonuję **dodawanie**

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » **Kolejność operacji**
- » Tworzę nowy obiekt typu Complex, będę w nim przechowywał wynik działania
- » Wykonuję dodawanie
 - **this** jest opcjonalne

jaką wartość ma this->re_ ???

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » **Kolejność operacji**
- » Tworzę nowy obiekt typu Complex, będę w nim przechowywał wynik działania
- » Wykonuję dodawanie

jaką wartość ma right.im_ ???

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » **Kolejność operacji**
- » Tworzę nowy obiekt typu Complex, będę w nim przechowywał wynik działania
- » Wykonuję dodawanie
- » Wyrażenie
 $c1 + c2$
 $c1.operator+(c2)$
tworzy nowy, tymczasowy obiekt, jest on **zwracany przez wartość** i przypisany do **c** i **d**

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

- » **Kolejność operacji**
- » Tworzę nowy obiekt typu Complex, będę w nim przechowywał wynik działania
- » Wykonuję dodawanie
- » Wyrażenie
 $c1+c2$
 $c1.operator+(c2)$
tworzy nowy, tymczasowy obiekt, jest on zwracany przez wartość i przypisany do **c** i **d**
- » **result** jest obiektem, jest wynikiem działania $c1+c2$

Przeciążenie “+”

» Deklaracja wymaga konstruktora domyślnego, należy go jawnie zdefiniować

```
class Complex {};  
  
Complex Complex::operator+(const  
Complex &right){  
    Complex result;  
    result.re_ = this->re_ + right.re_;  
    result.im_ = this->im_ + right.im_;  
    return result;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    Complex d = c1.operator+(c2);  
  
    cout << c.re_ << c.im_ << endl;  
}
```

Przeciążenie “+”

```
class Complex {};
```

```
Complex Complex::operator+(const
Complex &right){
    Complex result;
    result.re_ = this->re_ + right.re_;
    result.im_ = this->im_ + right.im_;
    return result;
}
```

```
int main() {
    Complex c1(1, 3); // 1+j3
    Complex c2(3, -7); // 3-j7

    Complex c = c1 + c2;
    Complex d = c1.operator+(c2);

    cout << c.re_ << c.im_ << endl;
}
```

- » Deklaracja wymaga konstruktora domyślnego, należy go jawnie zdefiniować
- » W tym momencie następuje **przypisanie**, wykonuje go domyślny operator przypisania:

```
Complex::operator=(const Complex&)
```

- » Operator ten jest aut. tworzony przez kompilator: przepisuje składowe
- » Można go oczywiście redefiniować



quiz

PI11_overl+

socrative.com

- login
- student login

Room name:

KWANTAGH

przeciążenie poza klasą

jako metoda lub jako funkcja

Przeciążenie “+”

- » Operator można zdefiniować:
- » **wewnątrz klasy**
- » **poza nią** (jako funkcję)
 - left: c1
 - right: c2

```
Complex Complex::operator+(const Complex &right);  
Complex c = c1 + c2;  
Complex d = c1.operator+(c2);
```

```
Complex operator+(const Complex& left, const Complex& right);  
Complex c = c1 + c2;  
Complex d = c1.operator+(c2);  
Complex d = operator+(c1, c2);
```

Przeciążenie “+”

- » Operator można zdefiniować:
- » wewnątrz klasy
- » poza klasą (jako funkcję)
 - left: c1
 - right: c2
- » Nie wszystkie operatory mogą być definiowane poza klasą

Przeciążenie “+”

Expression	As member function	As non-member function
@a	(a).operator@ ()	operator@ (a)
a@b	(a).operator@ (b)	operator@ (a, b)
a=b	(a).operator= (b)	cannot be non-member
a(b...)	(a).operator()(b...)	cannot be non-member
a[b]	(a).operator[](b)	cannot be non-member
a->	(a).operator-> ()	cannot be non-member
a@	(a).operator@ (0)	operator@ (a, 0)

- » Operator można zdefiniować:
- » wewnątrz klasy
- » poza klasą (jako funkcję)
 - left: c1
 - right: c2
- » Nie wszystkie operatory mogą być definiowane poza klasą

<http://en.cppreference.com/w/cpp/language/operators>

Przeciążenie “<<”

```
class Complex {  
    public:  
        float re_, im_;  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
    Complex c2(3, -7); // 3-j7  
  
    Complex c = c1 + c2;  
    cout << c << endl;  
}
```

» Instrukcję:

cout << c << endl

» Można zapisać:

cout.operator<<(c) << endl

» Czyli **operator <<** nie wykonuje się na klasie **Complex** tylko na klasie **ostream**, na obiekcie **std::cout**

» **Przeciążenie operatora << musi być zrealizowane poza klasą**

Przeciążenie “<<”

» Deklaracja/definicja klasy nie zawiera żadnej “wzmianki” o przeciążeniu operatora “<<”

```
class Complex {  
    public:  
        float re_, im_;  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

Przeciążenie “<<”

```
class Complex {  
    public:  
        float re_, im_;  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

- » Deklaracja/definicja klasy nie zawiera żadnej “wzmianki” o przeciążeniu operatora “<<”
- » Przeciążenie **operator<<** jest funkcją, nie metodą

Przeciążenie “<<”

```
class Complex {  
    public:  
        float re_, im_;  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

- » Deklaracja/definicja klasy nie zawiera żadnej “wzmianki” o przeciążeniu operatora “<<”
- » Przeciążenie **operator<<** jest funkcją, nie metodą
- » Jak kompilator odnajduje odpowiednią implementację “<<” ???

Przeciążenie “<<”

```
class Complex {  
    public:  
        float re_, im_;  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

- » Deklaracja/definicja klasy nie zawiera żadnej “wzmianki” o przeciążeniu operatora “<<”
- » Przeciążenie **operator<<** jest funkcją, nie metodą
- » Jak kompilator odnajduje odpowiednią implementację “<<” ???
 - **typ lewego argumentu**

Przeciążenie “<<”

```
class Complex {
public:
    float re_, im_;
    Complex(float, float);
    Complex operator+(...);
};

ostream &operator<< (ostream &out,
const Complex &right){
    out << right.re_ << " ";
    out << right.im_ << endl;
    return out;
}

int main() {
    Complex c1(1, 3); // 1+j3

    cout << c1 << endl;
}
```

- » Deklaracja/definicja klasy nie zawiera żadnej “wzmianki” o przeciążeniu operatora “<<”
- » Przeciążenie `operator<<` jest funkcją, nie metodą
- » Jak kompilator odnajduje odpowiednią implementację “<<” ???
 - typ lewego argumentu
 - typ prawego argumentu

Przeciążenie "<<"

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

» Wywołanie kaskadowe:

```
cout << c1 << c2 << endl;
```

jest możliwe ponieważ
implementacja <<:

– otrzymuje referencję na **out**

Przeciążenie "<<"

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

» Wywołanie kaskadowe:

```
cout << c1 << c2 << endl;
```

jest możliwe ponieważ
implementacja <<:

- otrzymuje referencję na `out`
- wykonuje na niej operację

Przeciążenie “<<”

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

» Wywołanie kaskadowe:

```
cout << c1 << c2 << endl;
```

jest możliwe ponieważ
implementacja <<:

- otrzymuje referencję na `out`
- wykonuje na niej operację
- zwraca referencję na ten obiekt aby następny operator mógł wykonać następną operację

Przeciążenie "<<"

» Operator << można rozwinąć jako funkcje

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
}
```

Przeciążenie “<<”

» Operator << można rozwinąć jako funkcje

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
    operator<<(cout, c1) << endl;  
}
```

Przeciążenie “<<”

» Operator << można rozwinąć jako funkcje

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
    ( operator<<(cout, c1) )<<(endl);  
}
```

Przeciążenie "<<"

- » Operator << można rozwinąć jako funkcje
- » Referencja zwracana jako rezultat umożliwia kaskadowe wywołanie (wywołanie funkcji na wyniku funkcji)

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
    ( operator<<(cout, c1) )<<(endl);  
}
```

Przeciążenie "<<"

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
  
    cout << c1 << endl;  
    ( operator<<(cout, c1) )<<(endl);  
}
```

- » Operator << można rozwinąć jako funkcje
- » Referencja zwracana jako rezultat umożliwia kaskadowe wywołanie (wywołanie funkcji na wyniku funkcji)
- » Przyporządkowanie argumentów

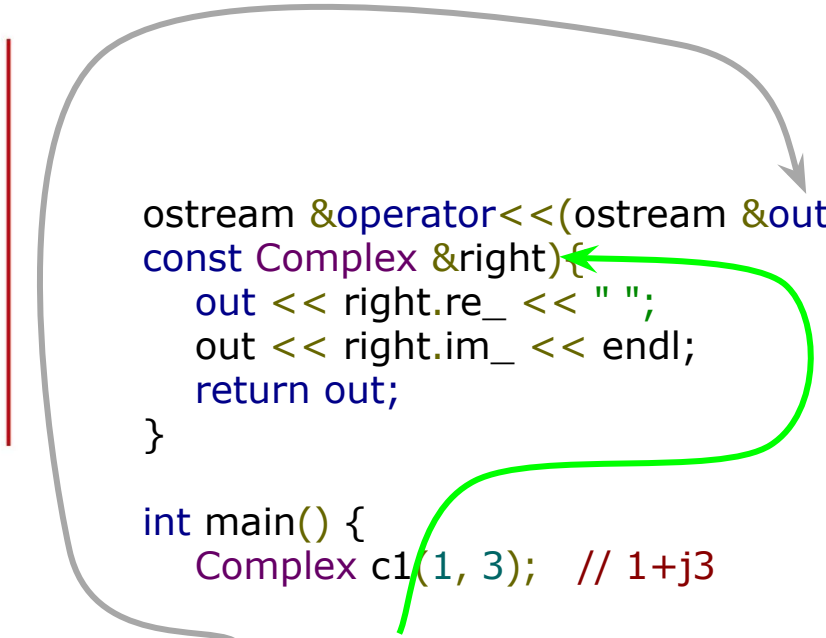
Przeciążenie "<<"

- » Operator << można rozwinąć jako funkcje
- » Referencja zwracana jako rezultat umożliwia kaskadowe wywołanie (wywołanie funkcji na wyniku funkcji)
- » Przyporządkowanie argumentów
 - std::cout

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
    ( operator<<(cout, c1) )<<(endl);  
}
```

Przeciążenie "<<"

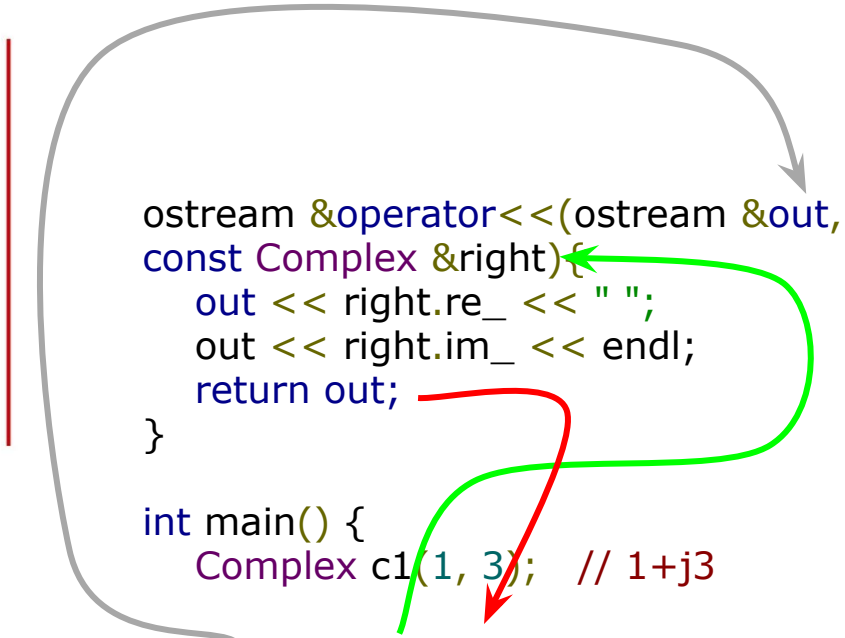


```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
    ( operator<<(cout, c1) )<<(endl);  
}
```

- » Operator << można rozwinąć jako funkcje
- » Referencja zwracana jako rezultat umożliwia kaskadowe wywołanie (wywołanie funkcji na wyniku funkcji)
- » Przyporządkowanie argumentów
 - std::cout
 - argument

Przeciążenie "<<"



```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

```
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
    ( operator<<(cout, c1) )<<(endl);  
}
```

- » Operator << można rozwinąć jako funkcje
- » Referencja zwracana jako rezultat umożliwia kaskadowe wywołanie (wywołanie funkcji na wyniku funkcji)
- » Przyporządkowanie argumentów
 - std::cout
 - argument
 - std::cout na którym można wykonać następną operację



quiz

PI11_overl<<

socrative.com

- login
- student login

Room name:

KWANTAGH

Przeciążenie "<<"

» Zazwyczaj w klasie dane będą prywatne

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
}
```

Przeciążenie "<<"

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
}
```

- » Zazwyczaj w klasie dane będą prywatne
- » Jak wtedy uzyskać dostęp do `re_` i `im_`?

Przeciążenie "<<"

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
}
```

- » Zazwyczaj w klasie dane będą prywatne
- » Jak wtedy uzyskać dostęp do `re_` i `im_`?
 - można dodać "getter"

Przeciążenie "<<"

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
};  
  
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}  
  
int main() {  
    Complex c1(1, 3); // 1+j3  
    cout << c1 << endl;  
}
```

» Zazwyczaj w klasie dane będą prywatne

» Jak wtedy uzyskać dostęp do `re_` i `im_`?

- można dodać "getter"
- można zaprzyjaźnić się z funkcją "operator<<"

error: 'float Complex::im_' is private
float re_, im_;

przyjaciół (**friend**)
ma dostęp do moich sekretów

friend

```
class Friendship{  
    private:  
        int x_;  
    public:  
        friend void f(Friendship &obj);  
        void setX(int x);  
        int getX();  
};  
  
void f(Friendship &obj){  
    obj.x_ = 10;  
}  
  
int main() {  
    Friendship ex;  
    f(ex);  
    cout << ex.getX() << endl;  
}
```

- » W klasie zadeklarowano, że funkcja `f(...)` będzie zaprzyjaźniona
- » To nie jest deklaracja metody!!!
- » Funkcja `f(...)` jest zwykłą funkcją, nie ma operatora dostępu
- Friendship::**
 - » Funkcja `f(...)` ma dostęp do prywatnych składowych klasy
 - » Funkcja zaprzyjaźniona może użyć dowolnej składowej obiektu który jest w zasięgu
 - » “Przyjaźń” inicjalizuje klasa

Przeciążenie "<<"

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
};
```

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

» Zazwyczaj w klasie dane będą prywatne

» Jak wtedy uzyskać dostęp do `re_` i `im_`?

- można dodać "getter"
- można zaprzyjaźnić się z funkcją "operator<<"

error: 'float Complex::im_' is private
float re_, im_;

Przeciążenie “<<”

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
        friend ostream &operator<<(  
            ostream &out, const Complex &right);  
};
```

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

- » Zazwyczaj w klasie dane będą prywatne
- » Jak wtedy uzyskać dostęp do `re_` i `im_`?
 - można dodać “getter”
 - można zaprzyjaźnić się z funkcją “operator<<”
- » Funkcja `operator<<(...)` jest teraz przyjacielem klasy `Complex` więc może używać jej prywatnych komponentów

Przeciążenie “<<”

```
class Complex {  
    private:  
        float re_, im_;  
    public:  
        Complex(float, float);  
        Complex operator+(...);  
        friend ostream &operator<<(  
            ostream &out, const Complex &right);  
};
```

```
ostream &operator<<(ostream &out,  
const Complex &right){  
    out << right.re_ << " ";  
    out << right.im_ << endl;  
    return out;  
}
```

- » Jest to typowa konstrukcja przeciążonego operatora <<
- » Możliwość definiowania przeciążonego operatora poza klasą ma na celu umożliwienie mieszania typów składników:

A << B;

c = a + b;



quiz

PI11_friend

socrative.com

- login
- student login

Room name:

KWANTAGH

**Żeby zrozumieć rekurencję musisz
najpierw zrozumieć rekurencję**

Rekurencja

- » Upraszczając (bardzo): rekurencja to odwołanie się w definicji funkcji do tej samej funkcji

```
int foo(int x){  
    y = foo(x-1);  
    return y;  
}
```

- » Jest podstawową techniką w językach funkcyjnych
- » Zastępuje iteracje (np. pętle)
- » Nadaje się do przeszukiwania struktur danych w postaci plików XML, nieregularnego drzewa, “linked list”, etc...
- » Zawsze zwiększa zapotrzebowanie programu na zasoby (RAM i CPU)

Rekurencja - silnia

» Algorytm obliczania silni:

$$n! = \prod_{k=1}^n k \quad \text{dla } n \geq 1$$

» Zatem:

$$4! = 1*2*3*4$$

Rekurencja - silnia

» Algorytm obliczania silni:

$$n! = \prod_{k=1}^n k \quad \text{dla } n \geq 1$$

» Można ten algorytm zapisać rekurencyjnie w postaci:

$$\begin{aligned} b_n &= n b_{n-1} \\ b_0 &= 1 \end{aligned}$$

Rekurencja - silnia

- » Algorytm obliczania silni:

$$n! = \prod_{k=1}^n k \quad \text{dla } n \geq 1$$

- » Można ten algorytm zapisać rekurencyjnie w postaci:

$$\begin{aligned} b_n &= n b_{n-1} \\ b_0 &= 1 \end{aligned}$$

pierwszą linię algorytmu należy iterować tak długo aż zostanie osiągnięty stan b_0 , który jest znany i wynosi 1

Rekurencja - silnia

algorytm:

$$\begin{aligned} b_n &= n b_{n-1} \\ b_0 &= 1 \end{aligned}$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

» Algorytm obliczania silni:

$$n! = \prod_{k=1}^n k \quad \text{dla } n \geq 1$$

» Można ten algorytm zapisać rekurencyjnie w postaci:

$$\begin{aligned} b_n &= n b_{n-1} \\ b_0 &= 1 \end{aligned}$$

pierwszą linię algorytmu należy iterować tak długo aż zostanie osiągnięty stan b_0 , który jest znany i wynosi 1

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact<=0){
        return 1;
    } else {
        return fact*factorial(fact-1);
    }
}
```

```
int main(){
    cout << factorial(3) << endl;
}
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

int factorial(int fact){
    if (fact<=0){
        return 1;
    } else {
        return fact*factorial(fact-1);
    }
}

int main(){
    cout << factorial(3) << endl;
}
```

Rekurencja - silnia

algorytm:

$$b_n = n \cdot b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
int main(){
    cout << factorial(3) << endl;
}
```

Rekurencja - silnia

algorytm:

$$b_n = n \cdot b_{n-1}$$

$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned}
 b_3 &= 3 * b_2 \\
 &= 3 * (2 * b_1) \\
 &= 3 * (2 * (1 * b_0)) \\
 &= 3 * (2 * (1 * 1)) \\
 &= 3 * (2 * 1) \\
 &= 3 * 2 \\
 &= 6
 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
int main(){
    cout << factorial(3) << endl;
}
```

Rekurencja - silnia

algorytm:

$$b_n = n \cdot b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$b_3 = 3 * b_2$$
$$= 3 * (2 * b_1)$$
$$= 3 * (2 * (1 * b_0))$$
$$= 3 * (2 * (1 * 1))$$
$$= 3 * (2 * 1)$$
$$= 3 * 2$$
$$= 6$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
```

Rekurencja - silnia

algorytm:

$$b_n = n \cdot b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3 * factorial(2))
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3 * factorial(2))
```


Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
factorial(3*factorial(2*factorial(1*factorial(0))))
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact<=0){
        return 1;
    } else {
        return fact*factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
factorial(3*factorial(2*factorial(1*factorial(0))))
factorial(3*factorial(2*factorial(1*1)))
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
factorial(3*factorial(2*factorial(1*factorial(0))))
factorial(3*factorial(2*factorial(1*1)))
factorial(3*factorial(2*1))
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
factorial(3*factorial(2*factorial(1*factorial(0))))
factorial(3*factorial(2*factorial(1*1)))
factorial(3*factorial(2*1))
factorial(3*2)
6
```

Rekurencja - silnia

algorytm:

$$b_n = n b_{n-1}$$
$$b_0 = 1$$

rozwiązanie:

$$\begin{aligned} b_3 &= 3 * b_2 \\ &= 3 * (2 * b_1) \\ &= 3 * (2 * (1 * b_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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

```
int factorial(int fact){
    if (fact <= 0){
        return 1;
    } else {
        return fact * factorial(fact-1);
    }
}
```

```
factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
factorial(3*factorial(2*factorial(1*factorial(0))))
factorial(3*factorial(2*factorial(1*1)))
factorial(3*factorial(2*1))
factorial(3*2)
6
```

Rekurencja - silnia

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

int factorial(int fact){
    if (fact<=0){
        return 1;
    } else {
        return fact*factorial(fact-1);
    }
}

int main(){
    cout << factorial(3) << endl;
}
```

- » Problem stopu
- » Warunek zakończenia algorytmu == warunek zakończenia rekurencji

Rekurencja - silnia

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

int factorial(int fact){
    return fact*factorial(fact-1);
}

int main(){
    cout << factorial(3) << endl;
}
```

- » Problem stopu
- » Warunek zakończenia algorytmu == warunek zakończenia rekurencji
- » Błędny warunek == nieskończona rekurencja:

“./ex11” terminated by signal
SIGSEGV (Address boundary
error)

Rekurencja - silnia

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

int factorial(int fact){
    return fact*factorial(fact-1);
}

int main(){
    cout << factorial(3) << endl;
}
```

- » Problem stopu
- » Można sobie wyobrazić implementację rekurencji:

```
int foo(int x, int maxRec){
    if(maxRec > ....){
        ????
```
- » **warunek** “awaryjnie” kończy algorytm - **to jest zły pomysł**
- » Warunek stopu powinien być cechą algorytmu
- » Głębokość rekurencji może zależeć od danych - **problem**

Rekurencja - silnia

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

int factorial(int fact){
    return fact*factorial(fact-1);
}

int main(){
    cout << factorial(3) << endl;
}

factorial(3)
factorial(3*factorial(2))
factorial(3*factorial(2*factorial(1)))
factorial(3*factorial(2*factorial(1....)))
factorial(3*factorial(2*factorial(1*1)))
factorial(3*factorial(2*1))
factorial(3*2)
```

- » Każde kolejne zagłębienie (zagnieżdżenie) to **wywołanie funkcji bez zakończenia poprzedniej**
- » Podczas wywołania funkcji na stosie odkładany jest adres powrotu, wartości rejestrów, etc...
- » Zbyt duży poziom zagłębienia przepełni stos
- » Wykorzystując rekurencję można uprościć algorytm, kosztem zasobów (RAM+CPU)



quiz

PI11_req

socrative.com

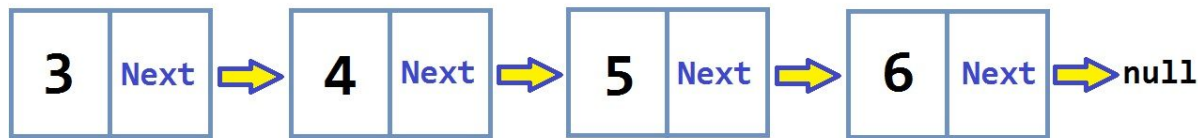
- login
- student login

Room name:

KWANTAGH

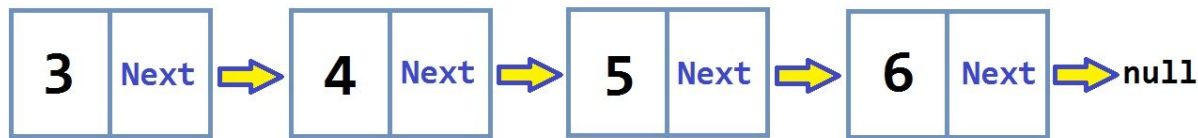
Lista jednokierunkowa ang. *linked list*

lista jednokierunkowa



- » Każdy bloczek to dowolnie złożona struktura
- » Każda struktura zawiera **wskaźnik na następną**
- » “Łańcuch” takich struktur **można w trakcie działania programu modyfikować**:
 - dostawić nowy element na końcu
 - wyrzucić [5] a [4]->next przypisać do [6]
 - zamienić [3] z [6] bez kopiowania, tylko przepisując odpowiednie [*]->next
 - zmienić wartość każdego elementu
- » Dostęp jest sekwencyjny

lista jednokierunkowa



» **Zalety:**

- duża elastyczność w manipulacji danymi
- zmiany nie wymagają kopiowania
- nie wymagana ciągła przestrzeń adresowa
- możliwe grafy, drzewa, dwukierunkowy listy

» **Wady:**

- większa zajętość pamięci (wymagany “next”)
- przeglądanie sekwencyjne (nie da się list[30])*
- tworzenie listy wymaga **new** ... czyli syscall*

lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};  
  
list *n0, *n1, *n2;  
  
n2 = new list;    // tail  
n2->x = 0;  
n2->next = NULL;  
  
n1 = new list;    //  
n1->x = 1;  
n1->next = n2;  
  
n0 = new list;    // head  
n0->x = 2;  
n0->next = n1;
```

- » Struktura składa się z węzłów, w których są dane oraz wskaźnika na następny węzeł
- » Węzły są tworzone dynamicznie, inicjalizowane i łączone ze sobą
- » NULL w ostatnim węźle oznacza koniec łańcucha

lista jednokierunkowa

- » Tworzenie listy w pętli:
 - koniec listy

```
struct list{  
    int x;  
    list *next;  
};  
  
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```


lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};
```

```
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```

- » Tworzenie listy w pętli:
 - koniec listy
 - utworzenie nowego węzła

lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};  
  
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```

- » Tworzenie listy w pętli:
- koniec listy
 - utworzenie nowego węzła
 - wypełnienie danymi

lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};  
  
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```

- » Tworzenie listy w pętli:
- koniec listy
 - utworzenie nowego węzła
 - wypełnienie danymi
 - **dowiązanie następnego węzła**

lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};  
  
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```

- » Tworzenie listy w pętli:
- koniec listy
 - utworzenie nowego węzła
 - wypełnienie danymi
 - dowiązanie następnego węzła
 - przesunięcie wskaźnika z następnego na bieżący węzeł

lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};  
  
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```

- » Tworzenie listy w pętli:
- koniec listy
 - utworzenie nowego węzła
 - wypełnienie danymi
 - dołączanie następnego węzła
 - przesunięcie wskaźnika z następnego na bieżący węzeł
 - zwrócenie wskaźnika na pierwszy węzeł listy

lista jednokierunkowa

```
struct list{  
    int x;  
    list *next;  
};  
  
list *createList(size_t size){  
    list *node = NULL;  
    for (int x=0; x<size; x++){  
        list *tmp = new list;  
        tmp->x=x;  
        tmp->next = node;  
        node = tmp;  
    }  
    return node;  
}
```

- » Tworzenie listy w pętli:
 - koniec listy
 - utworzenie nowego węzła
 - wypełnienie danymi
 - dołączanie następnego węzła
 - przesunięcie wskaźnika z następnego na bieżący węzeł
 - zwrócenie wskaźnika na pierwszy węzeł listy
- » Funkcja tworzy listę od końca do początku

Przeszukanie rekurencyjne

```
struct list{  
    int x;  
    list *next;  
};
```

```
int main(){  
    list *head = createList(10);  
    cout << find(head, 4);  
}
```

```
bool find(list *node, int x){  
    if (node->x == x){  
        return true;  
    }  
    if (!node->next){  
        return false;  
    }  
    return find(node->next, x);  
}
```

» Cel: przeszukaj całą listę, sprawdź czy zawiera "x"

» Utwórz listę

» Przeszukaj

Przeszukiwanie rekurencyjne

- » Algorytm w funkcji **find(...)**:
 - jeżeli bieżący element zawiera poszukiwaną wartość **zwróć true**

```
struct list{
    int x;
    list *next;
};

int main(){
    list *head = createList(10);
    cout << find(head, 4);
}

bool find(list *node, int x){
    if (node->x == x){
        return true;
    }
    if (!node->next){
        return false;
    }
    return find(node->next, x);
}
```


Przeszukiwanie rekurencyjne

```
struct list{  
    int x;  
    list *next;  
};  
  
int main(){  
    list *head = createList(10);  
    cout << find(head, 4);  
}
```

```
bool find(list *node, int x){  
    if (node->x == x){  
        return true;  
    }  
    if (!node->next){  
        return false;  
    }  
    return find(node->next, x);  
}
```

» Algorytm w funkcji **find(...)**:

- jeżeli bieżący element zawiera poszukiwaną wartość zwróć true
- jeżeli to jest ostatni element, zwróć false (na pewno nie zawiera poszukiwanej wartości)

Przeszukiwanie rekurencyjne

```
struct list{
    int x;
    list *next;
};

int main(){
    list *head = createList(10);
    cout << find(head, 4);
}

bool find(list *node, int x){
    if (node->x == x){
        return true;
    }
    if (!node->next){
        return false;
    }
    return find(node->next, x);
}
```

- » Algorytm w funkcji **find(...)**:
- jeżeli bieżący element zawiera poszukiwaną wartość zwróć true
 - jeżeli to jest ostatni element, zwróć false (na pewno nie zawiera poszukiwanej wartości)
 - kontynuuj wyszukiwania w następnym węźle

Przeszukiwanie rekurencyjne

```
struct list{
    int x;
    list *next;
};

int main(){
    list *head = createList(10);
    cout << find(head, 4);
}

bool find(list *node, int x){
    if (node->x == x){
        return true;
    }
    if (!node->next){
        return false;
    }
    return find(node->next, x);
}
```

- » Algorytm w funkcji **find(...)**:
- jeżeli bieżący element zawiera poszukiwaną wartość zwróć true
 - jeżeli to jest ostatni element, zwróć false (na pewno nie zawiera poszukiwanej wartości)
 - kontynuuj wyszukiwania w następnym węźle
 - przekaż z powrotem wynik wyszukiwania w następnych węzłach

Przeszukiwanie rekurencyjne

- » Algorytm w funkcji **find(...)**:
- » Algorytm w wersji z pętlą

```
struct list{  
    int x;  
    list *next;  
};
```

```
int main(){  
    list *head = createList(10);  
    cout << find(head, 4);  
}
```

```
bool find(list *node, int x){  
    if (node->x == x){  
        return true;  
    }  
    if (!node->next){  
        return false;  
    }  
    return find(node->next, x);  
}
```

```
bool findIter(list *node, int x){  
    do{  
        if (node->x == x){  
            return true;  
        }  
        node = node->next;  
    }while (node);  
    return false;  
}
```

Przeszukiwanie rekurencyjne

```
struct list{  
    int x;  
    list *next;  
};
```

```
int main(){  
    list *head = createList(10);  
    cout << find(head, 4);  
}
```

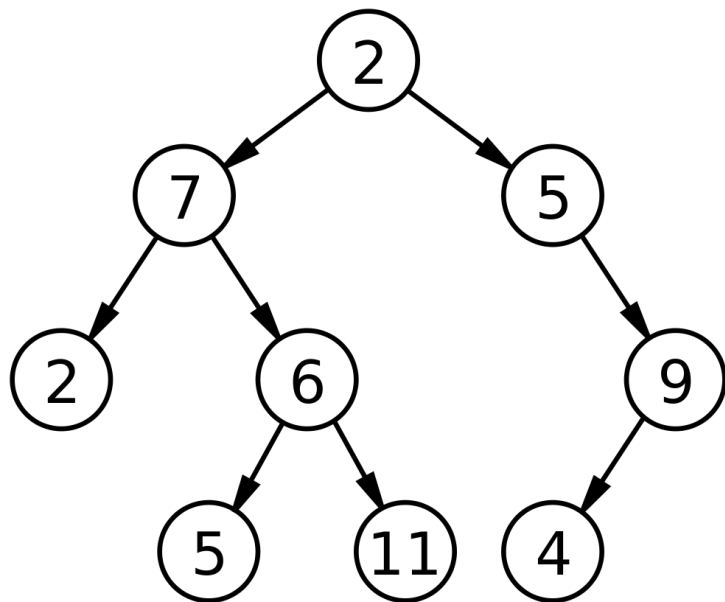
```
bool find(list *node, int x){  
    if (node->x == x){  
        return true;  
    }  
    if (!node->next){  
        return false;  
    }  
    return find(node->next, x);  
}
```

- » Algorytm w funkcji **find(...)**:
- » Algorytm w wersji z pętlą
- » Czy w tym wypadku warto stosować rekurencję?

NIE

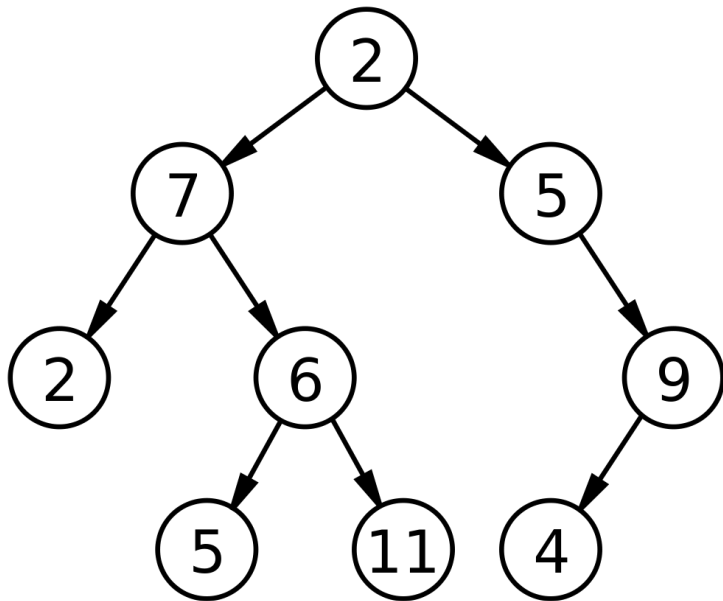
```
bool findIter(list *node, int x){  
    do{  
        if (node->x == x){  
            return true;  
        }  
        node = node->next;  
    }while (node);  
    return false;  
}
```

Drzewo binarne



- » **Struktura danych**
- » Każdy węzeł może mieć:
 - 0 potomków (np. node 2)
 - 1 potomka (np. node 9)
 - 2 potomków (np. node 7)

Drzewo binarne



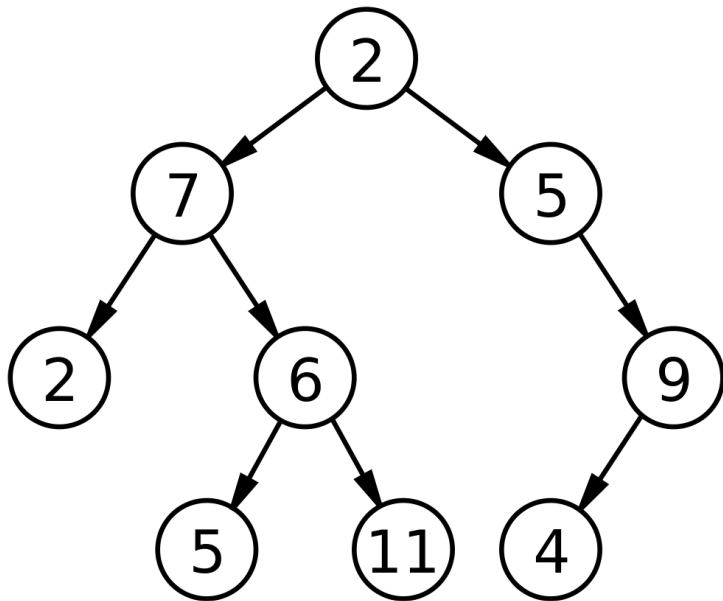
https://en.wikipedia.org/wiki/Binary_tree

- » Struktura opisująca węzeł drzewa binarnego jest podobna do listy

```
struct lista{  
    int x;  
    lista *next;  
};
```

```
struct node{  
    int x;  
    node *left;  
    node *right;  
};
```

Przeszukiwanie rekurencyjne



https://en.wikipedia.org/wiki/Binary_tree

- » Struktura opisująca węzeł drzewa binarnego jest podobna do listy

```
struct lista{  
    int x;  
    lista *next;  
};
```

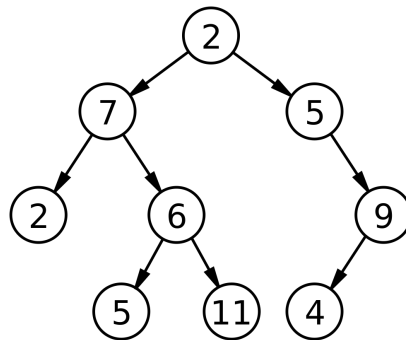
```
struct node{  
    int x;  
    node *left;  
    node *right;  
};
```


Przeszukiwanie rekurencyjne

» Algorytm przeszukiwania:

```
struct node{
    int x;
    node *left, *right;
};

bool find(node *current, int x){
    if (!current){
        return false;
    }
    if (current->x == x){
        return true;
    }
    if (find(current->left, x)){
        return true;
    }
    if (find(current->right, x)){
        return true;
    }
    return false;
}
```

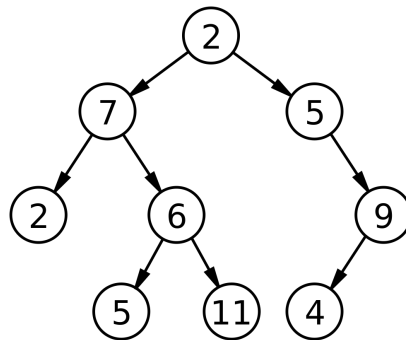


Przeszukiwanie rekurencyjne

```
struct node{  
    int x;  
    node *left, *right;  
};
```

```
bool find(node *current, int x){  
    if (!current){  
        return false;  
    }  
    if (current->x == x){  
        return true;  
    }  
    if (find(current->left, x)){  
        return true;  
    }  
    if (find(current->right, x)){  
        return true;  
    }  
    return false;  
}
```

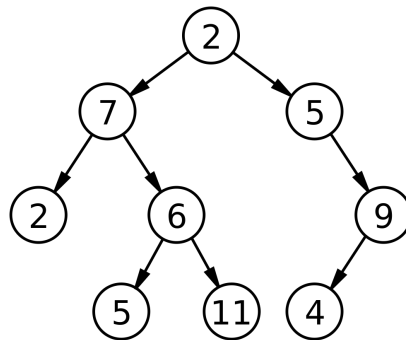
- » Algorytm przeszukiwania:
- sprawdzam czy bieżący nie jest NULL



Przeszukiwanie rekurencyjne

```
struct node{  
    int x;  
    node *left, *right;  
};  
  
bool find(node *current, int x){  
    if (!current){  
        return false;  
    }  
    if (current->x == x){  
        return true;  
    }  
    if (find(current->left, x)){  
        return true;  
    }  
    if (find(current->right, x)){  
        return true;  
    }  
    return false;  
}
```

- » Algorytm przeszukiwania:
- sprawdzam czy bieżący nie jest NULL
 - warunek końca (znaleziona wartość)



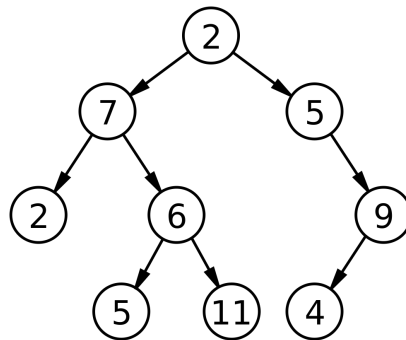
Przeszukiwanie rekurencyjne

```

struct node{
    int x;
    node *left, *right;
};

bool find(node *current, int x){
    if (!current){
        return false;
    }
    if (current->x == x){
        return true;
    }
    if (find(current->left, x)){
        return true;
    }
    if (find(current->right, x)){
        return true;
    }
    return false;
}
  
```

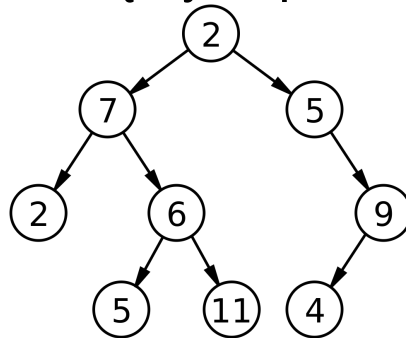
- » Algorytm przeszukiwania:
- sprawdzam czy bieżący nie jest NULL
 - warunek końca (znaleziona wartość)
 - wywołuję algorytm dla lewej i prawej gałęzi



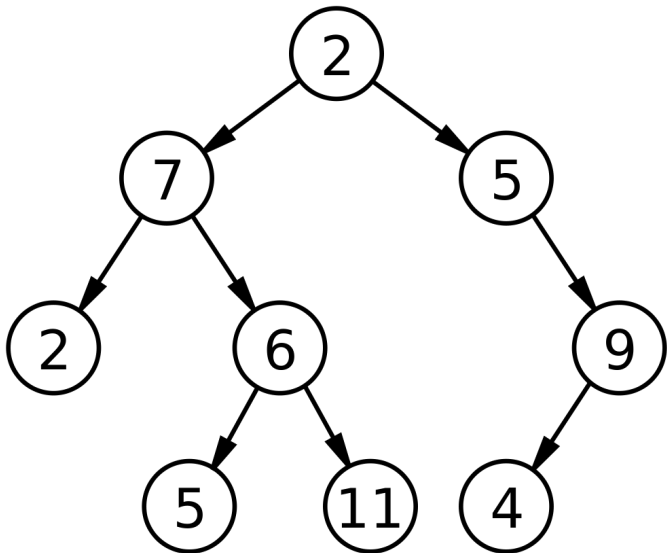
Przeszukiwanie rekurencyjne

```
struct node{  
    int x;  
    node *left, *right;  
};  
  
bool find(node *current, int x){  
    if (!current){  
        return false;  
    }  
    if (current->x == x){  
        return true;  
    }  
    return (find(current->left, x) ||  
            find(current->right, x));  
}
```

- » Algorytm przeszukiwania:
- sprawdzam czy bieżący nie jest NULL
 - warunek końca (znaleziona wartość)
 - wywołuję algorytm dla lewej i prawej gałęzi
 - zwięzły zapis



Przeszukiwanie rekurencyjne



- » Algorytm rekurencyjny świetnie działa dla problemów rekurencyjnych
- » Upraszcza zapis algorytmu
- » Umożliwia (zastępuje) realizację pętli dla języków które nie posiadają pętli
- » Wymaga większych zasobów (RAM + CPU)
- » Potencjalnie może powodować błąd przepełnienia stosu

Dziękuję