

# Dzisiejszy wykład

## # Klasa string

- wersja prosta
- wersja ze zliczaniem odwołań

## # Wyjątki

## # Specyfikator *volatile*

## # Semaforey

# Klasa *string*

- Przetwarzanie tekstów jest powszechną dziedziną zastosowań komputerów
- W języku C i C++ brak prawdziwego wbudowanego typu łańcuchowego, `char*` jest trudny w użyciu i podatny na błędy

```
class mystring
{
    char *dane;
public:
    mystring ();
    ~mystring ();
    mystring (const char *s);
    unsigned int length () const;
    mystring (const mystring & src);
    mystring &operator= (const mystring & src);
    mystring &operator+= (const mystring & src);
    char operator[] (unsigned int i) const;
    char &operator[] (unsigned int i);
    friend ostream & operator<< (ostream & o, const mystring & s);
};
inline mystring operator+ (const mystring & s1, const mystring & s2);
```

Referencja

Brak referencji

# Klasa *string* - konstruktor i lista inicjacyjna

## # Bez użycia listy inicjacyjnej

```
mystring ()  
{  
    dane = NULL;  
};
```

## # Z użyciem listy inicjacyjnej

```
mystring ():dane (NULL) {};
```

## # W tym przypadku nie ma żadnej różnicy.

## # Różnica pojawia się, jeżeli pola posiadają konstruktory

- Bez listy inicjacyjnej: wywołany konstruktor bezparametrowy, a następnie operator przypisania
- Z listą inicjacyjną: wywołany konstruktor z parametrem

## # Zastosowanie inicjalizacji pól obiektowych klasy skraca program i przyspiesza jego wykonanie

# Klasa *string* - operator +

- # Specyfikator friend nie jest potrzebny, funkcja korzysta wyłącznie ze składowych publicznych klasy *string*
- # Podczas zwracania wartości z funkcji zostanie użyty konstruktor kopiujący

```
inline mystring operator+ (const mystring & s1, const mystring & s2)
{
    mystring s (s1);
    return s += s2;
}
```

# Klasa *string* - operator *char\**

- # W klasie można zdefiniować operatory konwersji do innego typu
- # Często użyteczne w przypadku wykorzystywania funkcji przyjmujących typy wbudowane w język (np. *char\**)

```
mystring::operator char*()  
{  
    if(!dane)  
    {  
        dane=new char[1];  
        dane[0]='\0';  
    }  
    return dane;  
};
```

- # W klasie *string* możliwość kolizji z operatorem indeksowania

```
mystring s;  
s[0]='a'; //s.operator[](0) or (s.operator char*())[0] ?
```

# Wyjątki

## ⚡ Poprawny program sprawdza kody błędów

```
int fun()
{
    //...
    fd=open("file",O_RDWR);
    if(fd==-1)
        return -1;
    if(fread(fb,buffer,15)!=15)
        return -1;
    if(fwrite(fd,buffer,4)!=4)
        return -1;
    //...
    return 0;
}
```

- ⚡ Sprawdzanie kodów błędów za każdym razem jest nużące i łatwo o nim zapomnieć
- ⚡ W niektórych przypadkach trudno jest zwrócić kod błędu, np. *atoi()* może zwrócić dowolną liczbę całkowitą i nie ma wartości, która może sygnalizować błąd
- ⚡ Zwracanie błędów z konstruktorów obiektów jest utrudnione

# Wyjątki

## # Język C++ dostarcza nowy mechanizm zgłaszania błędów: wyjątki

```
class myexception{};
void f1()
{
    if(...)
        throw myexception();
};
void f2()
{
    f1();
};
int main()
{
    try {
        f2();
    } catch(myexception&)
    {
        cout << "myxception caught"<<endl;
    }
    return 0;
};
```

Klasa wyjątku, umożliwiająca rozróżnienie błędów

Zgłoszenie błędu

Wykrycie błędu

# Wyjątki

# Rozróżnienie błędów następuje na podstawie typu zgłaszanego wyjątku

```
class bad_index{};
class no_memory {};
void f1()
{
    if(...)
        throw bad_index();
    if(...)
        throw no_memory();
};
//...
try {
    f2();
} catch(bad_index&)
{
    //...
} catch (no_memory&)
{
    //...
}
```

# Operator new zgłasza wyjątek *bad\_alloc* w przypadku braku pamięci



# Wyjątki

- Podczas zgłaszania wyjątku wywoływane są destruktory wszystkich lokalnych obiektów na drodze od funkcji wywoływanej do wywołującej

```
#include <iostream>
using namespace std;
class myexception{};
class tester
{
    string name;
public:
    tester(const string& n): name(n)
    {
        cout<<name<<" () "<<endl;
    };
    ~tester()
    {
        cout<<"~"<<name<<" () "<<endl;
    };
};
```

```
void f1()
{
    tester f1("f1");
    throw myexception();
    cout << "Exiting f1"<<endl;
};

void f2()
{
    tester f2("f2");
    f1();
    cout << "Exiting f2"<<endl;
};

int main()
{
    tester main("main");
    f2();
    return 0;
};
```

# Wyjątki

- # Jeżeli nie ma odpowiedniej instrukcji *catch*, wykonywanie programu jest przerywane
- # Możemy zmodyfikować klasę *string* tak, aby zgłaszała wyjątki

```
class mystring
{
    char *dane;
public:
    class index_out_of_range{};
    //...
    char operator[] (unsigned int i) const
    {
        if (!dane)
            throw index_out_of_range();
        if (i >= strlen (dane))
            throw index_out_of_range();
        return dane[i];
    }
};
```

# Problemy z wyjątkami

- ✘ Nieuważne stosowanie wyjątków może prowadzić do wycieku zasobów lub niespójności danych

```
void fun()
{
    char* a = new char[1024];
    char* b = new char[1024];
    //...
    delete [] a;
    delete [] b;
};
```

Jeżeli *new* zgłosi wyjątek,  
obszar zaalokowany dla  
zmiennych *a* nie zostanie  
zwolniony

```
void fun()
{
    char* a = NULL, *b=NULL;
    try{
        a = new char[1024];
        b = new char[1024];
        //...
    } catch (...)
    {
        delete [] a;
        delete [] b;
        throw;
    }
    delete [] a;
    delete [] b;
};
```

Metoda naprawienia  
błędu, poprawna ale nie  
najlepsza

# Problemy z wyjątkami

⚠ Nieuważne stosowanie wyjątków może prowadzić do wycieku zasobów lub niespójności danych

```
mystring & operator=
    (const mystring & src)
{
    if (this != &src)
    {
        delete [] dane;
        if (src.dane)
        {
            dane = new char
                [strlen(src.dane) + 1];
            strcpy (dane, src.dane);
        }
        else
            dane = 0;
    };
    return *this;
}
```

Jeżeli *new* zgłosi wyjątek, zawartość łańcucha zostanie utracona, a *dane* będzie miała nieprawidłową wartość, co spowoduje problemy w destruktorze

```
mystring & operator=
    (const mystring & src)
{
    if (this != &src)
    {
        char* nowedane;
        if (src.dane)
        {
            nowedane = new char
                [strlen (src.dane) + 1];
            strcpy (nowedane, src.dane);
        }
        else
            nowedane = 0;
        delete [] dane;
        dane=nowedane;
    };
    return *this;
}
```

Poprawiona wersja nie niszczy łańcucha w przypadku braku pamięci

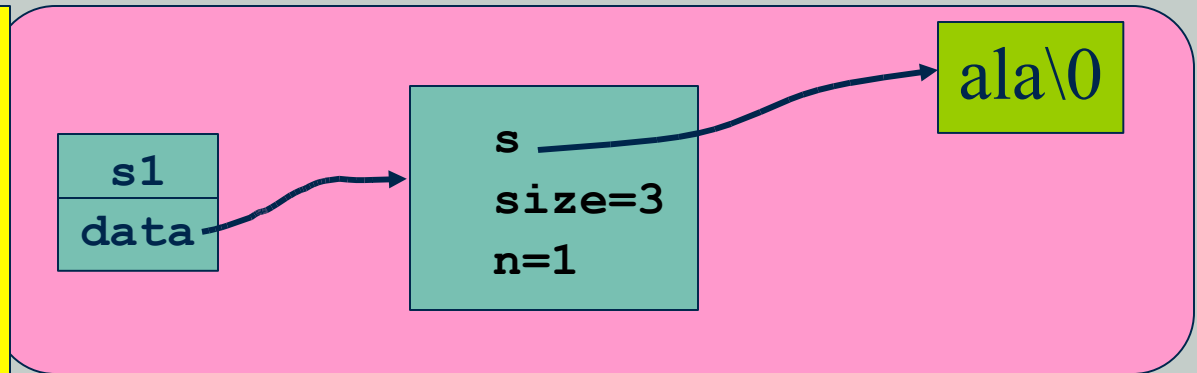
## Klasa *string* - implementacja ze zliczaniem odwołań

- # Kopiowanie łańcucha tekstowego jest operacją dość kosztowną
- # Łańcuchy tekstowe często przesyłane są przez wartość
- # Szukamy sposobu implementacji, która zmniejszy liczbę alokacji pamięci i kopiowań
- # Rozwiązaniem jest zastosowanie zliczania odwołań

# Klasa *string* - implementacja ze zliczaniem odwołań

- # Klasa jest jedynie uchwytym - zawiera wskaźnik do rzeczywistej implementacji łańcucha

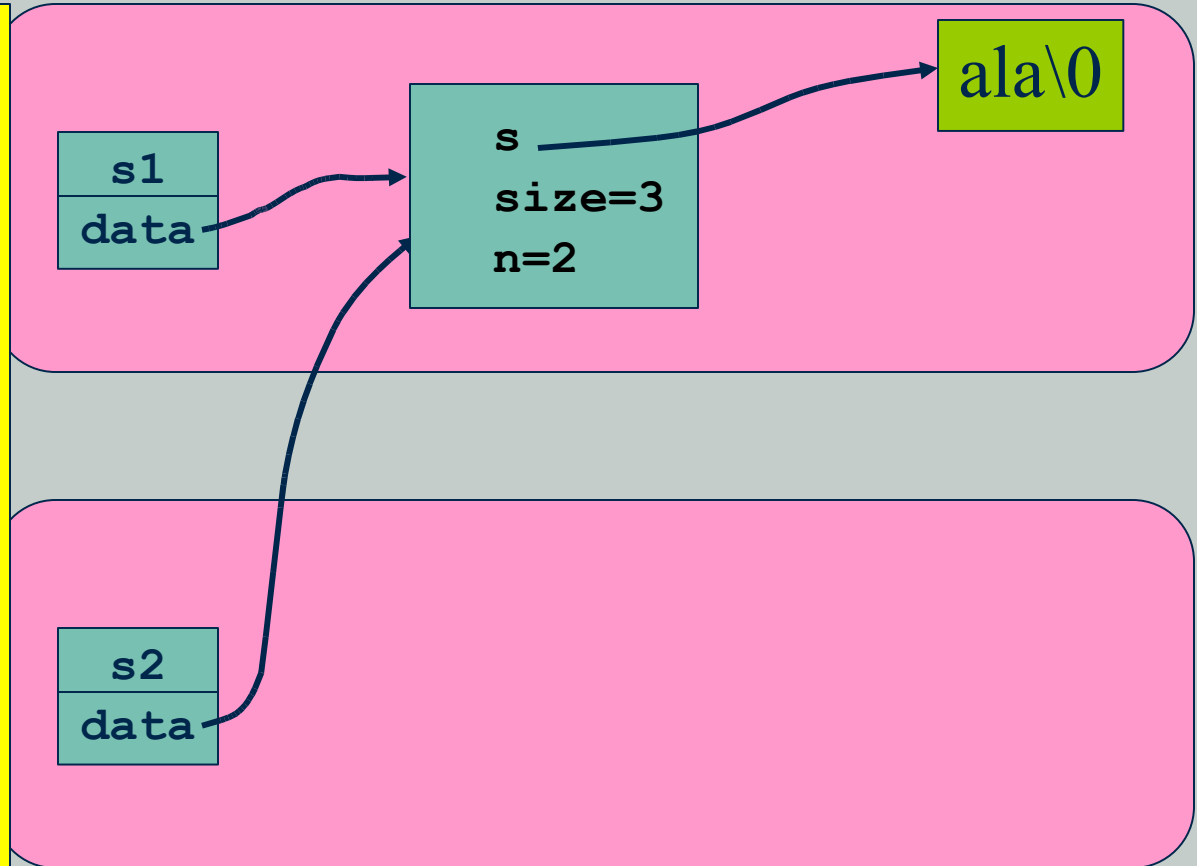
```
class string{  
    struct rctext;  
    rctext* data;  
public:  
    string(char* s);  
    string& operator=  
        (const string&);  
    ~string();  
    ...  
};  
struct string::rctext  
{  
    char* s;  
    unsigned int size;  
    unsigned int n;  
    ...  
};  
string s1("ala");
```



# Klasa *string* - implementacja ze zliczaniem odwołań

- # Kopiowanie łańcucha kopiuje wskaźnik do wewnętrznej reprezentacji i zwiększa licznik odwołań

```
class string{  
    struct rctext;  
    rctext* data;  
public:  
    string(char* s);  
    string& operator=  
        (const string&);  
    ~string();  
    ...  
};  
struct string::rctext  
{  
    char* s;  
    unsigned int size;  
    unsigned int n;  
    ...  
};  
string s1("ala");  
string s2=s1;
```

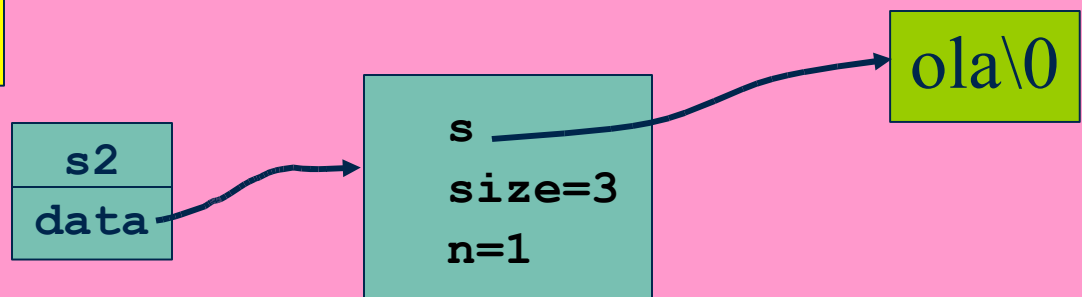
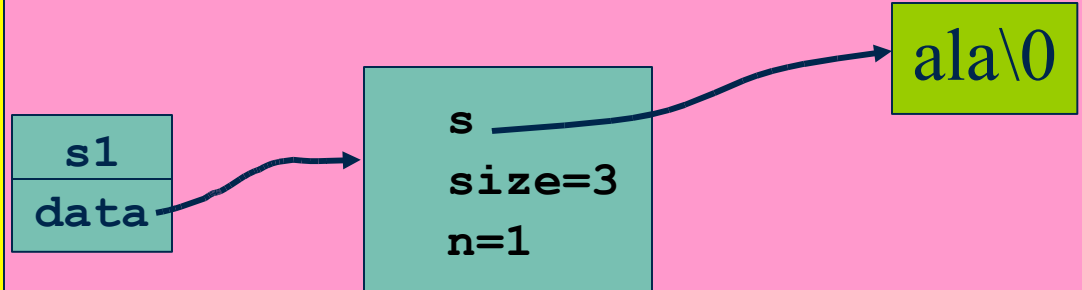


# Klasa *string* - implementacja ze zliczaniem odwołań

# Modyfikacja jednej z kopii łańcucha powoduje rozdzielenie s1 i s2

```
class string{  
public:  
    char read  
        (unsigned int i) const;  
    void write  
        (unsigned int i, char c);  
    ...  
};
```

```
string s1("ala");  
string s2=s1;  
s2.write(0, 'o');
```





## Klasa *string* - implementacja ze zliczaniem odwołań

- # Chcemy, żeby operator indeksowania działał poprawnie dla zapisu i odczytu

```
class string{  
    ...  
};  
  
string s1("ala");  
string s2=s1;  
char a=s2[0];  
s2[0]='o';
```

- # Nie chcemy, aby odczyt znaku powodował wykonanie kopii łańcucha
- # Możemy to osiągnąć poprzez zwracanie przez operator indeksowania nie referencji do znaku, ale specjalnej klasy *Cref*

# Klasa *string* - implementacja ze zliczaniem odwołań

# Klasa Cref zachowuje się jak char, ale umożliwia rozróżnienie pisania i czytania

```
class string{
public:
    string(char* s);
    string& operator=(const string&);
    ~string();
    void check(unsigned int i) const;
    Cref operator[](unsigned int i);
    char operator[](unsigned int i) const;
    char string::read(unsigned int i) const;
    void string::write(unsigned int i, char c){
        data = data->detach();
        data->s[i] = c;
    }
    ...
};

string::Cref
    string::operator[](unsigned int i)
{
    check(i);
    return Cref(*this,i);
};
```

```
class string::Cref
{
    friend class string;
    string& s;
    unsigned int i;
    Cref (string& ss, unsigned int ii)
        :s(ss), i(ii) {};
public:
    operator char() const
    {
        return s.read(i);
    }
    string::Cref& operator = (char c)
    {
        s.write(i,c);
        return *this;
    };
    string::Cref& operator = (const Cref& ref)
    {
        return operator= ((char)ref);
    };
};
```

# rcstring i wielowątkowość

# Implementacja ze zliczaniem odwołań może powodować problemy w programach wielowątkowych

```
#include <stdio.h>
#include <pthread.h>
#include "rcstring.h"
rcstring s ("ala");
void * thread1 (void *p)
{
    for (unsigned int i = 0;
         i < 100000; i++)
    {
        rcstring s1 (s);
    }
    pthread_exit (p);
}
void * thread2 (void *p)
{
    for (unsigned int i = 0;
         i < 100000; i++)
    {
        rcstring s1 (s);
    }
    pthread_exit (p);
}
```

```
int main ()
{
    pthread_t t1;
    pthread_t t2;
    pthread_create (&t1, NULL, thread1, NULL);
    pthread_create (&t2, NULL, thread2, NULL);
    pthread_join (t1, (void **) &retval1);
    pthread_join (t2, (void **) &retval2);
    printf ("RefCount=%d\n", s.getRefCount ());
};
```

```
inline rcstring::rcstring(const rcstring& x)
{
    x.data->n++;
    data=x.data;
}
inline rcstring::~rcstring() {
    if(--data->n==0)
        delete data;
}
```

Dwa wątki mogą jednocześnie zmieniać wartość n

# rcstring i wielowątkowość

- ⌘ Operacja "n++" nie jest operacją niepodzielną
- ⌘ Składa się z trzech faz:
  - Pobranie wartości z pamięci
  - Zwiększenie wartości
  - Wpisanie wartości do pamięci
- ⌘ Te operacje mogą się przeplatać w kilku wątkach
- ⌘ W efekcie zmienna ma niewłaściwą wartość końcową (zwiększenie o jeden, zamiast o dwa)

```
thread1: d0=n;  
thread2: d0=n;  
thread1: d0=d0+1;  
thread2: d0=d0+1;  
thread1: n=d0;  
thread2: n=d0;
```

- ⌘ Konieczność stosowania obiektów synchronizacyjnych (semaforów) lub specjalnych instrukcji asemblera

# Specyfikator *volatile*

- # Specyfikator *volatile* oznacza, że wartość zmiennej może zmieniać się poza kontrolą programu i w związku z tym zabrania optymalizacji odnoszących się do tej zmiennej (np. przechowywania w rejestrach)
- # Używana, kiedy odwołania do zmiennej są odwołaniami do rejestrów urządzeń wejścia/wyjścia
- # Niestety, nie pomoże nam w przypadku łańcucha ze zliczaniem odwołań

# Specyfikator *volatile*

- # Dodanie specyfikatora **volatile** spowoduje, że program zakończy się zgodnie z naszymi oczekiwaniami
- # To jedynie ilustracja - użycie **volatile** do synchronizacji pamięci między wątkami nie jest poprawne

```
#include <stdio.h>
#include <pthread.h>
int val = 0;
void * fun1 (void *)
{
    val = 0;
    while (1)
        if (val != 0)
            break;
    printf ("wyszliśmy z petli\n");
    pthread_exit (NULL);
}
int main ()
{
    pthread_t w;
    pthread_create
        (&w, NULL, fun1, NULL);
    val = 1;
    pthread_join (w, NULL);
};
```

```
#include <stdio.h>
#include <pthread.h>
volatile int val = 0;
void * fun1 (void *)
{
    val = 0;
    while (1)
        if (val != 0)
            break;
    printf ("wyszliśmy z petli\n");
    pthread_exit (NULL);
}
int main ()
{
    pthread_t w;
    pthread_create
        (&w, NULL, fun1, NULL);
    val = 1;
    pthread_join (w, NULL);
};
```

# Semafor

```
struct rcstring::rctext
{
    char *s;
    unsigned int size;
    unsigned int n;
    pthread_mutex_t mutex;
    rctext (unsigned int nsize,
            const char *p)
    {
        n = 1;
        size = nsize;
        s = new char[size + 1];
        strncpy (s, p, size);
        s[size] = '\\0';
        pthread_mutex_init
            (&mutex, NULL);
    };
    unsigned int AtomicIncrement()
    {
        pthread_mutex_lock(&mutex);
        unsigned int retval=n++;
        pthread_mutex_unlock(&mutex);
        return retval;
    };
    unsigned int AtomicDecrement()
    {
        pthread_mutex_lock(&mutex);
        unsigned int retval=n--;
        pthread_mutex_unlock(&mutex);
        return retval;
    };
    //...
};
```

