Podejście obiektowe, przeciążenia operatorów, referencje

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Zakład Podstaw Cybernetyki i Robotyki
Instytut Informatyki, Automatyki i Robotyki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Niniejsza prezentacja została wykonana przy użyciu systemu składu LaTeX *oraz stylu* `beamer`, *którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:

`http://latex-beamer.sourceforge.net`

Spis treści

Spis treści

Podejście obiektowe

Programowanie obiektowe oparte jest na *podejściu obiektowym* do analizy problemu oraz syntezy i implementacji jego rozwiązania.

Podejście obiektowe

Programowanie obiektowe oparte jest na *podejściu obiektowym* do analizy problemu oraz syntezy i implementacji jego rozwiązania.

Podejście obiektowe bazuje na fundamentalnej cesze aktywności intelektualnej, która pozwala ludziom (i nie tylko) wyróżniać odrębne obiekty w swoim otoczeniu, przypisywać im własności oraz określać sposób ich interakcji między sobą i otoczeniem.

Podejście obiektowe

Programowanie obiektowe oparte jest na *podejściu obiektowym* do analizy problemu oraz syntezy i implementacji jego rozwiązania.

Podejście obiektowe bazuje na fundamentalnej cesze aktywności intelektualnej, która pozwala ludziom (i nie tylko) wyróżniać odrębne obiekty w swoim otoczeniu, przypisywać im własności oraz określać sposób ich interakcji między sobą i otoczeniem.

Wyróżnianie obiektów może być dokonywane na różne sposoby. Oparte jest ono na obserwacji i wcześniejszej wiedzy.

Podejście obiektowe

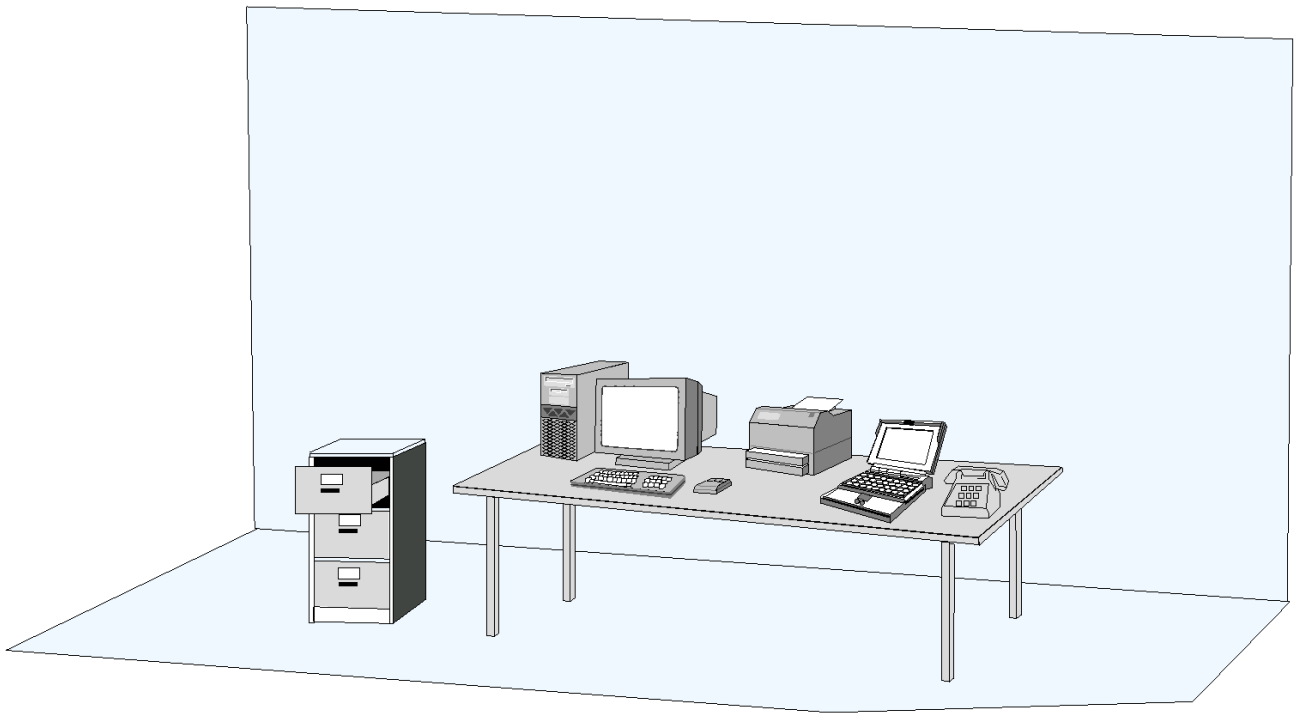
Programowanie obiektowe oparte jest na *podejściu obiektowym* do analizy problemu oraz syntezy i implementacji jego rozwiązania.

Podejście obiektowe bazuje na fundamentalnej cesze aktywności intelektualnej, która pozwala ludziom (i nie tylko) wyróżniać odrębne obiekty w swoim otoczeniu, przypisywać im własności oraz określać sposób ich interakcji między sobą i otoczeniem.

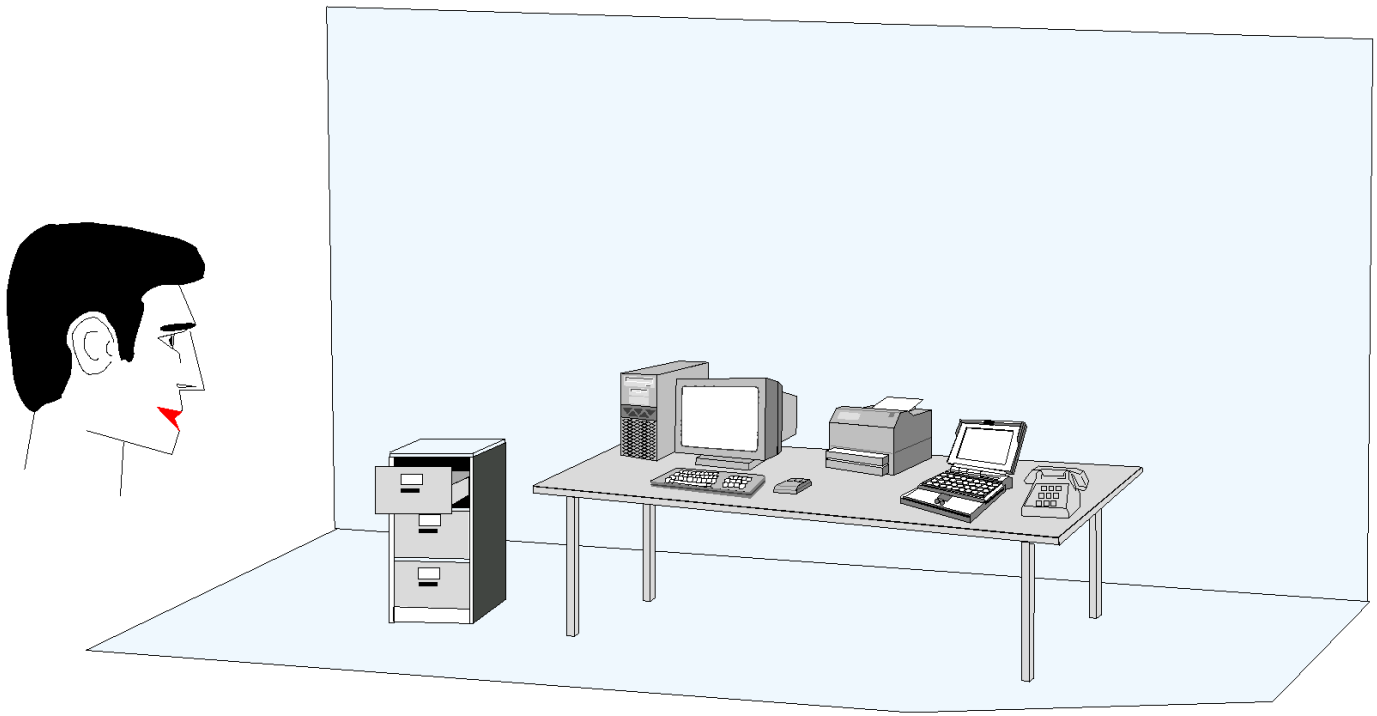
Wyróżnianie obiektów może być dokonywane na różne sposoby. Oparte jest ono na obserwacji i wcześniejszej wiedzy.

Obiektowi lub obiektom jesteśmy w stanie przypisać *pojęcia*. Proces ten nazywa się ***postrzeganiem***.

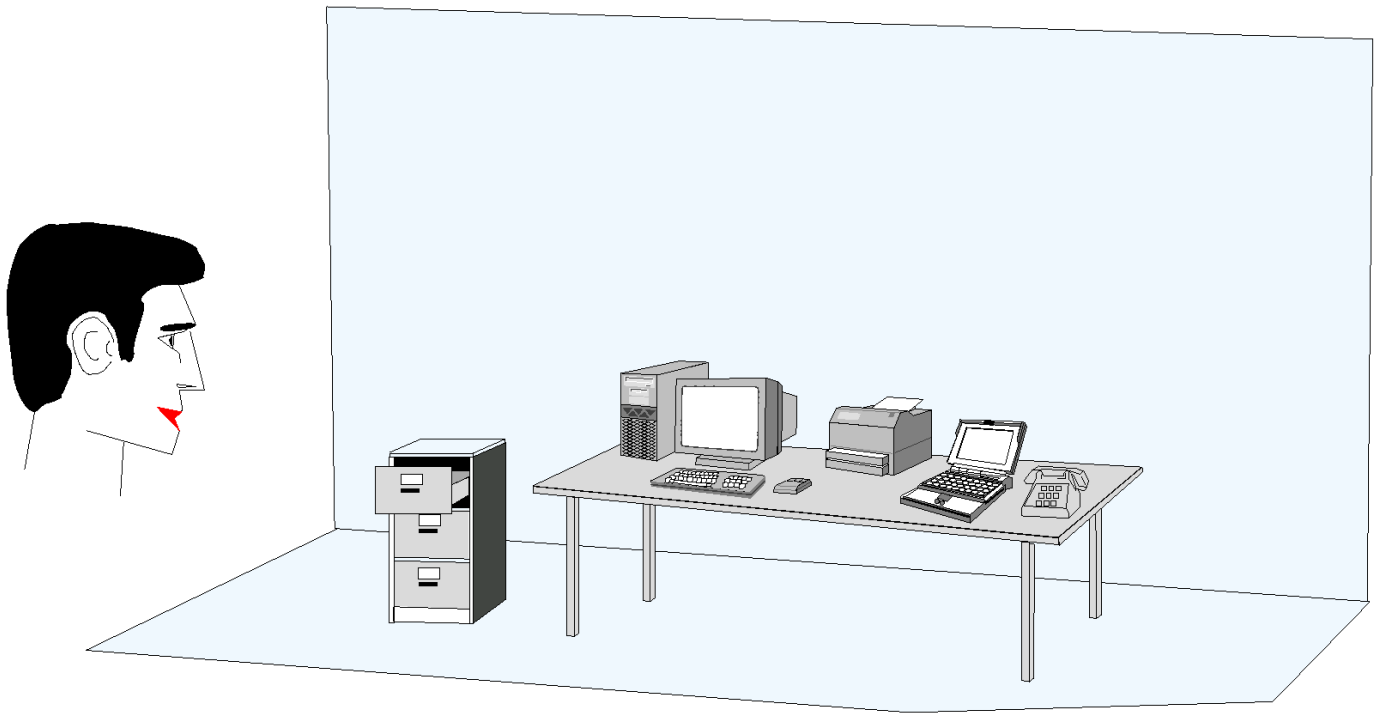
Postrzeganie



Postrzeganie

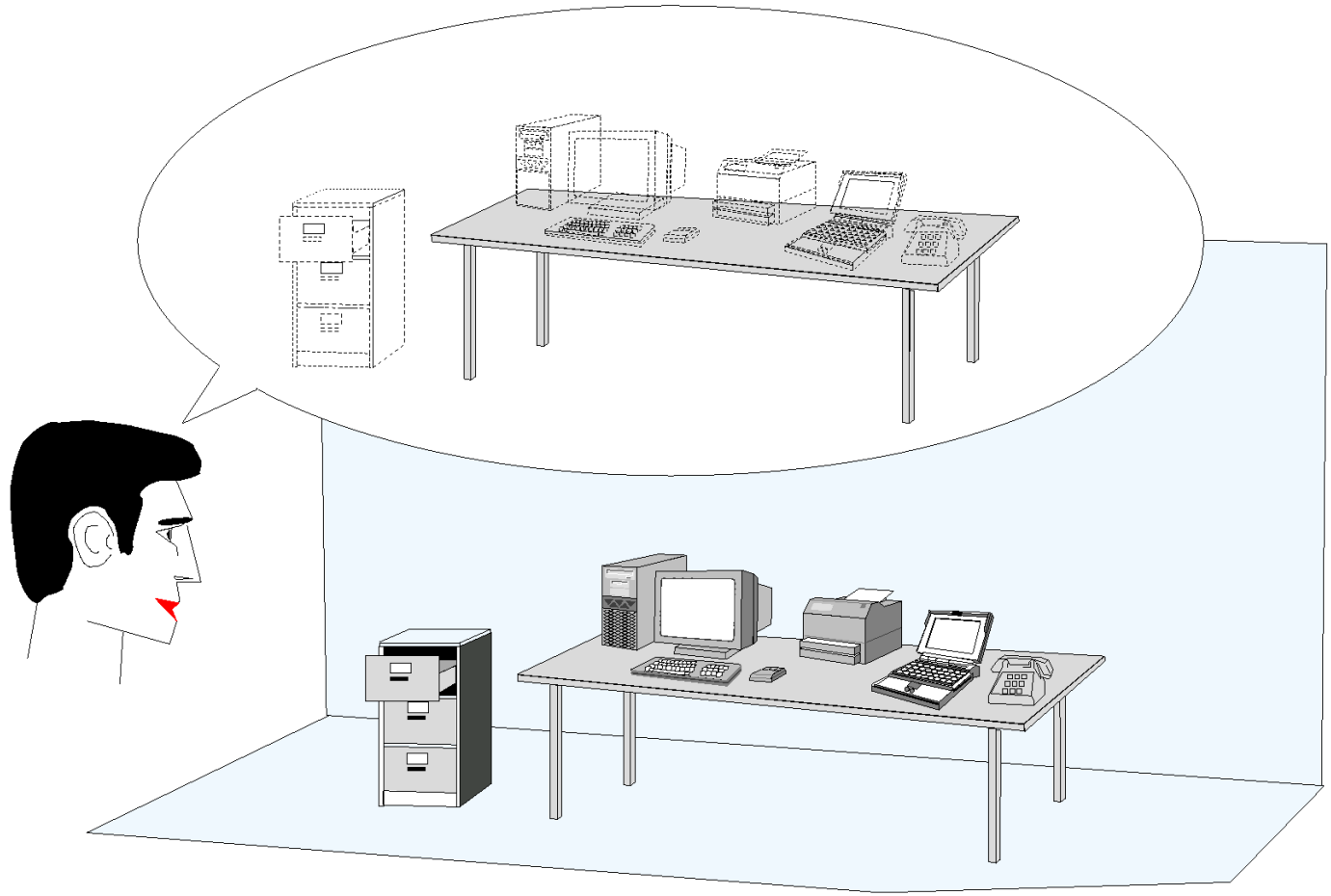


Postrzeganie



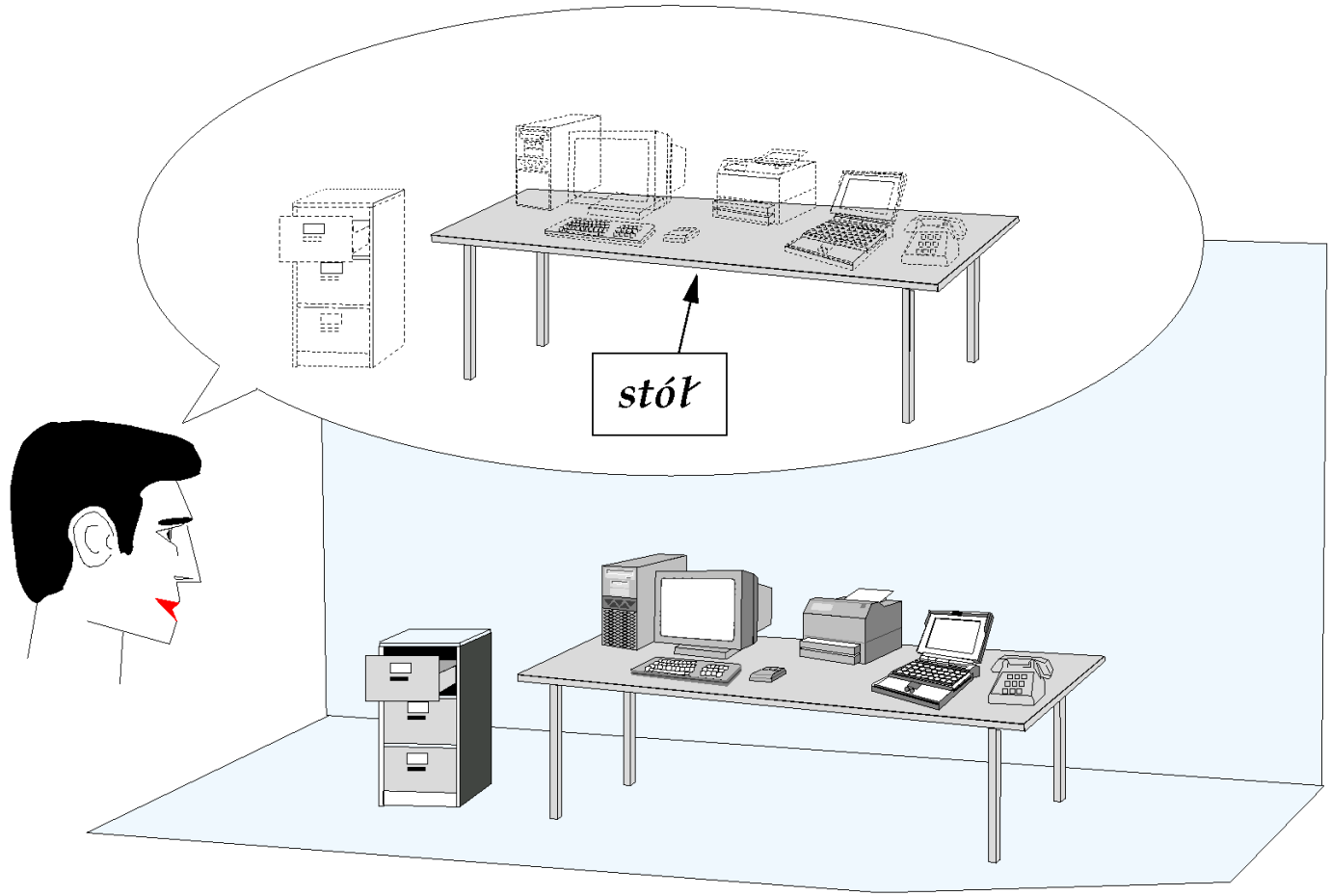
Obserwując otoczenie jesteśmy w stanie wyodrębnić przedmioty . . .

Postrzeganie



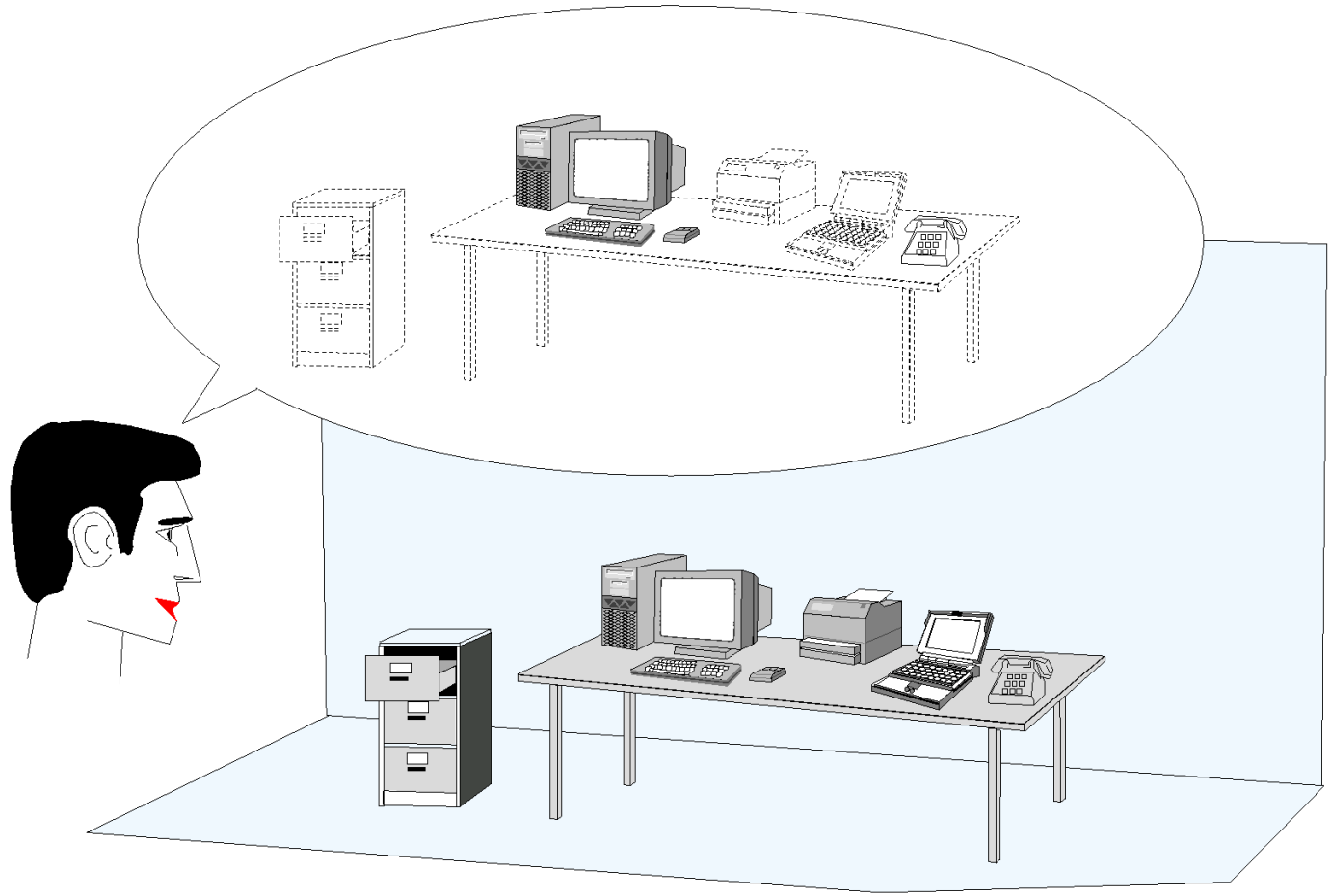
Obserwując otoczenie jesteśmy w stanie wyodrębnić przedmioty . . .

Postrzeganie



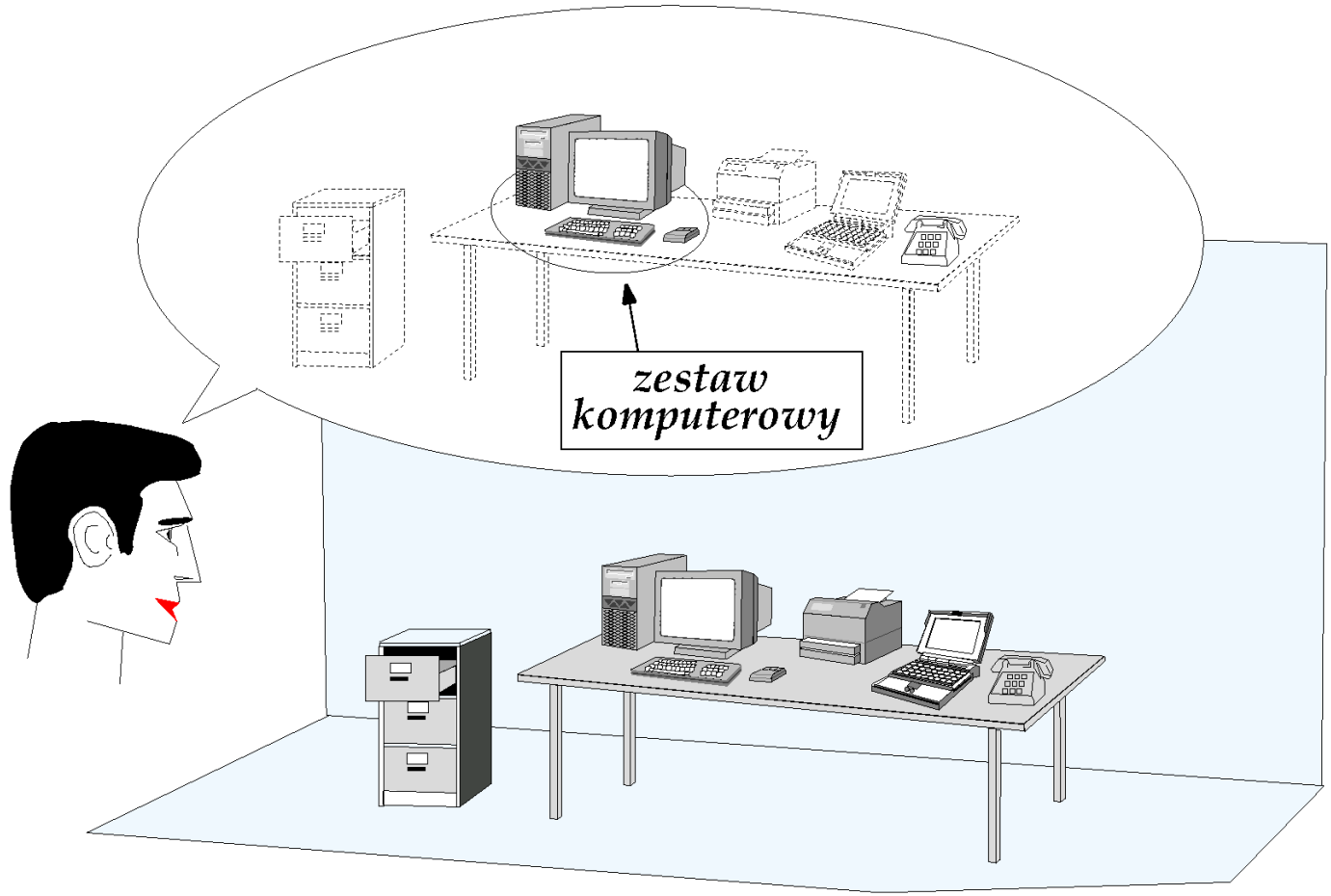
Obserwując otoczenie jesteśmy w stanie wyodrębnić przedmioty i przypisać im pojęcia.

Postrzeganie



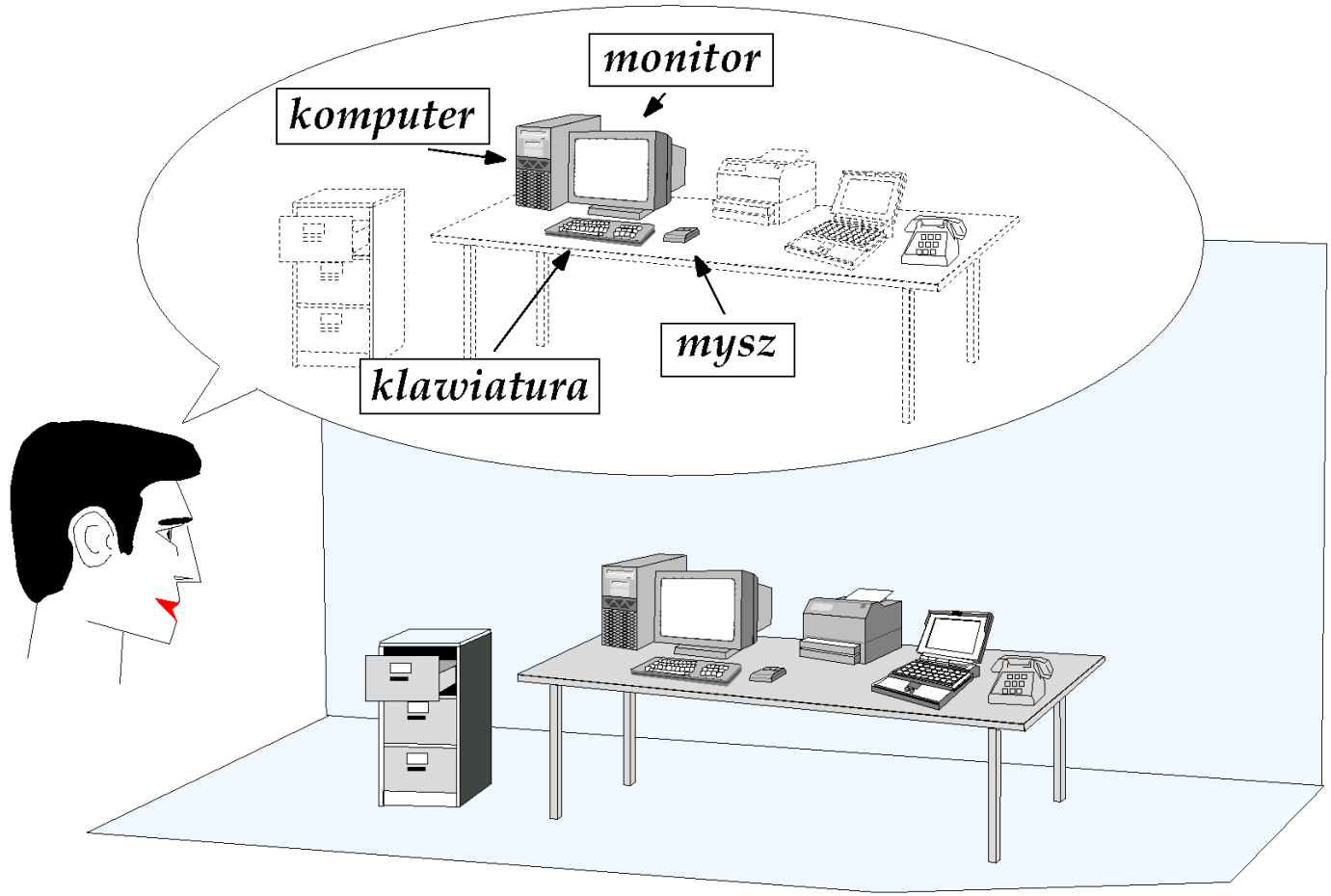
Wyodrębnieniu może podlegać zestaw elementów, jako osobna całość.

Postrzeganie



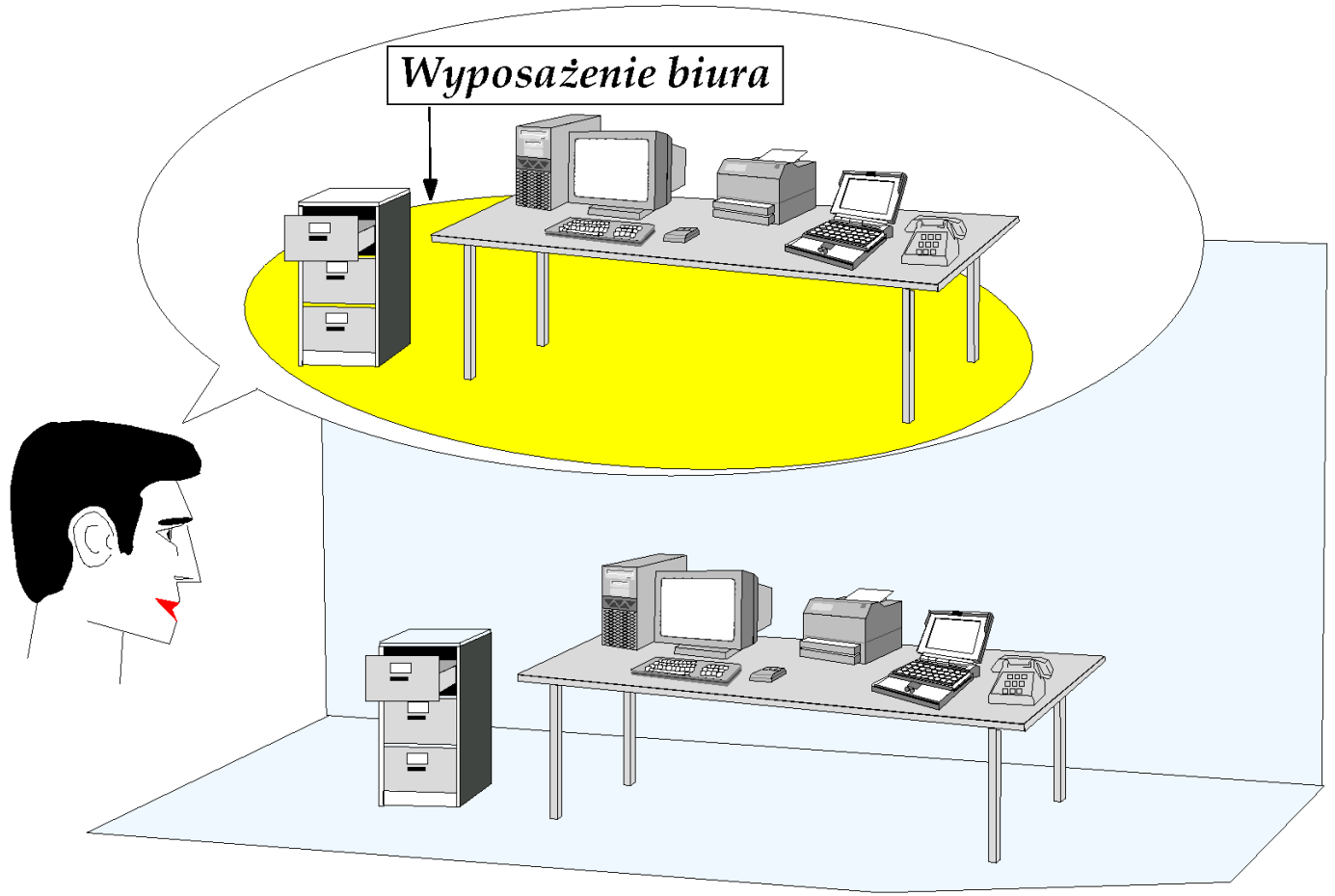
Pojęcie odnosi się wówczas do zbioru elementów między którymi zachodzą odpowiednie relacje.

Postrzeganie



Każdemu z elementów może być osobno wyróżniony poprzez przypisanie mu indywidualnego pojęcia.

Postrzeganie



Znajdując cechy wspólne wszystkich elementów możemy również przyporządkować *pojęcie* ich zbiorowi.

Spis treści

Czym jest pojęcie

Istotnym elementem aktu *wyróżnienia* jakiegoś tworu lub wyobrażenia abstrakcyjnego jest przypisanie mu pewnego ***pojęcia***.

Czym jest pojęcie

Istotnym elementem aktu *wyróżnienia* jakiegoś tworu lub wyobrażenia abstrakcyjnego jest przypisanie mu pewnego ***pojęcia***.

Pojęcie jest wyobrażeniem lub oznaczeniem, które stosujemy do rzeczy lub wyobrażeń abstrakcyjnych.

Czym jest pojęcie

Istotnym elementem aktu *wyróżnienia* jakiegoś tworu lub wyobrażenia abstrakcyjnego jest przypisanie mu pewnego ***pojęcia***.

Pojęcie jest wyobrażeniem lub oznaczeniem, które stosujemy do rzeczy lub wyobrażeń abstrakcyjnych.

Przypisywanie *pojęć* jest możliwe dzięki rozpoznaniu własności wspólnych dla reprezentantów zbiorów, do których stosuje się dane *pojęcie*.

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Przykłady *pojęć*:

materialne	niematerialne	relacyjne	zdarzenia	inne

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Przykłady *pojęć*:

materialne	niematerialne	relacyjne	zdarzenia	inne
pojazd budynek atom				

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Przykłady *pojęć*:

materialne	niematerialne	relacyjne	zdarzenia	inne
pojazd budynek atom	czas poprawność firma			

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Przykłady *pojęć*:

materialne	niematerialne	relacyjne	zdarzenia	inne
pojazd budynek atom	czas poprawność firma	posiadanie przynależność małżeństwo		

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Przykłady *pojęć*:

materialne	niematerialne	relacyjne	zdarzenia	inne
pojazd budynek atom	czas poprawność firma	posiadanie przynależność małżeństwo	spotkanie zakup wyjazd	

Czym jest pojęcie

Przyswojenie sobie zbioru *pojęć* pozwala nadawać znaczenie obiektom znajdującym się w naszym otoczeniu.

Przykłady *pojęć*:

materialne	niematerialne	relacyjne	zdarzenia	inne
pojazd budynek atom	czas poprawność firma	posiadanie przynależność małżeństwo	spotkanie zakup wyjazd	wzorowy nietypowy ikona

Czym jest pojęcie

Termin ***pojęcie*** zawiera:

- **intensję** – treść pojęcia
- **ekstensję** – zakres pojęcia

Czym jest pojęcie

Termin ***pojęcie*** zawiera:

- **intensję** – treść pojęcia
- **ekstensję** – zakres pojęcia

Czym jest pojęcie

Termin ***pojęcie*** zawiera:

- **intensję** – treść pojęcia
- **ekstensję** – zakres pojęcia

Czym jest pojęcie

Termin ***pojęcie*** zawiera:

- **intensję** – treść pojęcia
- **ekstensję** – zakres pojęcia

Intensja jest pełną definicją pojęcia i testu określającego, czy dane pojęcie odnosi się do danej rzeczy lub wyobrażenia abstrakcyjnego.

Czym jest pojęcie

Termin ***pojęcie*** zawiera:

- **intensję** – treść pojęcia
- **ekstensję** – zakres pojęcia

Intensja jest pełną definicją pojęcia i testu określającego, czy dane pojęcie odnosi się do danej rzeczy lub wyobrażenia abstrakcyjnego.

Ekstensja jest zbiorem wszystkich rzeczy i wyobrażeń abstrakcyjnych, do których stosuje się dane pojęcie.

Czym jest pojęcie

Termin ***pojęcie*** zawiera:

- **intensję** – treść pojęcia
- **ekstensję** – zakres pojęcia

Intensja jest pełną definicją pojęcia i testu określającego, czy dane pojęcie odnosi się do danej rzeczy lub wyobrażenia abstrakcyjnego.

Ekstensja jest zbiorem wszystkich rzeczy i wyobrażeń abstrakcyjnych, do których stosuje się dane pojęcie.

$$\textit{Trójka pojęciowa} = (\text{nazwa, intensja, ekstensja})$$

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Robot przemysłowy

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Robot przemysłowy

Przykłady trójek pojęciowych

Trójka pojęciowa = (*nazwa*, intensja, ekstensja)

Robot przemysłowy

nazwa
Robot przemysłowy

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, *intensja*, ekstensja)

Robot przemysłowy

nazwa
Robot przemysłowy

intensja
Maszyna manipulacyjna sterowana automatycznie za pomocą sygnałów generowanych w programowalnym układzie sterowania.

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, *ekstensja*)

Robot przemysłowy

nazwa
Robot przemysłowy

intensja
Maszyna manipulacyjna sterowana automatycznie za pomocą sygnałów generowanych w programowalnym układzie sterowania.

ekstensja
iRb-6, IRB1400, Puma 560, ...

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Robot przemysłowy

nazwa
Robot przemysłowy

intensja
Maszyna manipulacyjna sterowana automatycznie za pomocą sygnałów generowanych w programowalnym układzie sterowania.

ekstensja
iRb-6, IRB1400, Puma 560, ...

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Idealny człowiek

nazwa

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (*nazwa*, intensja, ekstensja)

Idealny człowiek

nazwa
Idealny człowiek

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, *intensja*, ekstensja)

Idealny człowiek

nazwa
Idealny człowiek

intensja
Uczciwy, rzetelny, . . .

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, *ekstensja*)

Idealny człowiek

nazwa
Idealny człowiek

intensja
Uczciwy, rzetelny, . . .

ekstensja


Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Idealny człowiek

nazwa
Idealny człowiek

intensja
Uczciwy, rzetelny, . . .

ekstensja


Niektóre *pojęcia* mogą nie mieć swoich reprezentantów.

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

40S25

nazwa

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (*nazwa*, intensja, ekstensja)

40S25

nazwa
40S25

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, *intensja*, ekstensja)

40S25

nazwa
40S25

intensja


ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, *ekstensja*)

40S25

nazwa
40S25

intensja


ekstensja
40S25, 40S25, . . .

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

40S25

nazwa
40S25

intensja
×

ekstensja
40S25, 40S25, . . .

Pojęcie może nie mieć swojej definicji. Przykład układu scalonego, którego dokumentacja i opis zostały zagubione.

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient

nazwa

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (*nazwa*, intensja, ekstensja)

Klient

nazwa
Klient

intensja

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, *intensja*, ekstensja)

Klient

nazwa
Klient

intensja
Osoba lub organizacja kupująca dobra lub usługi.

ekstensja

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, *ekstensja*)

Klient

nazwa
Klient

intensja
Osoba lub organizacja kupująca dobra lub usługi.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient

nazwa
Klient

intensja
Osoba lub organizacja kupująca dobra lub usługi.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient, Interesant

nazwa
Klient, Interesant

intensja
Osoba lub organizacja kupująca dobra lub usługi.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient, Interesant

nazwa
Klient, Interesant

intensja
Osoba lub organizacja kupująca dobra lub usługi.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego

Pojęcia mogą mieć synonimy.

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient

nazwa
Klient

intensja
Osoba lub organizacja kupująca dobra lub usługi.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient

nazwa
Klient

intensja
Osoba lub organizacja kupująca dobra lub usługi.
Aplikacja programowa, która żąda od innej aplikacji realizacji usług.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego
`xclock`, `xterm`, ...

Przykłady trójek pojęciowych

Trójka pojęciowa = (nazwa, intensja, ekstensja)

Klient

nazwa
Klient

intensja
Osoba lub organizacja kupująca dobra lub usługi.
Aplikacja programowa, która żąda od innej aplikacji realizacji usług.

ekstensja
Jan Kowalski, Firma Jana Kowalskiego
`xclock`, `xterm`, ...

Pojęcia mogą mieć homonimy.

Spis treści

Obiekty

Obiekt ?! Co to takiego ???

Obiekty

Obiektem jest to coś, do czego da się zastosować jakieś *pojęcie*. Tak więc *obiekt* jest egzemplarzem pojęcia.

Obiekty

Obiektem jest to coś, do czego da się zastosować jakieś *pojęcie*. Tak więc *obiekt* jest egzemplarzem pojęcia.

Pojęcie $\Longrightarrow$ robot przemysłowy

Obiekty

Obiektem jest to coś, do czego da się zastosować jakieś *pojęcie*. Tak więc *obiekt* jest egzemplarzem pojęcia.

Pojęcie $\Longrightarrow$ robot przemysłowy
Obiekt $\Longrightarrow$ konkretny egzemplarz robota, np. robota IRB1400

Obiekty

Własności:

- *Obiekt* może mieć cechy, którym przypisywane są nazwy, np. kulistość.
- *Obiekt* może mieć atrybuty, np. promień kuli.
- *Obiektowi* możemy przyporządkować **stan**. **Stan** obiektu jest kolekcją atrybutów i związków dotyczących danego obiektu. Zmiana **stanu** jest zmianą atrybutu i/lub związków danego obiektu (np. położenie obiektu – x, y, z).
- *Obiekt* może mieć pewien ograniczony czas życia.
- *Obiekt* może być powiązany z innymi obiektami poprzez odwzorowania lub relacje. Odwzorowania i relacje mogą także być modelowane jako obiekty.

Obiekty

Własności:

- *Obiekt* może mieć cechy, którym przypisywane są nazwy, np. kulistość.
- *Obiekt* może mieć atrybuty, np. promień kuli.
- *Obiektowi* możemy przyporządkować **stan**. **Stan** obiektu jest kolekcją atrybutów i związków dotyczących danego obiektu. Zmiana **stanu** jest zmianą atrybutu i/lub związków danego obiektu (np. położenie obiektu – x, y, z).
- *Obiekt* może mieć pewien ograniczony czas życia.
- *Obiekt* może być powiązany z innymi obiektami poprzez odwzorowania lub relacje. Odwzorowania i relacje mogą także być modelowane jako obiekty.

Obiekty

Własności:

- *Obiekt* może mieć cechy, którym przypisywane są nazwy, np. kulistość.
- *Obiekt* może mieć atrybuty, np. promień kuli.
- *Obiektowi* możemy przyporządkować **stan**. **Stan** obiektu jest kolekcją atrybutów i związków dotyczących danego obiektu. Zmiana **stanu** jest zmianą atrybutu i/lub związków danego obiektu (np. położenie obiektu – x, y, z).
- *Obiekt* może mieć pewien ograniczony czas życia.
- *Obiekt* może być powiązany z innymi obiektami poprzez odwzorowania lub relacje. Odwzorowania i relacje mogą także być modelowane jako obiekty.

Obiekty

Własności:

- *Obiekt* może mieć cechy, którym przypisywane są nazwy, np. kulistość.
- *Obiekt* może mieć atrybuty, np. promień kuli.
- *Obiektowi* możemy przyporządkować **stan**. **Stan** obiektu jest kolekcją atrybutów i związków dotyczących danego obiektu. Zmiana **stanu** jest zmianą atrybutu i/lub związków danego obiektu (np. położenie obiektu – x, y, z).
- *Obiekt* może mieć pewien ograniczony czas życia.
- *Obiekt* może być powiązany z innymi obiektami poprzez odwzorowania lub relacje. Odwzorowania i relacje mogą także być modelowane jako obiekty.

Obiekty

Własności:

- *Obiekt* może mieć cechy, którym przypisywane są nazwy, np. kulistość.
- *Obiekt* może mieć atrybuty, np. promień kuli.
- *Obiektowi* możemy przyporządkować **stan**. **Stan** obiektu jest kolekcją atrybutów i związków dotyczących danego obiektu. Zmiana **stanu** jest zmianą atrybutu i/lub związków danego obiektu (np. położenie obiektu – x, y, z).
- *Obiekt* może mieć pewien ograniczony czas życia.
- *Obiekt* może być powiązany z innymi obiektami poprzez odwzorowania lub relacje. Odwzorowania i relacje mogą także być modelowane jako obiekty.

Obiekty

Własności:

- *Obiekt* może mieć cechy, którym przypisywane są nazwy, np. kulistość.
- *Obiekt* może mieć atrybuty, np. promień kuli.
- *Obiektowi* możemy przyporządkować **stan**. **Stan** obiektu jest kolekcją atrybutów i związków dotyczących danego obiektu. Zmiana **stanu** jest zmianą atrybutu i/lub związków danego obiektu (np. położenie obiektu – x, y, z).
- *Obiekt* może mieć pewien ograniczony czas życia.
- *Obiekt* może być powiązany z innymi obiektami poprzez odwzorowania lub relacje. Odwzorowania i relacje mogą także być modelowane jako obiekty.

Spis treści

Przeciążanie funkcji

Przeciążanie funkcji (ang. *function overloading*) pozwala na definiowanie funkcji o tych samych nazwach, które różnią się ilością lub typami parametrów. Mogą (lecz nie muszą) zwracać wartości różnego typu. Mechanizm ten pozwala stosować funkcję realizującej ten sam typ operacji w różnych kontekstach.

Przeciążanie funkcji

Przeciążanie funkcji (ang. *function overloading*) pozwala na definiowanie funkcji o tych samych nazwach, które różnią się ilością lub typami parametrów. Mogą (lecz nie muszą) zwracać wartości różnego typu. Mechanizm ten pozwala stosować funkcję realizującej ten sam typ operacji w różnych kontekstach.

Przykład:

```
void Wyswietl( float Liczba )
{
    cout << "Liczba: " << Liczba << endl;
}
```

Przeciążanie funkcji

Przeciążanie funkcji (ang. *function overloading*) pozwala na definiowanie funkcji o tych samych nazwach, które różnią się ilością lub typami parametrów. Mogą (lecz nie muszą) zwracać wartości różnego typu. Mechanizm ten pozwala stosować funkcję realizującej ten sam typ operacji w różnych kontekstach.

Przykład:

```
void Wyswietl( float Liczba )
{
    cout << "Liczba: " << Liczba << endl;
}

void Wyswietl( const char * Napis )
{
    cout << "Napis: \"" << Napis << "\"" << endl;
}
```

Przeciążanie funkcji

Przykład:

```
void  Wyswietl( float  Liczba )
{
    cout << "Liczba: " << Liczba << endl;
}

void  Wyswietl( const  char * Napis )
{
    cout << "Napis: \"" << Napis << "\"" << endl;
}


void  main( )
{
    Wyswietl(123.14);
    Wyswietl("Jak dobrze wstać skoro świt.");
}
```

Przeciążanie funkcji

Przykład:

```
void Wyswietl( float Liczba )
{
    cout << "Liczba: " << Liczba << endl;
}

void Wyswietl( const char * Napis )
{
    cout << "Napis: \"" << Napis << "\"" << endl;
}


void main( )
{
    Wyswietl(123.14);
    Wyswietl("Jak dobrze wstać skoro świt.");
}
```

Przeciążanie funkcji

Przykład:

```
void Wyswietl( float Liczba )
{
    cout << "Liczba: " << Liczba << endl;
}

void Wyswietl( const char * Napis )
{
    cout << "Napis: \"" << Napis << "\"" << endl;
}


void main( )
{
    Wyswietl(123.14);
    Wyswietl("Jak dobrze wstać skoro świt.");
}
```

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeładowanie — *Jak to rozumieć?*

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeładowanie — (1) do istniejącego ładunku dodajemy nowy przekraczając dopuszczalną obciążalność lub pojemność.

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeładowanie — (2) istniejącą zawartość wyładowujemy i na jej miejsce wprowadzamy nowy ładunek.

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeciążenie — *A to jak rozumić?*

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeciążenie — (1) do istniejącego ładunku dodajemy nowy przekraczając dopuszczalny ciężar.

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeciążenie — (2) do istniejących obowiązków (zadań, powinności itd.) dodajemy nowe przekraczając nominalny zakres.

Nieszczęsne słowo: overload

Przeładowanie czy przeciążenie?

Przeładowanie — (1) do istniejącego ładunku dodajemy nowy przekraczając dopuszczalną obciążalność lub pojemność.

Przeładowanie — (2) istniejącą zawartość wyładowujemy i na jej miejsce wprowadzamy nowy ładunek.

Przeciążenie — (1) do istniejącego ładunku dodajemy nowy przekraczając dopuszczalny ciężar.

Przeciążenie — (2) do istniejących obowiązków (zadań, powinności itd.) dodajemy nowe przekraczając nominalny zakres.

Spis treści

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 )
{
    LZespolona Wynik;
    Wynik.re = Skl1.re + Skl2.re;
    Wynik.im = Skl1.im + Skl2.im;
    return Wynik;
}

int main( )
{
    LZespolona Z1, Z2, Wynik;

    Wynik = Dodaj(Z1, Z2);
}
```

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 )
{
    LZespolona Wynik;
    Wynik.re = Skl1.re + Skl2.re;
    Wynik.im = Skl1.im + Skl2.im;
    return Wynik;
}

int main( )
{
    LZespolona Z1, Z2, Wynik;
    Wynik = Dodaj(Z1, Z2);
}
```

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 )
{
    LZespolona Wynik;
    Wynik.re  = Skl1.re + Skl2.re;
    Wynik.im = Skl1.im + Skl2.im;
    return  Wynik;
}

int main( )
{
    LZespolona   Z1, Z2, Wynik;

    Wynik = Dodaj(Z1, Z2);
}
```

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
{
    LZespolona Wynik;
    Wynik.re  = Skl1.re + Skl2.re;
    Wynik.im = Skl1.im + Skl2.im;
    return Wynik;
}

int main( )
{
    LZespolona Z1, Z2, Wynik;

    Wynik = operator + (Z1, Z2);
}
```

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
{
    LZespolona Wynik;
    Wynik.re  = Skl1.re + Skl2.re;
    Wynik.im = Skl1.im + Skl2.im;
    return Wynik;
}

int main( )
{
    LZespolona  Z1, Z2, Wynik;

    wynik = Z1 + Z2;
}
```

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
```

Nie możemy definiować funkcji operatorowych, gdy ich wszystkie parametry są typów wbudowanych takich jak **float**, **int**, itd. np.

```
int operator + (int arg1, float arg2)
{
  ...
}
```

Dla tych typów definicje tych operacji są już *wbudowane* w kompilator.

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
```

Uwaga: Wbudowanych definicji operatorów dodawania nie można jawnie wywołać tak jak operacji np. dla symboli, tzn.

```
Liczba = operator + (10, 2);
```

Arytmetyka liczb zespolonych

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
```

Uwaga: Wbudowanych definicji operatorów dodawania nie można jawnie wywołać tak jak operacji np. dla symboli, tzn.

~~Liczba = operator + (10, 2);~~

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona   Z1, Z2, Wynik;

    wynik = Z1 + Z2;
}
```

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona   Z1, Z2, Wynik;

    Wynik = Z1 + +Z2;
}
```

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona    Z1, Z2, Wynik;

    Wynik = Z1 + (+Z2);
}
```

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona    Z1, Z2, Wynik;

    Wynik = Z1 + operator+(Z2);
}
```

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...
LZespolona operator + ( LZespolona Arg)
 {
    ???
 }

 int main( )
{
    LZespolona   Z1, Z2, Wynik;

    Wynik = Z1 + operator+(Z2);
 }
```

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...
LZespolona operator + ( LZespolona Arg)
 {
   return Arg;
 }

 int main( )
{
    LZespolona Z1, Z2, Wynik;

    Wynik = Z1 + operator+(Z2);
 }
```

Przeciążanie operatora +

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...
LZespolona operator + ( LZespolona Arg)
 {
    return Arg;
 }

 int main( )
{
    LZespolona Z1, Z2, Wynik;

    Wynik = Z1 + +Z2;
 }
```

Przeciążanie operatora –

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona   Z1, Z2, Wynik;

    wynik = Z1 + –Z2;
}
```

Przeciążanie operatora −

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona   Z1, Z2, Wynik;

    Wynik = Z1 + -Z2;
}
```

Przeciążanie operatora −

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona  Z1, Z2, Wynik;

    Wynik = Z1 + (-Z2);
}
```

Przeciążanie operatora –

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...

int main( )
{
    LZespolona    Z1, Z2, Wynik;

    Wynik = Z1 + operator–(Z2);
}
```

Przeciążanie operatora –

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...
LZespolona operator – ( LZespolona Arg)
 {
    ???
 }

 int main( )
{
    LZespolona   Z1, Z2, Wynik;

    Wynik = Z1 + operator–(Z2);
 }
```

Przeciążanie operatora –

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...
LZespolona operator – ( LZespolona Arg)
 {
   Arg.re = –Arg.re; Arg.im = –Arg.im; return Arg;
 }

 int main( )
{
    LZespolona Z1, Z2, Wynik;

    Wynik = Z1 + operator–(Z2);
 }
```

Przeciążanie operatora –

```
struct LZespolona {
    double re, im;
};

LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 )
 ...
LZespolona operator – ( LZespolona Arg)
 {
   Arg.re = –Arg.re; Arg.im = –Arg.im; return Arg;
 }

 int main( )
{
    LZespolona Z1, Z2, Wynik;

    Wynik = Z1 + –Z2;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
void  Dodaj( LZespolona  Arg1, LZespolona  Arg2 );
LZespolona  operator + ( LZespolona  Arg1, LZespolona  Arg2 );
 ...

int  main( )
{
     LZespolona    Wynik, Z1;



     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
void  Dodaj( LZespolona  Arg1, LZespolona  Arg2 );
LZespolona  operator + ( LZespolona  Arg1, LZespolona  Arg2 );
 ...

int  main( )
{
     LZespolona    Wynik, Z1;



     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona { double re, im; };
...
void Dodaj( LZespolona Arg1, LZespolona Arg2 );

LZespolona operator + ( LZespolona Arg1, LZespolona Arg2 );
...

int main( )
{
    LZespolona Wynik, Z1;


    Wynik += Z1;
}
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
void  Dodaj( LZespolona  Arg1, LZespolona  Arg2 );

void  operator += ( LZespolona  Arg1, LZespolona  Arg2 );
 ...

int  main( )
{
     LZespolona    Wynik, Z1;


     Wynik += Z1;
}
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
void  Dodaj( LZespolona  Arg1, LZespolona  Arg2 );

void  operator += ( LZespolona  Arg1, LZespolona  Arg2 );
 ...

int  main( )
{
     LZespolona    Wynik, Z1;


     operator+= (Wynik, Z1);
     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
LZespolona  DodajIPodstaw( LZespolona  Arg1, LZespolona  Arg2 );
void  operator += ( LZespolona  Arg1, LZespolona  Arg2 );
 ...

int  main( )
{
     LZespolona    Wynik, Z1;


     operator+= (Wynik, Z1);
     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona { double re, im; };
 ...
LZespolona DodajIPodstaw( LZespolona Arg1, LZespolona Arg2 );
void operator += ( LZespolona Arg1, LZespolona Arg2 );
 ...

int main( )
{
    LZespolona Wynik, Z1;

    DodajIPodstaw(Wynik, Z1);
    operator+= (Wynik, Z1);
    Wynik += Z1;
}
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
LZespolona  DodajIPodstaw( LZespolona  Arg1, LZespolona  Arg2 );
 {
     return  Arg1 = Arg1 + Arg2;
 }
 ...
int  main( )
{
     LZespolona     Wynik, Z1;

     DodajIPodstaw(Wynik, Z1);
     operator+= (Wynik, Z1);
     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
LZespolona  DodajIPodstaw( LZespolona  Arg1, LZespolona  Arg2 );
 {
     return  Arg1 = Arg1 + Arg2;
 }
 ...
int  main( )
{
     LZespolona     Wynik, Z1;

     DodajIPodstaw(Wynik, Z1);
     operator+= (Wynik, Z1);
     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
LZespolona  DodajIPodstaw( LZespolona  *pArg1, LZespolona  Arg2 );
 {
     return  *pArg1 = *pArg1 + Arg2;
 }
 ...
int  main( )
{
     LZespolona     Wynik, Z1;

     DodajIPodstaw(&Wynik, Z1);
     operator+= (Wynik, Z1);
     Wynik += Z1;
 }
```

A jak zdefiniować operację dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
LZespolona  DodajIPodstaw( LZespolona  *pArg1, LZespolona  Arg2 );
 {
     return  *pArg1 = *pArg1 + Arg2;
 }
 ...
int  main( )
{
     LZespolona     Wynik, Z1;

     DodajIPodstaw(&Wynik, Z1);
     operator+= (&Wynik, Z1);
     &Wynik += Z1;
 }
```

Spis treści

O referencji

Referencja w C++ jest pochodnym typem danych (podobnie jak wskaźnik). Umożliwia pośrednie odwoływanie się do zmiennych i obiektów.

Kłopoty z terminem *referencja*

- w informatyce przez referencję rozumie się pewną ogólną koncepcję typu danych. Określa się w ten sposób obiekt/zmienną zawierającą informację będącą odsyłaczem do właściwej struktury danych. Zmienne wskaźnikowe w C sa przykładem realizacji koncepcji referencji.
- w C++ referencja jest konkretną implementacją typu danych, który jest istotnie różny od typu wskaźnikowego.

O referencji

Referencja w C++ jest pochodnym typem danych (podobnie jak wskaźnik). Umożliwia pośrednie odwoływanie się do zmiennych i obiektów.

Kłopoty z terminem *referencja*

- w informatyce przez referencję rozumie się pewną ogólną koncepcję typu danych. Określa się w ten sposób obiekt/zmienną zawierającą informację będącą odsyłaczem do właściwej struktury danych. Zmienne wskaźnikowe w C sa przykładem realizacji koncepcji referencji.
- w C++ referencja jest konkretną implementacją typu danych, który jest istotnie różny od typu wskaźnikowego.

O referencji

Referencja w C++ jest pochodnym typem danych (podobnie jak wskaźnik). Umożliwia pośrednie odwoływanie się do zmiennych i obiektów.

Kłopoty z terminem *referencja*

- w informatyce przez referencję rozumie się pewną ogólną koncepcję typu danych. Określa się w ten sposób obiekt/zmienną zawierającą informację będącą odsyłaczem do właściwej struktury danych. Zmienne wskaźnikowe w C sa przykładem realizacji koncepcji referencji.
- w C++ referencja jest konkretną implementacją typu danych, który jest istotnie różny od typu wskaźnikowego.

O referencji

Referencja w C++ jest pochodnym typem danych (podobnie jak wskaźnik). Umożliwia pośrednie odwoływanie się do zmiennych i obiektów.

Kłopoty z terminem *referencja*

- w informatyce przez referencję rozumie się pewną ogólną koncepcję typu danych. Określa się w ten sposób obiekt/zmienną zawierającą informację będącą odsyłaczem do właściwej struktury danych. Zmienne wskaźnikowe w C sa przykładem realizacji koncepcji referencji.
- w C++ referencja jest konkretną implementacją typu danych, który jest istotnie różny od typu wskaźnikowego.

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

```
int    Licznik;
```


Dla przykładu rozważmy deklarację zmiennej typu **int**.

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

```
int      Licznik;
int* const  wskLicznik = &Licznik;
```

Możemy utworzyć stały wskaźnik na zmienną.

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

```
int     Licznik;
int* const  wskLicznik = &Licznik;

*wskLicznik = 15;
cerr << Licznik;
```

Co się wyświetli po wykonaniu przedstawionych instrukcji?

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

```
int     Licznik;
int* const  wskLicznik = &Licznik;

*wskLicznik = 15;
cerr << Licznik;
```

```
int     Licznik;
int     &refLicznik = Licznik;
```

Definiując referencję (podobnie jak stały wskaźnik) należy ją od razu zainicjalizować

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

```
int     Licznik;
int* const  wskLicznik = &Licznik;

*wskLicznik = 15;
cerr << Licznik;
```

```
int     Licznik;
int     &refLicznik = Licznik;

refLicznik = 15;
cerr << Licznik;
```

Referencją posługujemy się tak samo jak samą zmienną.

Referencja i wskaźnik

W pierwszym przybliżeniu referencja (w C++) może być uważana jako typ wskaźnikowy stały.

```
char     Znak;
char* const  wskZnak = &Znak;

*wskZnak = 'a';
cerr << Znak;
```

```
char     Znak;
char     &refZnak = Znak;

refZnak = 'a';
cerr << Znak;
```

Referencją posługujemy się tak samo jak samą zmienną. Dotyczy to zmiennych każdego typu.

Spis treści

Referencje versus zmienne wskaźnikowe

Cechy wspólne:

- poprzez referencje i wskaźniki odwołujemy się do istniejących zmiennych/obiektów,
- do jednego i tego samego obiektu można utworzyć kilka zmiennych referencyjnych lub zmiennych wskaźnikowych, w których znajduje się adres tego obiektu,
- jeżeli parametr funkcji/metody jest wskaźnikiem lub referencją, to przekazanie zmiennej poprzez ten parametr skutkuje tym, że dokonane modyfikacje wartości zmiennej będą widoczne na *zewnątrz* funkcji/metod.

Referencje versus zmienne wskaźnikowe

Cechy wspólne:

- poprzez referencje i wskaźniki odwołujemy się do istniejących zmiennych/obiektów,
- do jednego i tego samego obiektu można utworzyć kilka zmiennych referencyjnych lub zmiennych wskaźnikowych, w których znajduje się adres tego obiektu,
- jeżeli parametr funkcji/metody jest wskaźnikiem lub referencją, to przekazanie zmiennej poprzez ten parametr skutkuje tym, że dokonane modyfikacje wartości zmiennej będą widoczne na *zewnątrz* funkcji/metod.

Referencje versus zmienne wskaźnikowe

Cechy wspólne:

- poprzez referencje i wskaźniki odwołujemy się do istniejących zmiennych/obiektów,
- do jednego i tego samego obiektu można utworzyć kilka zmiennych referencyjnych lub zmiennych wskaźnikowych, w których znajduje się adres tego obiektu,
- jeżeli parametr funkcji/metody jest wskaźnikiem lub referencją, to przekazanie zmiennej poprzez ten parametr skutkuje tym, że dokonane modyfikacje wartości zmiennej będą widoczne na *zewnątrz* funkcji/metod.

Referencje versus zmienne wskaźnikowe

Cechy wspólne:

- poprzez referencje i wskaźniki odwołujemy się do istniejących zmiennych/obiektów,
- do jednego i tego samego obiektu można utworzyć kilka zmiennych referencyjnych lub zmiennych wskaźnikowych, w których znajduje się adres tego obiektu,
- jeżeli parametr funkcji/metody jest wskaźnikiem lub referencją, to przekazanie zmiennej poprzez ten parametr skutkuje tym, że dokonane modyfikacje wartości zmiennej będą widoczne na *zewnątrz* funkcji/metod.

Referencje versus zmienne wskaźnikowe

Różnice:

- referencje można zainicjalizować adresem zmiennej/obiektu tylko raz, w momencie jej definicji,
- jest niedopuszczalny brak inicjalizacji referencji,
- referencja nie może mieć adresu **nullptr** (NULL – C++98).

Referencje versus zmienne wskaźnikowe

Różnice:

- referencje można zainicjalizować adresem zmiennej/obiektu tylko raz, w momencie jej definicji,
- jest niedopuszczalny brak inicjalizacji referencji,
- referencja nie może mieć adresu **nullptr** (NULL – C++98).

Referencje versus zmienne wskaźnikowe

Różnice:

- referencje można zainicjalizować adresem zmiennej/obiektu tylko raz, w momencie jej definicji,
- jest niedopuszczalny brak inicjalizacji referencji,
- referencja nie może mieć adresu **nullptr** (NULL – C++98).

Referencje versus zmienne wskaźnikowe

Różnice:

- referencje można zainicjalizować adresem zmiennej/obiektu tylko raz, w momencie jej definicji,
- jest niedopuszczalny brak inicjalizacji referencji,
- referencja nie może mieć adresu **nullptr** (NULL – C++98).

Referencje versus zmienne wskaźnikowe

Zalety stosowania referencji:

- referencja jest bezpieczniejszym sposobem odwołania się do zmiennej. Nie daje jednak stuprocentowej pewności, że zmienna/obiekt, do której odwołujemy się przez referencję, istnieje.
- sposób odwołania się do zmiennej/obiektu jest dla programisty *przezroczysty*. Referencję traktujemy tak samo jak zmienną, do której ta referencja odwołuje się.
- odwoływanie się poprzez referencję może pozwalać kompilatorowi na lepszą optymalizację kodu (brak zmian przechowywanego adresu).

Referencje versus zmienne wskaźnikowe

Zalety stosowania referencji:

- referencja jest bezpieczniejszym sposobem odwołania się do zmiennej. Nie daje jednak stuprocentowej pewności, że zmienna/obiekt, do której odwołujemy się przez referencję, istnieje.
- sposób odwołania się do zmiennej/obiektu jest dla programisty *przezroczysty*. Referencję traktujemy tak samo jak zmienną, do której ta referencja odwołuje się.
- odwoływanie się poprzez referencję może pozwalać kompilatorowi na lepszą optymalizację kodu (brak zmian przechowywanego adresu).

Referencje versus zmienne wskaźnikowe

Zalety stosowania referencji:

- referencja jest bezpieczniejszym sposobem odwołania się do zmiennej. Nie daje jednak stuprocentowej pewności, że zmienna/obiekt, do której odwołujemy się przez referencję, istnieje.
- sposób odwołania się do zmiennej/obiektu jest dla programisty *przezroczysty*. Referencję traktujemy tak samo jak zmienną, do której ta referencja odwołuje się.
- odwoływanie się poprzez referencję może pozwalać kompilatorowi na lepszą optymalizację kodu (brak zmian przechowywanego adresu).

Referencje versus zmienne wskaźnikowe

Zalety stosowania referencji:

- referencja jest bezpieczniejszym sposobem odwołania się do zmiennej. Nie daje jednak stuprocentowej pewności, że zmienna/obiekt, do której odwołujemy się przez referencję, istnieje.
- sposób odwołania się do zmiennej/obiektu jest dla programisty *przezroczysty*. Referencję traktujemy tak samo jak zmienną, do której ta referencja odwołuje się.
- odwoływanie się poprzez referencję może pozwalać kompilatorowi na lepszą optymalizację kodu (brak zmian przechowywanego adresu).

Referencje versus zmienne wskaźnikowe

Wady stosowania referencji:

- brak elastyczności jaką dają wskaźniki (możliwość zmiany adresu).

Referencje versus zmienne wskaźnikowe

Wady stosowania referencji:

- brak elastyczności jaką dają wskaźniki (możliwość zmiany adresu).

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
```

```
int  Zm;
int * const  wskZm = &Zm;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;

refZm = 2;
```

```
int  Zm;
int * const  wskZm = &Zm;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;

refZm = 2;
cout <<  Zm <<  endl;
```

```
int  Zm;
int * const  wskZm = &Zm;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;

refZm = 2;
cout <<  Zm <<  endl;
```

```
int  Zm;
int * const  wskZm = &Zm;

*wskZm = 2;
cout <<  Zm <<  endl;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

```
int  Zm;
int * const  wskZm = &Zm;
   ...
   ...
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Jaka wartość zostanie wyświetlona w tym przypadku?

```
int  Zm;
int * const  wskZm = &Zm;
   ...
   ...
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Jaka wartość zostanie wyświetlona w tym przypadku?

```
int  Zm;
int * const  wskZm = &Zm;
   ...
   ...
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

A jaka wyświetli się tutaj?

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Jaka wartość zostanie wyświetlona w tym przypadku?

```
int  Zm;
int * const  wskZm = &Zm;
   ...
   ...
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Tego nikt nie wie!!! Gdyż ...

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Jaka wartość zostanie wyświetlona w tym przypadku?

```
int  Zm;
int * const  wskZm = &Zm;
   ...
(int *)wskZm = new   int ;
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Tego nikt nie wie!!! Gdyż można wymusić zmianę wskaźnika.

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Jaka wartość zostanie wyświetlona w tym przypadku?

```
int  Zm;
int * const  wskZm = &Zm;
   ...
const_cast <int *>(wskZm) = new   int ;
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Tego nikt nie wie!!! Gdyż można wymusić zmianę wskaźnika.

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Tu na pewno wyświetli się 2, gdyż ...

```
int  Zm;
int * const  wskZm = &Zm;
   ...
const_cast <int *>(wskZm) = new   int ;
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Tego nikt nie wie!!! Gdyż można wymusić zmianę wskaźnika.

Referencje versus zmienne wskaźnikowe

```
int  Zm;
int  & refZm = Zm;
   ...
   ...
   ...
refZm = 2;
cout <<  Zm <<  endl;
```

Tu na pewno wyświetli się 2, gdyż referencja nie może być zmieniona.

```
int  Zm;
int * const  wskZm = &Zm;
   ...
const_cast <int *>(wskZm) = new   int ;
   ...
*wskZm = 2;
cout <<  Zm <<  endl;
```

Tego nikt nie wie!!! Gdyż można wymusić zmianę wskaźnika.

Spis treści

Przekazywanie parametrów przez referencję

```
void Poteguj(int & Wart)
{
   Wart *= Wart;
}

int main( )
{
   int Zm=2;

   Poteguj(Zm);
   cout << Zm << endl;
}
```

Używając referencji możemy poprzez parametr przekazać do funkcji zmienną, której zmiany wartości będą widoczne na „zewnątrz”.

Przekazywanie parametrów przez referencję

```
void Poteguj(int & Wart)
{
   Wart *= Wart;
}

int main( )
{
   int Zm=2;

   Poteguj(Zm);
   cout << Zm << endl;
}
```

```
void Poteguj(int * const wWart)
{
   *wWart *= *wWart;
}

int main( )
{
   int Zm=2;

   Poteguj(&Zm);
   cout << Zm << endl;
}
```

Analog tej konstrukcji możemy napisać wykorzystując przekazywanie parametru poprzez wskaźnik.

Definiowanie operacji dodania i podstawienia?

```
struct LZespolona { double re, im; };
 ...
LZespolona DodajIPodstaw( LZespolona *pArg1, LZespolona Arg2 );
{
    return *pArg1 = *pArg1 + Arg2;
}
 ...
int main( )
{
    LZespolona Wynik, Z1;

    DodajIPodstaw(&Wynik, Z1);
    operator+= (&Wynik, Z1);
    &Wynik += Z1;
}
```

Definiowanie operacji dodania i podstawienia?

```
struct LZespolona { double re, im; };
 ...
LZespolona DodajIPodstaw( LZespolona &Arg1, LZespolona Arg2 );
 {
    return Arg1 = Arg1 + Arg2;
 }
 ...
int main( )
{
    LZespolona Wynik, Z1;

    DodajIPodstaw(Wynik, Z1);
    operator+= (&Wynik, Z1);
    &Wynik += Z1;
 }
```

Definiowanie operacji dodania i podstawienia?

```
struct LZespolona { double re, im; };
...
LZespolona DodajIPodstaw( LZespolona &Arg1, LZespolona Arg2 );
LZespolona operator += ( LZespolona *pArg1, LZespolona Arg2 );
...

int main( )
{
    LZespolona Wynik, Z1;

    DodajIPodstaw(Wynik, Z1);
    operator+= (&Wynik, Z1);
    &Wynik += Z1;
}
```

Definiowanie operacji dodania i podstawienia?

```
struct LZespolona { double re, im; };
 ...
LZespolona DodajIPodstaw( LZespolona &Arg1, LZespolona Arg2 );
LZespolona operator += ( LZespolona &Arg1, LZespolona Arg2 );
 ...

int main( )
{
    LZespolona Wynik, Z1;

    DodajIPodstaw(Wynik, Z1);
    operator+= (Wynik, Z1);
    Wynik += Z1;
}
```

Definiowanie operacji dodania i podstawienia?

```
struct LZespolona {   double  re, im;   };
 ...
LZespolona  DodajIPodstaw( LZespolona  &Arg1, LZespolona  Arg2 );
LZespolona&  operator += ( LZespolona  &Arg1, LZespolona  Arg2 );
 {
     return  Arg1 = Arg1 + Arg2;
 }

int  main( )
{
     LZespolona    Wynik, Z1;

     operator+= (Wynik, Z1);
     Wynik += Z1;
 }
```

Spis treści

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;




}
```

Proste zadanie

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

1. wykryć błędne wprowadzenie liczby,
2. poinformować użytkownika o błędzie,
3. zignorować wprowadzony napis,
4. ponownie podjąć próbę czytania,
5. i tak aż do skutku.

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;
   cin >> Arg;



}
```

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

1. wykryć błędne wprowadzenie liczby,
2. poinformować użytkownika o błędzie,
3. zignorować wprowadzony napis,
4. ponownie podjąć próbę czytania,
5. i tak aż do skutku.

Rozwiązanie

Najpierw musimy podjąć próbę wczytania liczby.

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;
   cin >> Arg;
   while ( cin.fail( ) ) {


   }
}
```

Program ma wczytać liczbę. ***Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis.*** Program musi:

1. ***wykryć błędne wprowadzenie liczby,***
2. poinformować użytkownika o błędzie,
3. zignorować wprowadzony napis,
4. ponownie podjąć próbę czytania,
5. ***i tak aż do skutku.***

Rozwiązanie

Następnie należy sprawdzić czy operacja się powiodła. Jeżeli nie (metoda `fail` zwróci wartość `true`), to należy rozpocząć pętlę, w której próby wczytania będą podejmowane, aż do pozytywnego skutku.

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;
   cin >> Arg;
   while ( cin.fail( ) ) {
     cout << "Blad. Wpisz ponownie." << endl;


   }
}
```

Program ma wczytać liczbę.Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

1. wykryć błędne wprowadzenie liczby,
2. ***poinformować użytkownika o błędzie,***
3. zignorować wprowadzony napis,
4. ponownie podjąć próbę czytania,
5. i tak aż do skutku.

Rozwiązanie

Najpierw musimy użytkownika poinformować o błędzie.

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;
   cin >> Arg;
   while ( cin.fail( ) ) {
     cout << "Blad. Wpisz ponownie." << endl;
     cin.clear( );


   }
}
```

Program ma wczytać liczbę.Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

1. wykryć błędne wprowadzenie liczby,
2. poinformować użytkownika o błędzie,
3. zignorować wprowadzony napis,
4. ponownie podjąć próbę czytania,
5. i tak aż do skutku.

Rozwiązanie

Błąd wykonania operacji w prowadza strumień w stan *fail*. To powoduje, że od tego momentu nie można wykonać jakiejkolwiek operacji wczytania (będą one ignorowane).
Aby wyprowadzić strumień z tego stanu i przywrócić mu możliwość wykonywania dalszych operacji, należy wywołać metodę `clear`.

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;
   cin >> Arg;
   while ( cin.fail( ) ) {
      cout << "Blad. Wpisz ponownie." << endl;
      cin.clear( );
      cin.ignore(10000,'\n');

   }
}
```

Program ma wczytać liczbę.Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

1. wykryć błędne wprowadzenie liczby,
2. poinformować użytkownika o błędzie,
3. ***zignorować wprowadzony napis,***
4. ponownie podjąć próbę czytania,
5. i tak aż do skutku.

Rozwiązanie

Teraz możemy zignorować cały napis, którą błędnie wprowadził użytkownik. Będąc bardziej precyzyjnym, metoda `ignore` ignoruje 10000 znaków, chyba, że wcześniej napotka znak przejścia do nowej linii (użytkownik wprowadza go naciskając klawisz Enter). Wszystko będzie dobrze działać pod warunkiem, że użytkownik w jednej linii nie wprowadzi więcej niż 10000 znaków. Można śmiało założyć, że prędzej się zmęczy i da spokój ;)

Jak radzić sobie z błędami czytania

```
int main( )
{
   int Arg;
   cin >> Arg;
   while ( cin.fail( ) ) {
      cout << "Blad. Wpisz ponownie." << endl;
      cin.clear( );
      cin.ignore(10000,'\n');
      cin >> Arg;
   }
}
```

Program ma wczytać liczbę.Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

1. wykryć błędne wprowadzenie liczby,
2. poinformować użytkownika o błędzie,
3. zignorować wprowadzony napis,
4. ***ponownie podjąć próbę czytania,***
5. i tak aż do skutku.

Rozwiązanie

Wykonujemy ponowną próbę wczytania liczby.

Koniec prezentacji
Dziękuję za uwagę