

Podstawowe typy danych₃

- **char** — Jednocześnie działa jako typ znakowy oraz liczbowy typ dla małych liczb całkowitych ■
- **int** — podstawowy typ dla liczb całkowitych ■
- **float** — podstawowy typ liczb zmiennoprzecinkowych ■
- **double** — typ liczb zmiennoprzecinkowe z podwójną precyzją ■
- **bool** — typ zmiennych logicznych (Boolowskich) ■

Zmienna typu 'bool' przyjmuje jako wartości tylko 'true' albo 'false'!

Typ zmiennych logicznych⁵

- wyrażenie w nawiasach po **if** lub **while**: **wyrażenie boolowskie** ■
 - wyrażenia boolowskie budowane są przy pomocy słowa kluczowego **bool** i operatorów logicznych:
 - **and**, również pisane **&&** ■
 - **or**, również pisane **||** ■
 - **not**, również pisane **!** ■
-
- operatory porównania **<**, **>**, **<=**, **>=**, **==**, **!=**: zwracają wartość typu **bool** ■

Typ zmiennych logicznych₆

```
int main(){
    bool b=false;
    cout << "b=" << b << endl;
    cout << "!b=" << !b << endl;

    int j=2,k=-1;
    b= (j>0) && (k!=0);
    cout << "b=" << b << "\n";
    if(b) cout << "b is true\n";
    else cout << "b is false\n";

    b= (j<0) || (k==0);
    cout << "b=" << b << endl;
    if(b) cout << "b is true\n";
    else cout << "b is false\n";
}
```

Liczby pierwsze₇

Exercise 3.1. Napisz funkcję z jednym argumentem całkowitym bez znaku, która zwraca **true**, jeśli argument jest liczbą pierwszą, a w przeciwnym razie **false**.

Ex. 3.1 zapisany jako przykład53.cpp

Typ char₈

- **char**: służy do przechowywania 8-bitowych liczb całkowitych
- interpretowane jako kody znaków w standardzie ASCII ■
- literały znakowe pisane zamknięty apostrofami

```
char x='A' ;
```

■

- na wydruku pokazuje znak, a nie jego kod ■

Specyfikatory⁹

Niektóre typy danych C / C++ mogą być modyfikowane za pomocą tzw. specyfikatorów.

Specyfikatory: **short**, **long**, **signed** oraz **unsigned**. ■

Specyfikatory **short** oraz **long**

- sugerują pojemność (maksymalny zakres liczb) odpowiednio niższą lub wyższą niż typ podstawowy. ■
- Rzeczywista pojemność zależy od kompilatora ■
- C/C++ wymaga jedynie by
 - typ **char** miał minimalną pojemność 8 bitów, ■
 - typ **short int** miał minimalną pojemność 16 bitów, ■
 - typ **long int** miał minimalną pojemność 32 bitów, ■

Specyfikatory **signed** oraz **unsigned** 10

- określają, czy jeden bit ma być używany jako znak, czy nie. Wpływa to na pojemność typu. ■
- mogą być używane z typami **char** i **int**. ■

Rzeczywistą pojemność (w bajtach) dla każdego typu można sprawdzić, wywołując operator **sizeof**, przy czym argumentem jest nazwa typu lub zmienna danego typu.

Typ wyliczeniowy ¹¹

- Pisząc

```
enum weekday {Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

definiujemy własny typ, tak zwany **typ wyliczeniowy**. ■

- Możemy teraz deklarować zmienne typu **weekday**:

```
weekday payDay=pia;
```

■

- Kompilator kojarzy liczby całkowite z elementami typu wyliczeniowego, zaczynając od zera. Można to zmodyfikować poprzez jawne skojarzenie określonego typu elementu z inną liczbą:

```
enum controlKeys {tab=8, enter=13 }
```

■

Typ wyliczeniowy ¹²

- Nie możemy bezpośrednio przypisać wartości całkowitej do zmiennej typu wyliczeniowego. Konieczna jest jawna konwersja. ■
- Zatem instrukcja

```
payDay=4; // brak konwersji  
spowoduje błąd. ■
```

- Możemy napisać

```
payDay=weekday(7);
```

choć nie przypisuje to żadnego dnia z listy, ponieważ sobota to dzień tygodnia (6), a niedziela to dzień tygodnia (0).

```
enum weekDay{Sun, Mon, Tue, Wed, Thu, Fri, Sat};  
int main(int argc, char* argv[]){  
    using namespace std;  
  
    weekDay payDay=Fri;  
  
    //payDay=4; // lack of conversion  
    //payDay=weekDay(-3); // non-existing value  
    //payDay=weekDay(8); // non-existing value  
    std::cout << payDay << endl;  
}
```

Typ `void`₁₄

- Typ `void` jest używany w C/C++ do zaznaczenia typu pustego, to jest typu, który nie przyjmuje żadnych wartości. ■
- Dla wtajemniczonych: Jest on używany ze wskaźnikami do konstruowania wskaźników do obiektów nieznanego typu:

```
void * p;
```



- Jest używany również z funkcjami, aby wskazać, że funkcja nie zwraca żadnej wartości:

```
void f ();
```

Słowo kluczowe **typedef** ¹⁵

- Słowo kluczowe **typedef** nie definiuje nowego typu.
- Służy do nadania nowej nazwy istniejącemu typowi. ■
- Nowa nazwa może być krótsza i bardziej opisowa ■

```
typedef unsigned long int Ulong;  
typedef int* IntPtr; //Dla wtajemniczonych  
Ulong, j;  
IntPtr p, q;
```

Kontrola typów w C/C++ ¹⁶

Kontrola typów to działanie kompilatora, którego celem jest zminimalizowanie błędów związanych z niezgodnością typów przy podstawieniu lub wywołaniu funkcji. Dzieli się na **statyczną** t.j. wykonywaną w czasie kompilacji oraz **dynamiczną**, wykonywaną przez kod wygenerowany przez kompilator.■

- C ma stosunkowo słabą kontrolę typów. ■
- C++ charakteryzuje się stosunkowo silną, ale tylko statyczną kontrolą typów ■
- Jednak w celu zagwarantowania zgodności z C sprawdzanie typu w C++ nie jest pełne. ■
- Brak dynamicznego sprawdzania typu wynika z orientacji C++ na wydajność kodu. ■

- **literał**: wartość zmiennej wprowadzonej bezpośrednio do programu ■
- Literały są używane do inicjalizacji zmiennych oraz jako argument funkcji i operatorów ■

Literały całkowite ¹⁸

- reprezentują liczby całkowite ■
 - – Dziesiętne: 2015, −1 ■
 - ósemkowe, rozpoczynają się zerem: 00, 02, 0123 ■
 - szesnastkowe, rozpoczynają się 0x, używają dodatkowych cyfr a,b,c,d,e,f: 0x1a7d ■
-
- przyrostek U: typ **unsigned int**, na przykład, 7U ■
 - przyrostek L: type **long int** na przykład 1L ■

Literały zmiennoprzecinkowe ¹⁹

- reprezentują liczby zmiennoprzecinkowe
 - pisane z kropką dziesiętną: 3.14, 1.0 ■
 - lub z kropką dziesiętną i wykładnikiem: 1.23e - 45, -1.17e10 ■
- domyślnie są typu **double** ■
- pisane z przyrostkiem f są typu **float** eg. 3.14f ■

Są tylko dwa: **false** i **true**

- pisane jako znak w apostrofach ■
- Reprezentują wartość liczbową kodu znakowego w standardzie ASCII ■
- Istnieją również znaki specjalne poprzedzone odwrotnym ukośnikiem: `'\n'` (znak zmiany linii), `'\t'` (tabulator) ■
- Mogą być zapisane kodem ósemkowym poprzedzonym odwrotnym ukośnikiem i zerem, n.p. `'\012'` ■
- Mogą być zapisane kodem szesnastkowym poprzedzonym odwrotnym ukośnikiem i znakiem x, n.p. `'\xff'` ■

- **Literały tekstowe**: ciąg znaków ujęty w cudzysłów

```
"This is a text literal \n"
```



- używane do wypisywania tekstu na standardowym wyjściu
- mogą być używane do inicjalizacji zmiennych typu string

```
cout << "Tests of character and string literals:\n";
char c='?', d='\t' e='\x21';
cout << "c=" << c << "d=" << d << "e=" << e << "\n";
c='\\';
d='\'';
e='\042';
cout << "c=" << c << "d=" << d << "e=" << e << "\n";
cout << "\nTests of integer literals:\n";
int i=1, j=020, k=0xff;
cout << "i=" << i << "j=" << j << "k=" << k << "\n";
cout << "\nTests of floating literals:\n";
float x=7.14, y=6e3, z=11.3e-4;
cout << "x=" << x << "y=" << y << "z=" << z << "\n";
```

Deklarowanie zmiennych ²⁴

- Deklarujemy zmienne, wpisując typ, a następnie listę nazw zmiennych

```
int counter, number;
```

W C/C++ wszystkie zmienne muszą być zadeklarowane przed ich użyciem.

Zmienne lokalne i globalne ²⁵

- **zmienna lokalna** : deklarowana wewnątrz bloku ■
- **zmienna globalna**: deklarowana na zewnątrz bloku ■

Globalne nie są już często popularne.

Zakres ważności zmiennej ²⁶

- Zakres ważności zmiennej globalnej jest od deklaracji do końca pliku
- Zakres ważności zmiennej lokalnej jest od deklaracji do końca najbardziej wewnętrznego bloku zawierającego deklarację

Czas życia zmiennej²⁷

- Czas życia zmiennej globalnej jest od samego początku wykonywania do samego końca wykonywania programu
- Czas życia zmiennej lokalnej jest od momentu realizacji deklaracji do momentu, w którym wykonanie programu przechodzi przez koniec najbardziej wewnętrznego bloku zawierającego deklarację.

Zaśnianie zmiennych ²⁸

```
/* Declaration of a global variable */  
char c = 'G' ;  
  
int main(){  
    cout << "Execution begins\n";  
    cout << "Global variable c contains " << c << "\n";  
    {  
        cout << "Entering a block\n";  
        cout << "Global variable c is " << c << "\n";  
        /* Declaration of a local variable */  
        char c = 'L' ;  
        cout << "Local variable c is " << c << "\n" ;  
        cout << "Global variable c is " << ::c << "\n" ;  
        cout << "Leaving the block\n";  
    }  
    cout << "Global variable c is " << c << "\n";  
}
```

Zaślanianie zmiennych lokalnych ²⁹

```
int main(){
    cout << "Execution begins\n";
    char c = 'G' ;
    cout << "Local variable c is " << c << "\n" ;
    {
        cout << "Entering internal block\n";
        cout << "External c is " << c << "\n" ;
        /* Declaration of an internal local variable */
        char c = 'L' ;
        cout << "Internal c is " << c << "\n" ;
        cout << "External c is inaccessible\n" ;
        cout << "Leaving internal block\n";
    }
    cout << "External c is " << c << "\n" ;
}
```

Statyczne zmienne lokalne ³⁰

- **Statyczna zmienna lokalna**: zmienna z zakresem ważności jak zmienna lokalna i czasem życia zmiennej globalnej ■
- ```
static int i;
```

■
- statyczna zmienna lokalna jest inicjowana tylko raz podczas ładowania programu do pamięci!

## Statyczne zmienne lokalne <sup>31</sup>

```
int main(){
 int n=3;

 for(int i=0;i<n;i++){
 int cnt=0;
 static int staticCnt=0;
 cnt++;
 staticCnt++;
 cout << "cnt=" << cnt << "\n";
 cout << "staticCnt=" << staticCnt << "\n";
 }
 // staticCnt is invisible here
 // cout << "staticCnt==" << staticCnt << "\n";
}
```

## Obiekty stałe<sup>32</sup>

Mamy trzy metody używania stałych obiektów:

- poprzez literał

```
field=3.14 * r * r;
```



- przez dyrektywę preprocesora

```
#define PI 3.14
....
field=PI * r * r;
```



- przez modyfikator **const** (najlepsze rozwiązanie)

```
const float pi=3.14;
....
field=pi * r * r;
```



Ta opcja jest polecana

# Typ odnośnikowy i odnośniki 2

- **typ odnośnikowy** pochodny względem typu **T**: typ, którego obiekty służą do pokazywania na obiekty typu **T** ■
- **odnośniki**: obiekty typu odnośnikowego. ■
- inaczej: obiekty, które zawierają jedynie adresy innych obiektów ■
- odnośniki zazwyczaj przechowywane są w zmiennych ■
- obiekty wskazywane przez odnośniki: tak zmienne jak i obiekty anonimowe ■

W przeciwieństwie do nazw, odnośniki w trakcie kompilacji nie znikają. W skompilowanym programie odnośniki pozostają obiektami, które pokazują na inne obiekty.

Temat nie obowiązkowy, lecz polecam do wyuczenia najlepiej poszukać samemu jak to działa

# Interpretacja typów odnośnikowych<sub>3</sub>

- dwie interpretacje: bliska, daleka ■
- bliska: przy użyciu odnośnika interpretujemy go dosłownie jako adres obiektu, na który odnośnik pokazuje ■
- daleka: przy użyciu odnośnika interpretujemy go jako obiekt, na który odnośnik pokazuje ■
- interpretacja bliska zawsze przy inicjalizacji odnośnika ■
- w pozostałych sytuacjach: tak interpretacja bliska jak i daleka, w zależności od konkretnego rodzaju odnośnika i kontekstu ■

# Referencje <sup>4</sup>

- **referencja**: odnośnik, dla których zawsze stosowana jest interpretacja daleka ■
- C++:

```
int n=1;
int& r = n;
r=2;
```

■



# Wskaźniki<sub>5</sub>

- **wskaźnik**: odnośnik, dla których zawsze stosowana jest interpretacja bli-ska ■
- C++:

```
int n;
int* p=&n;
```

■

- odniesienie do obiektu, na który wskaźnik pokazuje wymaga wykonania jawnie operatora \*, t.zw. **operatora wyłuskania** ■
- C++:i

```
*p=5;
```

■

Odnośników używa się

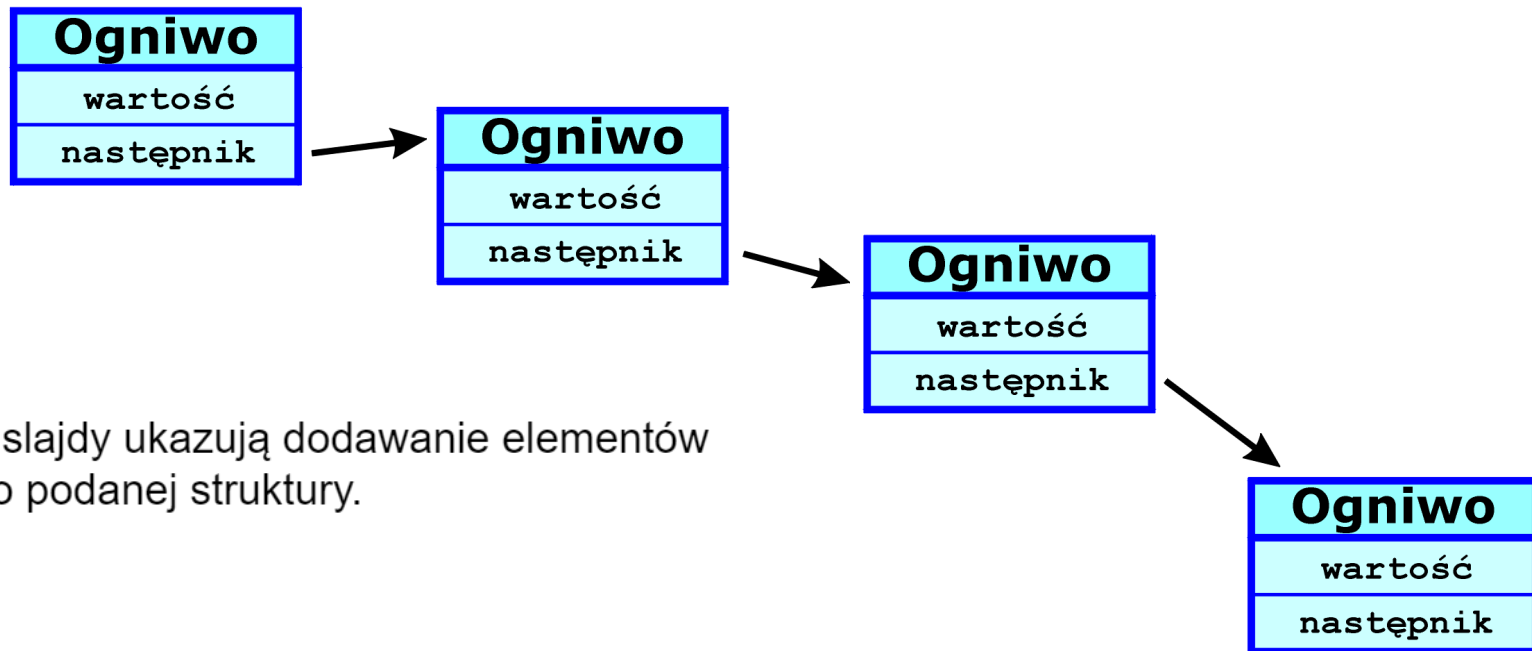
- przy tworzeniu obiektów na stercie. ■
- do tworzenia wiązanych struktur danych takich jak lista czy drzewo ■
- w celu umożliwienia funkcji zmiany wartości przekazanych jej argumentów. ■



Ponadto wskaźniki umożliwiają:

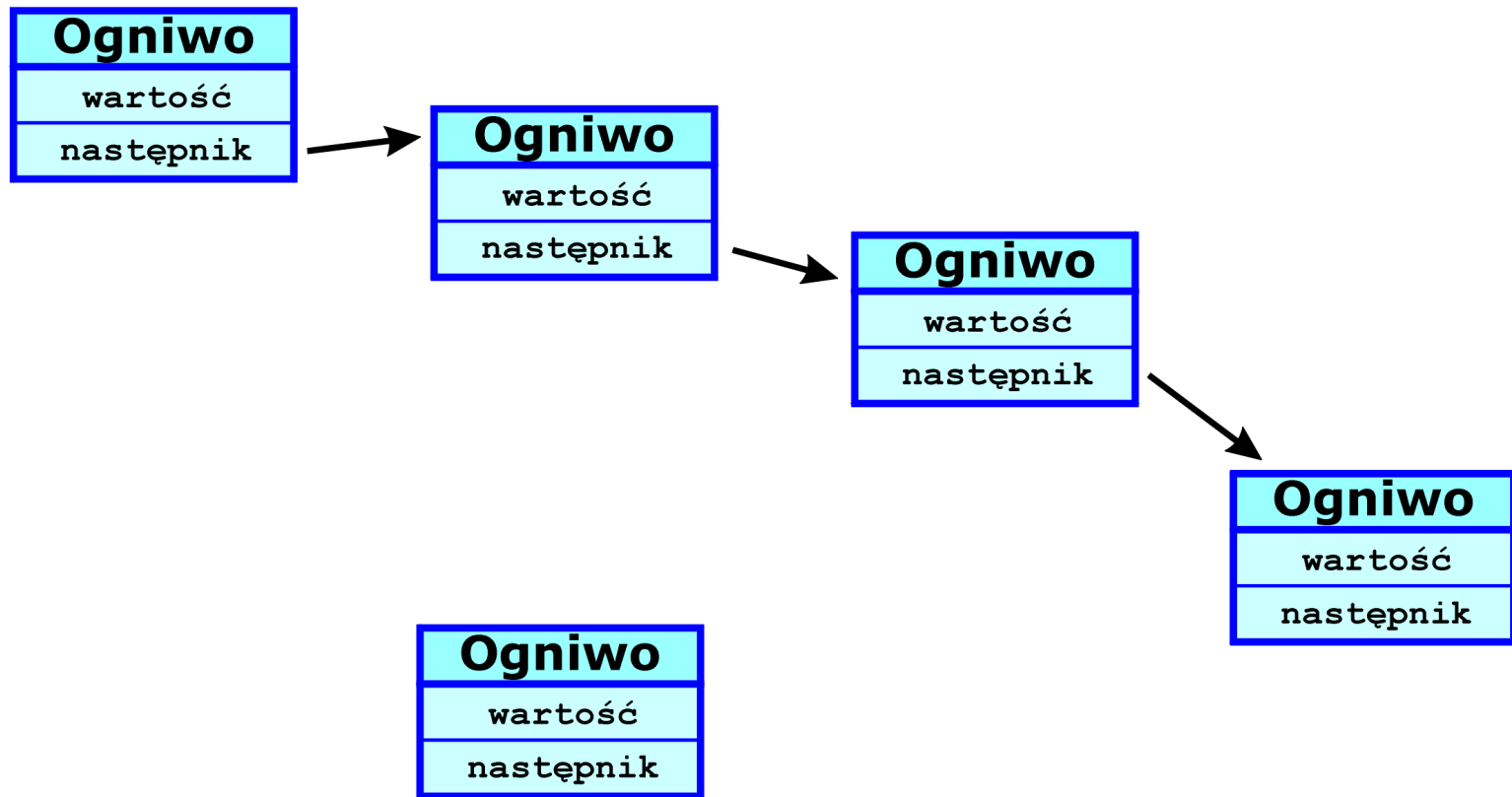
- bezpośredni dostęp do pamięci komputera ■
- szybszy dostęp do elementów tablic ■

# Wiązane struktury danych 7

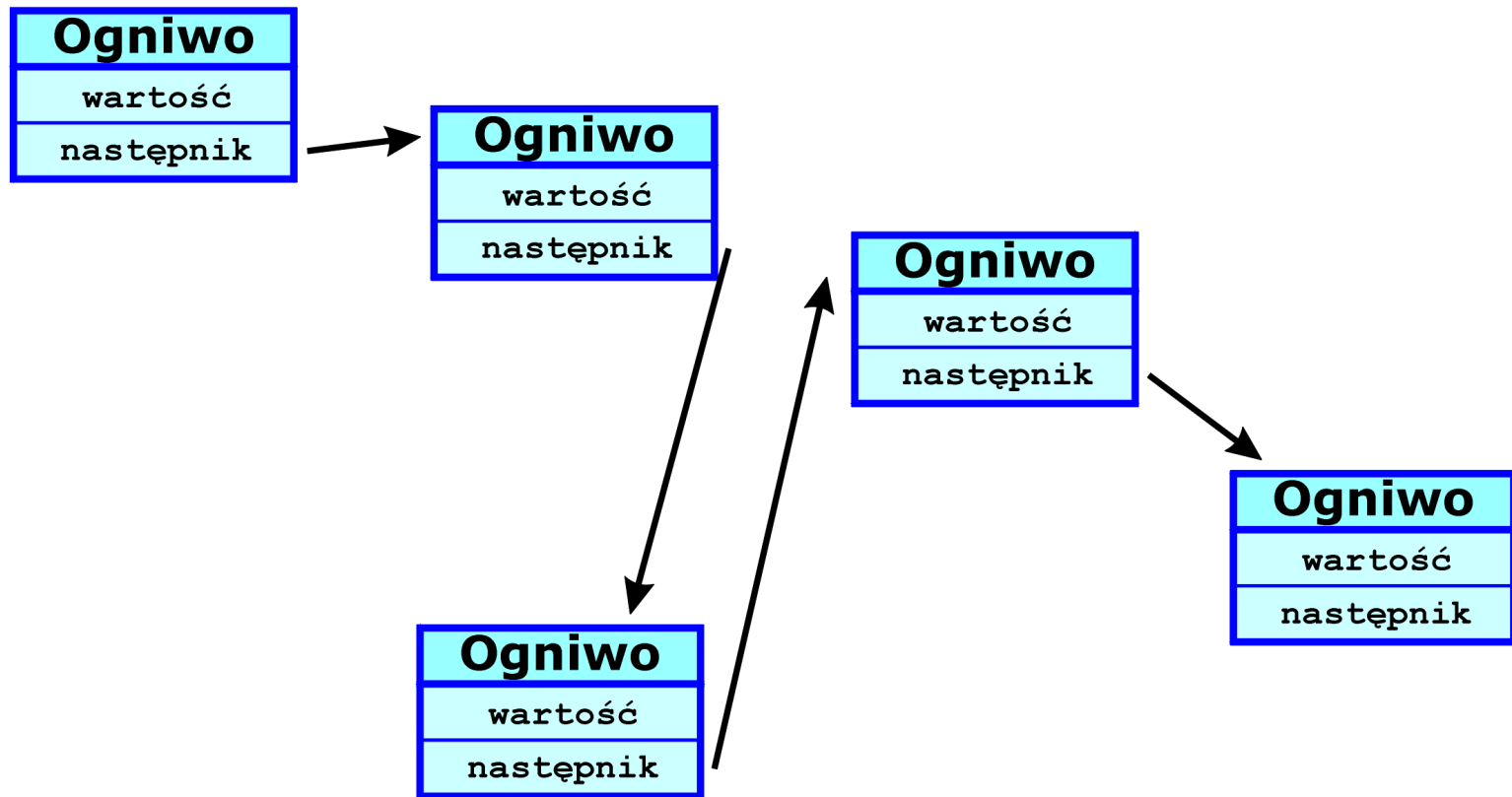


3 slajdy ukazują dodawanie elementów do podanej struktury.

# Wiązane struktury danych 8

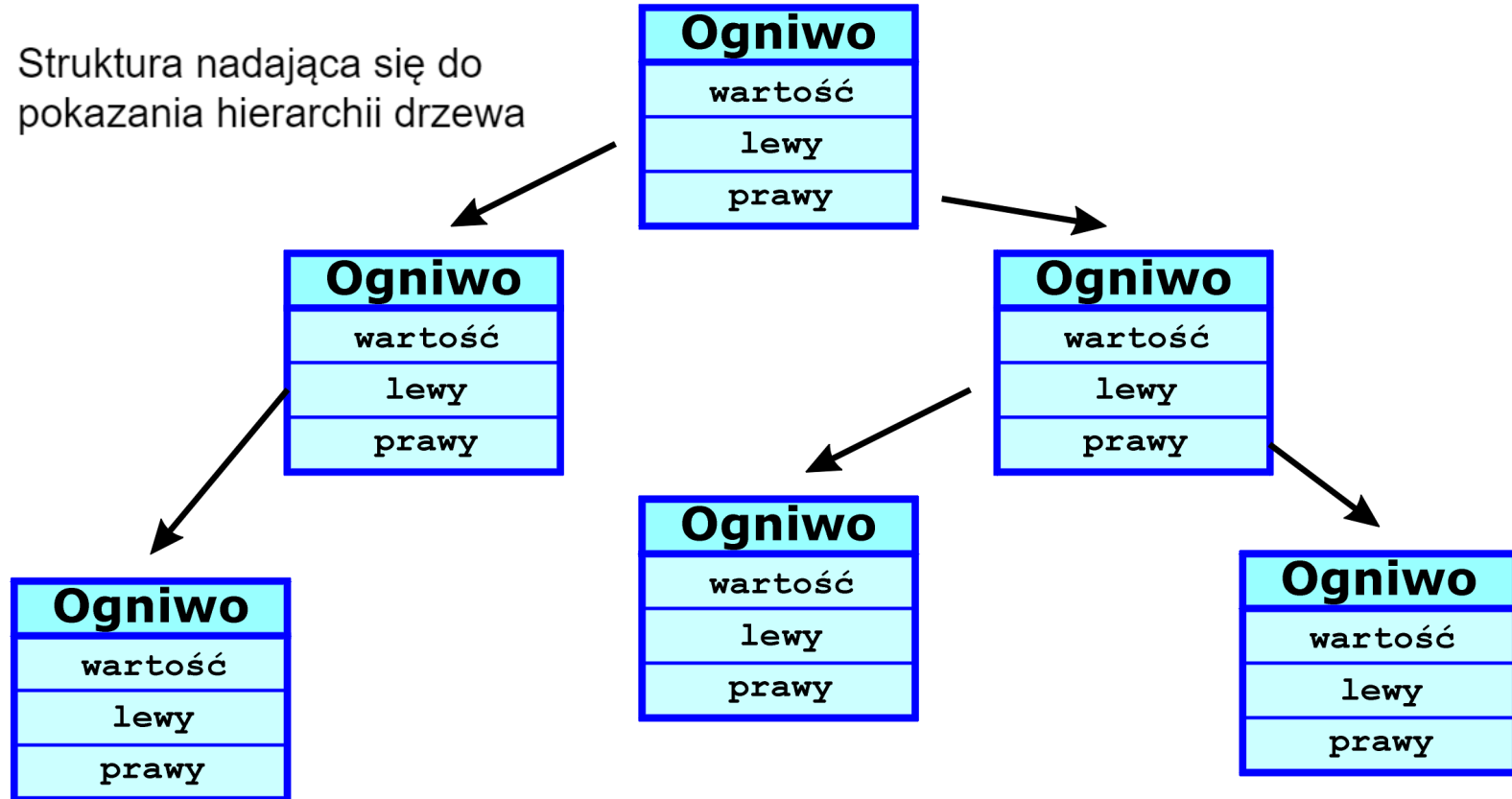


# Wiązane struktury danych 9



# Wiązane struktury danych 10

Struktura nadająca się do pokazania hierarchii drzewa



# Arytmetyka wskaźników <sup>11</sup>

- **arytmetyka wskaźników**: operacje dodawania do wskaźnika liczby, odejmowania i porównywania wskaźników ■
- **pożytek**: bezpośrednie operacje na adresie, co pozwala na tworzenie niskopoziomowego kodu ■
- **kompilator nie jest w stanie sprawdzić poprawności**: arytmetyka wskaźników niebezpieczna ■

- W kodzie generowanym przez kompilator referencja jest traktowana jako zmienna, w której przechowuje się adresy obiektów. ■
- Kod maszynowy wygenerowany w przypadku referencji jest taki sam jak w przypadku użytego w jej miejsce wskaźnika ■
- Na poziomie asemblera nie ma różnicy między referencją, a wskaźnikiem. ■
- Różnica pojawia się przy tłumaczeniu programu przez kompilator i sprowadza się do zastosowania interpretacji bliskiej w odniesieniu do wskaźników, a interpretacji dalekiej w odniesieniu do referencji. ■



# Różnice między wskaźnikami, a referencjami 13

|        | Referencja                                                                                                                                                  | Wskaźnik                                                                                                                                            |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Zalety | <ul style="list-style-type: none"><li>• nie wymaga operatora wyłuskania ■</li><li>• nie grozi zapisaniem niewłaściwego miejsca w pamięci ■</li></ul>        | <ul style="list-style-type: none"><li>• umożliwia arytmetykę wskaźników ■</li><li>• pozwala na wskazywanie na dowolne miejsce w pamięci ■</li></ul> |
| Wady   | <ul style="list-style-type: none"><li>• nie umożliwia arytmetyki wskaźników ■</li><li>• nie pozwala na wskazywanie na dowolne miejsce w pamięci ■</li></ul> | <ul style="list-style-type: none"><li>• wymaga operatora wyłuskania ■</li><li>• grozi zapisaniem niewłaściwego miejsca w pamięci ■</li></ul>        |

# Wskaźniki w miejsce referencji <sup>14</sup>

- referencję zawsze można zastąpić wskaźnikiem ■
- lepiej tego nie robić:
  - arytmetyka wskaźników niepotrzebna, ale przez pomyłkę może zostać użyta i to w sposób niebezpieczny ■
  - konieczny operator wyłuskania ■

Wskaźniki należy stosować tylko wtedy gdy jest to absolutnie konieczne do osiągnięcia wysokiej wydajności kodu, bo używanie wskaźników grozi wygenerowaniem trudnych do zlokalizowania błędów.

# Typy referencyjne <sup>15</sup>

- odnośniki w formie pośredniej — w zależności od kontekstu stosowana jest albo interpretacja bliska lub daleka ■
- terminologia: wskaźniki lub referencje, w zależności od tego, która interpretacja jest stosowana częściej. ■
- przykład: typy referencyjne ■
- typ referencyjny: odnośnik jest jedynym sposobem dostępu do obiektu ■
- zmienna typu referencyjnego: w rzeczywistości odnośnik, któremu zazwyczaj najbliżej do referencji ■
- Java:

```
Samochod pierwszy=new Samochod();
```

■

- dwa rodzaje spojrzenia:
  - niskopoziomowe: obiekt traktujemy jako odnośnik ■
  - wysokopoziomowe: obiekt, będący w istocie odnośnikiem, utożsamiamy z obiektem, na który ten odnośnik pokazuje ■
- 
- na codzień spojrzenie wysokopoziomowe ■
- subtelności w sposobie postępowania z typami referencyjnym łatwiej zrozumieć poprzez spojrzenie niskopoziomowe: zwrot **traktowany jako odnośnik**. ■

## Wartość null<sup>17</sup>

- wartość specjalna: odnośnik nie pokazuje na żaden obiekt ■
- w C++ jest to 0, w Javie i C# **null** ■
- Java i C#: wartość domyślna, gdy tworzymy obiekt typu referencyjnego i nie inicjalizujemy go ■
- Java, C#: instrukcja nie tworzy obiektu!

Samochod drugi;

■

# Porównanie interpretacji typów odnośnikowych w C++, C# i Java<sub>18</sub>

Podstawowe sytuacje, w których występują różnice w interpretacji odnośników zebrane są w poniższej tabeli.

|               | Wsk. | Wsk. | Typ ref.                     | Typ ref. | Ref. |
|---------------|------|------|------------------------------|----------|------|
| Język         | C++  | C#   | Java                         | C#       | C++  |
| inicjalizacja | B    | B    | B                            | B        | B    |
| =             | B    | B    | B                            | B        | D    |
| ==, !=        | B    | B    | B                            | D        | D    |
| <, >, <=, >=  | B    | B    | —                            | D        | D    |
| +, -, ++, --  | B    | B    | — (D dla + w <b>String</b> ) | D        | D    |
| .             | —    | —    | D                            | D        | D    |
| —>            | D    | D    | —                            | —        | —    |

B — interpretacja bliska, D — interpretacja daleka ■

- zmienne lokalne, globalne i tablice to za mało:
  - wielkość zwykłej tablicy musi być znana w momencie kompilacji ■
  - czas życia obiektu nie pasuje ani do zmiennych globalnych ani do lokalnych ■
- potrzebne obiekty, które mogą być powoływać do życia i likwidowane w dowolnym czasie ■
- problem: jak przydzielać pamięć
  - zmienne globalne: źle, czas życia zmiennej to czas życia programu ■
  - zmienne lokalne: źle, czas życia od wejścia do bloku do wyjścia z bloku ■

**Sberta**: obszar pamięci przeznaczony dla zmiennych dowolnego typu, które mogą być w dowolnym momencie tworzone i likwidowane ■



# Tworzenie i likwidacja obiektów na stercie 4

- bezpośrednio przed utworzeniem obiektu przydzielenie pamięci ■
- przydział w miarę wolnego miejsca ■
- obiekt likwidowany zwalnia pamięć ■
- brak likwidacji prowadzi do przepełnienia sterty ■
- **wyciek pamięci**: błąd w programie polegający na stopniowym zapełnianiu, a wreszcie przepełnieniu pamięci sterty na skutek braku likwidacji niepotrzebnych obiektów ■
- **poszatkovanie sterty**: wolne miejsce sumarycznie duże, składa się z dużej ilości małych porcji ■

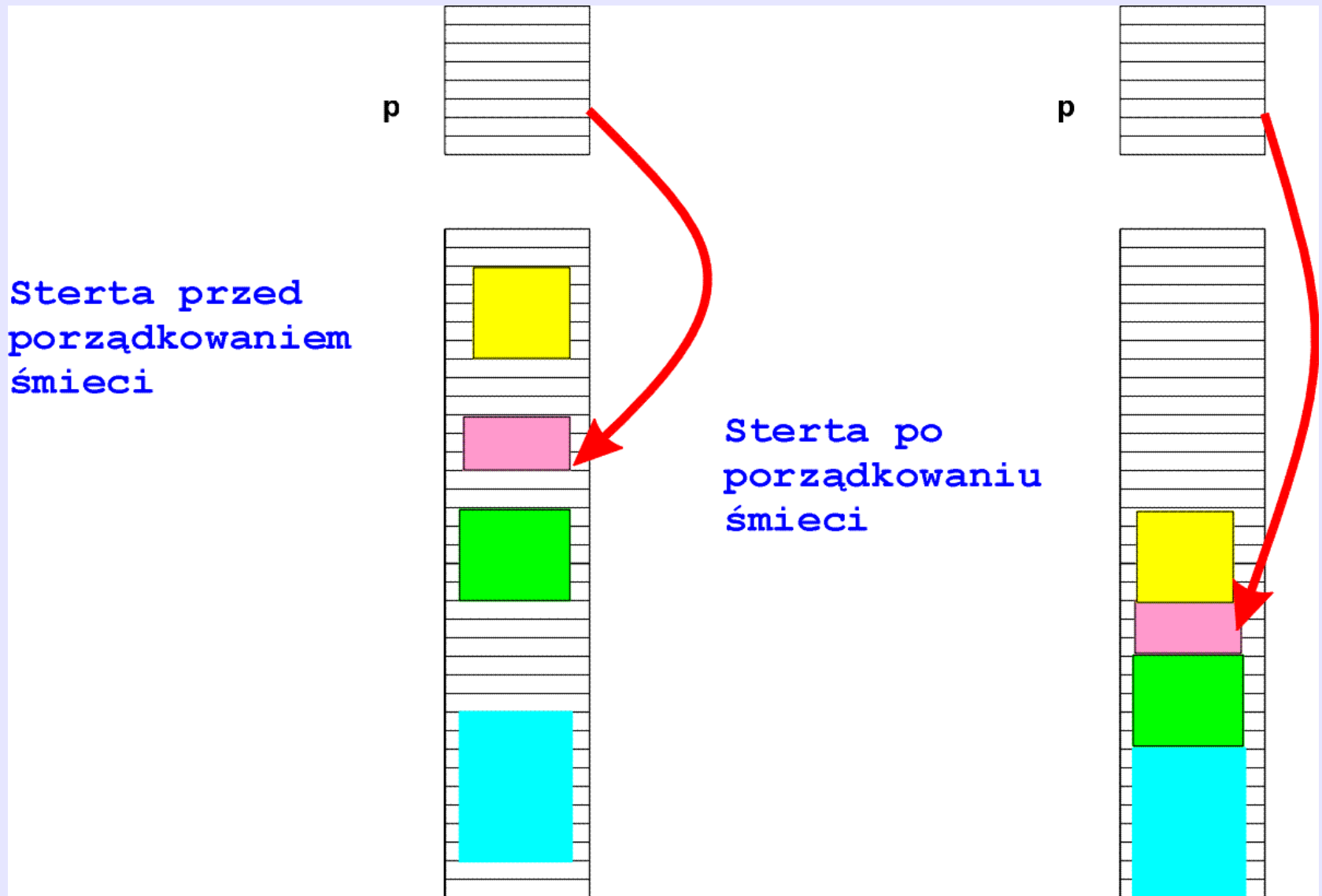
# Tworzenie i likwidacja obiektów na sterce 5

- Obiektom tworzonym na sterce nie nadaje się nazwy ■
- Dostęp do takich obiektów jest tylko poprzez odnośniki ■

# Sterna z automatycznym odśmiecaniem 6

- obiekt istnieje co najmniej tak długo jak długo istnieje choć jedna referencja do niego ■
- mechanizm obsługi steru sam decyduje kiedy należy zwolnić pamięć ■
- w sytuacji poszatkowania realokacja obiektów ■
- w konsekwencji konieczność modyfikacji wszystkich odnośników pokazujących na obiekty na sterze ■
- proces określany mianem **garbage collection** (odśmiecanie) ■
- cena: obniżenie wydajności ■
- dostępna w C#, Javie, językach skryptowych, ale nie w C/C++ ■

# Sterta z automatycznym odśmiecaniem 7



# Jak może działać automatyczny odśmiecacz? <sup>8</sup>

Odśmiecacz potrzebuje pomysłu na

- odszukiwanie niepotrzebnych obiektów ■
- przeadresowywanie referencji ■

W tym celu stosowane są różne metody

- Zapamiętywanie referencji wskazujących na obiekt
  - jest powolna ■
  - nie wykrywa pętli w referencjach ■

■

- Analiza ścieżek dostępu wychodzących ze stosu i pamięci statycznej ■

# Wskaźniki<sub>3</sub>

- **Wskaźnik** to zmienna, która pokazuje na inną zmienną (przechowuje adres zmiennej)

- 

```
int* p;
```



- Wskaźnikom przypisujemy wartości poprzez operator &

```
int n;
p=&n;
```



# Wskaźniki<sub>4</sub>

Wykorzystanie poprzez operator wyłuskania:

```
int n;
p=&n;
*p=12;
cout << n;
```

- Wyrażenie postaci **\*p**, gdzie **p** jest wskaźnikiem pokazującym na zmienną **x** może być wykorzystane wszędzie tam gdzie można użyć **x**. ■
- W szczególności, **\*p** można użyć po lewej stronie operatora **=**.

**Ćwiczenie.** Napisz krótki program, który deklaruje dwa wskaźniki: jeden do zmiennej typu int, drugi do zmiennej typu float, a następnie przypisuje wskaźnikom wartości i wypisuje ich wartości (adresy) oraz wartości zmiennych, na które wskaźniki pokazują.

Ćwiczenie zapisane jako przykład54.cpp



## Wskaźniki<sub>6</sub>

```
int main(){
 int n=3;
 int* p=&n;

 (*p)=5;
 cout << "n=" << n << ", *p=" << *p;
 cout << ", p=" << p << "\n";

 (*p)--;
 cout << "n=" << n << ", *p=" << *p;
 cout << ", p=" << p << "\n";
}
```

n=5, \*p=5, p=0x22ff1c

n=4, \*p=4, p=0x22ff1c

# Deklaracja wskaźnika <sup>7</sup>

- formy równoważne

```
int* p;
int * p;
int *p;
```



- Pierwsza konwencja: uwypukla nowy typ
- Niestety nie działa, gdy chcemy zadeklarować więcej wskaźników w jednej deklaracji.

```
int* p, q;
```



- Trzeba napisać:

```
int *p, *q;
```

# Deklaracja wskaźnika <sup>8</sup>

- Zapis **int\*** ma jednak swoje zalety ■
- Najlepiej pisać **int\*** i używać oddzielnej deklaracji dla każdego wskaźnika

```
int* p;
int* q;
```



- Można też użyć skrótu:

```
typedef int* IntPtr;
```



- Używając skrótu można zadeklarować kilka wskaźników w jednej deklaracji.

```
IntPtr p, q;
```

# Arytmetyka wskaźników<sup>9</sup>

```
int n;
p= &n;
p++;
```

```
int n[20], *p, *q;
p=&n[3];
q=&n[7];
cout << q - p;
```

```
int main(){
 char c[]="programming";
 cout << "-";
 for(char* q=&c[0]; q<=&c[10]; ++q){
 cout << *q << "-";
 *q-=32;
 }
 /* We test modification effect */
 cout << "\n\n";
 for(char* q=&c[0]; q<=&c[10]; ++q){
 cout << *q << "-";
 }
}
```

Pętla wyświetla dane z tablicy, ponieważ adresy tablicy są ustawione obok siebie.

**Ćwiczenie.** Napisz program, który wczytuje ze standardowego wejścia 10 liczb całkowitych i zapisuje je w tablicy, a następnie przegląda tablicę wykorzystując wskaźniki w celu znalezienia największej liczby w tablicy.

Ćwiczenie zapisane jako przyklad55.cpp

```
int main(){
 /* We cannot assume, that the variables
 declared jointly share nearby places */

 int k=1,n=3,m=7;
 int* p=&n;

 cout << "k=" << k << "\n";
 cout << "n=" << n << "\n";
 cout << "m=" << m << "\n";
 cout << "*(p-1)=" << *(p-1) << "\n";
 cout << "*p=" << *p << "\n";
 cout << "*(p+1)=" << *(p+1) << "\n";
}
```

przyklad56.cpp



- Kod generowany przez kompilator nie sprawdza, czy wskaźnik rzeczywiście pokazuje na obiekt właściwego typu oraz czy wpisywanie w to miejsce jest sensowne i bezpieczne. ■
- Wskaźniki w C/C++ dają programiście pełną kontrolę nad całą pamięcią. Pozwala to na pisanie bardzo szybkich programów. Jednak znacząco zwiększa to ryzyko pomyłek skutkujących trudnymi do zlokalizowania błędami. ■

# Porównywanie wskaźników <sup>15</sup>

- Wskaźniki mogą być porównywane przy użyciu operatorów `==`, `!=`, `<`, `>`, `<=`, `>=` ■
- – wskaźniki są równe, gdy pokazują na ten sam obiekt ■
  - wskaźniki są różne, gdy pokazują na różne obiekty ■
  - Jeśli dwa wskaźniki pokazują na obiekty tej samej tablicy, to wskaźnik o mniejszej wartości pokazuje na element tablicy o mniejszym indeksie. ■
- konwencja: żaden obiekt nie jest przechowywany pod adresem zero. ■
- wskaźnik ustawiony na zero oznacza, że w danym momencie wskaźnik nie pokazuje na żaden obiekt. ■

# Tablica jako wskaźnik <sup>16</sup>

- Nazwa tablicy jest synonimem adresu elementu tablicy o indeksie zero
- 

```
int t[]={0,1,2,3,4,5};
cout << *(t+3),
```



- Nazwa tablicy może być traktowana jako stały wskaźnik: jego wartość nie może być zmieniona
- notacja tablicowa jest możliwa przy użyciu dowolnego wskaźnika

```
int t[]={0,1,2,3,4,5};
int* p=t+1;
cout << p[2];
```



## Tablica jako wskaźnik 17

**Ćwiczenie.** Napisz program, który deklaruje tablicę **a** o pojemności 20 liczb typu **int** i przechowuje  $2i^2 + 1$  w elemencie tablicy o indeksie  $i$ . Wykorzystaj wskaźniki do przypisywania wartości elementom tablicy. Następnie zadeklaruj inny wskaźnik pokazujący na  $a[10]$ . Użyj go jako nazwy tablicy i, wykorzystując składnię tablicy napisz pętlę, która drukuje elementy  $a[10], a[11], \dots, a[14]$ .

Zarezerwowane: przyklad57.cpp

## Tablica jako wskaźnik <sup>18</sup>

```
int main(){
 int t[20];

 // Use an array as a pointer
 for(int i=0; i<20; i++) *(t+i)=i*i;

 int* p=&t[0];

 // Use a pointer as an array
 for(int i=0; i<20; i++){
 cout << i << "^2=" << p[i] << "\n";
 }
}
```

# C/C++ Referencje <sup>19</sup>

- **Referencja** w C++ jest czymś w rodzaju wskaźnika, który udaje, że jest zmienną ■
- uproszczony wygląd: na referencję można patrzeć jak na dodatkową nazwę zmiennej.■
- Ale: czasami jest to jedyna nazwa obiektu (obiekty na stercie) ■
- 

```
int n;
int& r=n;
```

■

- możliwa interpretacja **int&**: nowy typ, pochodny od typu **int** ■
- te same ograniczenia co przy wskaźnikach ■

- Referencja musi być zainicjalizowana w miejscu deklaracji i nie może być później zmieniona. ■
- Referencja może być użyta wszędzie tam gdzie zmienna. ■

**Ćwiczenie.** Napisz program, który deklaruje dwie referencje: jedną dla zmiennej typu **int**, a drugą dla typu **double**, przypisuje im wartości początkowe, a następnie wypisuje wartości zmiennych wskazywanych przez referencje. (via the references)

Ćwiczenie zapisane jako przykład58.cpp

```
int main(){
 int n=3;
 int& r=n;

 ++r;
 cout << n << "\\n";

}
```



# Wskaźniki i referencje, a modyfikator **const** 22

- **const** przed deklaracją:
  - wskaźnik tylko do odczytu:

```
float x;
const float* wsk=&x;
```



- referencja tylko do odczytu:

```
float x;
const float& ref=x;
```



# Wskaźniki i referencje, a modyfikator **const** 23

- modifier **const** po znaku \* ale przed nazwą wskaźnika:

```
float x;
float* const wsk=&x;
```



- **stały wskaźnik** - zawsze pokazuje na ten sam obiekt ■
- efekt jest zbliżony do użycia referencji, ale w przeciwieństwie do referencji konieczny jest operator \* (operator wyłuskania) ■

Definition wskaźnika tylko do odczytu syntaktycznie różni się od stałego wskaźnika jedynie lokalizacją znaku \* w odniesieniu do **const**.

# Użycie wskaźników i referencji do tworzenia obiektów na stercie <sup>24</sup>

- operator **new**: do tworzenia obiektów na stercie ■

- 

```
double* q=new double;
```

■

- Aby użyć obiektu potrzebujemy operatora wyłuskania

```
*q=1.5;
```

■

- Pozbycie się obiektu na stercie, gdy nie jest już potrzebny:

```
delete q;
```

■

# Użycie wskaźników i referencji do tworzenia obiektów na stercie<sup>25</sup>

- To samo z referencjami:

```
double& q=*new double;
q=1.5;
....
delete &q;
```




# Użycie wskaźników i referencji do tworzenia obiektów na stercie<sup>26</sup>

- stworzenie tablicy na stercie:

```
double* q=new double[n]
q[5]=1.12;
```



- Wielkość tablicy na stercie nie musi być znana w chwili kompilacji! 
- Zwolnienie pamięci

```
delete[] q;
```



# Brak wolnej pamięci na stercie <sup>27</sup>

- jeśli zabraknie wolnego miejsca na stercie ...
  - C: zwracany adres zero ■
  - C++: rzucony tzw. wyjątek ■

**Ćwiczenie.** Napisz program, który metodą prób i błędów sprawdza jakie jest największe  $n$ , takie, że sarta pomieści tablicę typu **double** o  $2^n$  elementach. Pamiętaj o zwolnieniu pamięci przed ponowną jej rezerwacją.

Zarezerwowane: przyklad59.cpp

# Funkcje rekurencyjne <sub>3</sub>

- funkcja może wywoływać sama siebie: **wywołanie rekurencyjne** ■■
- wywołanie musi być warunkowe, w przeciwnym razie wywołania zapętliłyby się ■
- typowe zastosowanie: ciągi definiowane rekurencyjnie

$$0! := 1$$

$$n! := (n - 1)! * n \text{ dla } n \geq 1$$



## Silnia rekurencyjnie <sup>4</sup>

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

```
int main(){
 for(int i=1;i<=10;++i){
 std::cout << "f[" << i << "]= " << factorial(i)
 << std::endl;
 }
}
```

przyklad60.cpp



# Funkcje rekurencyjne <sub>5</sub>

**Ćwiczenie.** Napisz funkcję rekurencyjną, która oblicza  $n$ ty wyraz ciągu zdefiniowanego rekurencyjnie wzorami

$$\begin{aligned}c_0 &:= 1 \\ c_n &:= (1 + c_{n-1})^2 \text{ dla } n \geq 1\end{aligned}$$

# Funkcje rekurencyjne <sub>6</sub>

- Ciąg Fibonacciego

$$\begin{aligned} f_0 &:= 0, \quad f_1 := 1 \\ f_n &:= f_{n-1} + f_{n-2} \text{ dla } n > 1 \end{aligned}$$

# Fibonacci rekurencyjnie<sub>7</sub>

```
int fib(int n){
 std::cout << "Evaluating fib for n=" << n
 << std::endl;
 if(n<=0) return 0;
 if(n==1) return 1;
 return fib(n-2)+fib(n-1);
}
```

przyklad61.cpp

## Fibonacci iteracyjnie<sub>8</sub>

```
int fib(int n){
 if(n<=0) return 0;
 if(n==1) return 1;
 int f=1,fp=0;
 for(int i=2;i<=n;++i){
 int t=fp+f;
 fp=f;
 f=t;
 }
 return f;
}
```

# Zalety i wady wywołania rekurencyjnego 9

- Zaleta: prostota, łatwość analizy kodu ■
- Wada: jest wolniejsze i wymaga więcej pamięci ■

**Ćwiczenie.** Napisz funkcję, która przyjmuje jako argument liczbę naturalną  $n$  i drukuje wszystkie sposoby posortowania elementów zbioru  $\{1, 2, \dots, n\}$ .

Zarezerwowane przyklad62.cpp

# Argumenty funkcji `main` <sup>11</sup>

- Funkcja **`main`** może być wywołana z następującymi argumentami

```
int main(int argc, char* argv[])
```



- **`argc`**: liczba łańcuchów znakowych w linii wywołania programu z wliczeniem nazwy programu ■
- **`argv`**: tablica wskaźników do łańcuchów znakowych przeczytanych z linii komend w kolejności pojawienia się na tej liście ■

## Argumenty funkcji main<sub>12</sub>

```
int main(int argc, char* argv[]){
 cout << "\nLiczba argumentow linii komend to ";
 cout << argc << "\n\n";
 for(int i=0;i<argc;i++){
 cout << "Slowo " << i << " to: \n ";
 cout << argv[i] << "\n";
 }
}
```



## Argumenty funkcji main<sup>13</sup>

**Ćwiczenie.** Napisz program, który wypisuje wszystkie argumenty z linii komend, które rozpoczynają się od cyfry.

przykład63.cpp

# Przekazywanie argumentów do to funkcji przez wartość <sup>14</sup>

```
int a_function(int i, double f){
 i++;
 f--;
 cout << "Inside: i=" << i << ", f=" << f << "\n";
 return 1;
}
```

```
int main(){
 int j=7;
 double g=1.5;
 cout << "Before: j=" << j << ", g=" << g << "\n";
 a_function(j,g);
 cout << "After: j=" << j << ", g=" << g << "\n";
}
```

# Przekazywanie argumentów do to funkcji przez wartość <sup>15</sup>

- Wykonywane przez skopiowanie argumentu do zmiennej lokalnej ■
- Zaleta: chroni przed niezamierzoną modyfikacją argumentu ■
- Wada: kopiowanie dużych argumentów jest kosztowne czasowo i pamięciowo

# Przekazywanie argumentów do to funkcji przez referencję <sup>16</sup>

```
int a_function(const int& i, const double& f){
 i++;
 f--;
 cout << "Wewnatrz: i=" << i << ", f=" << f << "\n";
 return 1;
}

int main(int argc, char* argv[]){
 int j=7;
 double g=1.5;
 cout << "Przed: j=" << j << ", g=" << g << "\n";
 a_function(j,g);
 cout << "Po: j=" << j << ", g=" << g << "\n";
 return 0;
}
```

przyklad64.cpp

# Przekazywanie argumentów do to funkcji przez referencję <sup>17</sup>

- Wykonywane przez skopiowanie adresu argumentu ■
- Zaleta: oszczędza czas i pamięć w przypadku dużych argumentów, bo kopiowany jest tylko adres.■
- Wada: nie chroni przed przypadkową modyfikacją chyba, że użyto **const**

# Przekazywanie argumentów do to funkcji przez referencję <sup>18</sup>

**Ćwiczenie.** Napisz funkcję, która przyjmuje dwa argumenty i wzajemnie zamienia ich wartości. Wytłumacz co stałoby się przy przekazywaniu argumentów przez wartość

# Przekazywanie argumentów do to funkcji przez wskaźnik <sup>19</sup>

```
int a_function(int* i, double* f){
 (*i)++;
 (*f)--;
 cout << "Inside: *i=" << *i << ", *f=" << *f << "\n";
 return 1;
}

int main()
{
 int j=7;
 double g=1.5;
 cout << "Before: j=" << j << ", g=" << g << "\n";
 a_function(&j,&g);
 cout << "After: j=" << j << ", g=" << g << "\n";
 return 0;
}
```

przyklad65.cpp

# Przekazywanie argumentów do to funkcji przez wskaźnik <sup>20</sup>

- Podobne do przekazywania argumentu przez referencję.■
- Wada: wymaga jawnego użycia operatora wyłuskania.



## Przekazywanie tablic do funkcji <sup>21</sup>

```
void f(int t[]) {
 for(int i=0; i<5; i++){
 t[i]++;
 cout << "Element[" << i << "]= " << t[i] << endl;
 }
}

int main() {
 int x[5] = {3, 2, 1, 2, 3};

 f(x);
 for(int i=0; i<5; i++){
 cout << "x[" << i << "]= " << x[i] << endl;
 }
}
```

Do funkcji jest przesyłana adres początkowy tablicy.

# Przekazywanie tablic do funkcji <sup>22</sup>

- Nie da się przekazać przez wartość ■
- Nagłówek funkcji

```
void f(int t[5])
```

uważany jest za identyczny z nagłówkiem

```
void f(int* t)
```

■

- W obu przypadkach do funkcji przekazywany jest adres początkowego elementu tablicy. ■
- Tak czy owak wielkość tablicy jest pomijana, więc trzeba sobie radzić samemu.

```
void f(int t[])
```

# Przekazywanie tablic do funkcji<sup>23</sup>

- Liczbę elementów przekazujemy jako oddzielny argument.

## Modyfikator **const** przy argumentach funkcji <sup>24</sup>

**const** użyty przy argumentach przekazanych przez wskaźnik lub referencję powoduje, że ten argument nie może być modyfikowany. ■

```
void f1(const int& i, const int* j) {
 i++; // wrong, the compiler will report an error
 *j+=5; // wrong, the compiler will report an error
}
```

## Modyfikator **const** przy argumentach funkcji <sup>25</sup>

```
void f1(const int& i){
 // wrong, the compiler will report an error:
 // i++;
 cout << "Argument value inside " << i << "\n";
}

void f2(const int* ptr){
 // wrong, the compiler will report an error:
 // (*ptr)*=2;
 cout << "Argument value inside " << *ptr << "\n";
}
```

# Modyfikator **const** przy argumentach funkcji <sup>26</sup>

- Użycie **const** musi być konsekwentne. ■
- W przeciwnym razie będzie problem z wywoływanymi funkcjami, które nie zmieniają argumentów, ale tego nie obiecują, co wywołuje konieczność kaskadowych zmian. ■

## Przetładowanie funkcji <sup>27</sup>

```
void write(int i){
 cout << "This is an int: " << i << "\n";
}

void write(float f){
 cout << "This is a float: " << f << "\n";
}

void write(float* f){
 cout << "This is a float: " << *f;
 cout << " transfered by a pointer\n";
}
```

Którą funkcję użyjesz zależy od argumentu jaki podasz

# Pewne formalnie różne typy traktowane są jako jednakowe <sup>28</sup>

Identification:

- **T\*** oraz **T[]** ■
- **T** oraz **const T** ■
- **T** oraz **T&** ■

```
void f(int* x)
void f(int x[]);
```

```
void f(int x);
void f(const int x);
```

```
void f(int x);
void f(int& x);
```



Stosowane są następujące zasady:

- (1) Pełna zgodność typów argumentów lub zgodność na zasadzie utożsamiania typów (np. **int** i **const int**) ■
- (2) zgodność uzyskana przy użyciu konwersji typów zwiększających pojemność np. **bool** na **int** albo **float** na **double** ■
- (3) zgodność uzyskana przy użyciu innych standardowych konwersji np. **int** na **double** ■
- (4) zgodność uzyskana przy użyciu konwersji użytkownika ■

W przypadku gdy dwie lub więcej funkcji daje się dopasować na tym samym poziomie kompilator zgłasza błąd.

**Ćwiczenie.** Napisz trzy wersje funkcji max obliczającej maksimum z trzech liczb oddzielnie dla typów: **int**, **float** i **double**.

przyklad66.cpp

# Zwracanie referencji i wskaźników przez funkcję <sup>31</sup>



```
int& h(int& x) {
 ...
 return x;
}
```

- W ten sposób zwrócić można wskaźniki i referencje otrzymane na liście argumentów lub wskaźniki/referencje obiektów utworzonych przez funkcję na stacku. ■

Nie wolno zwracać wskaźników i referencji do obiektów lokalnych!!!

## Zwracanie referencji i wskaźników przez funkcję <sup>32</sup>

**Ćwiczenie.** Napisz funkcję, która przyjmuje przez referencję dwa argumenty typu **int** i zwraca przez referencję argument, który ma większą wartość. Zdemostruj, że funkcji tej można użyć po lewej stronie argumentu przypisania. Wyjaśnij co dokładnie się wtedy dzieje.

Zarezerwowano: przyklad67.cpp

## Argumenty domyślne <sup>33</sup>

```
int volume(int l, int w=2, int h=1);
```

Wywołanie

```
int obj=volume(7);
```

jest równoważne wywołaniu

```
int obj=volume(7,2,1);
```

# Argumenty domyślne <sup>34</sup>

- Efekt jest równoważny zdefiniowaniu kolekcji przeładowanych funkcji ■
- Definicje tych funkcji są tworzone automatycznie przez kompilator poprzez dodanie stosownych instrukcji inicjalizujących. ■

- Jeśli podaliśmy domyślną wartość argumentu, musimy też podać domyślną wartość dla wszystkich kolejnych argumentów. ■
- Jeśli pominiemy argument przy wywołaniu funkcji, musimy też pominąć kolejne argumenty.

## Funkcje **inline** 35

Poprzedzając nagłówek funkcji słowem kluczowym **inline** wyrażamy życzenie, by kompilator przepisał ciało funkcji w miejscu wywołania. ■

```
inline double Fahrenheit2Celsius(double f){
 return (f - 32)/9 * 5;
}
```

- jest to tylko sugestia dla kompilatora ■
- użyteczne tylko dla krótkich funkcji

# Wskaźniki do funkcji <sup>36</sup>

- Kompilator identyfikuje nazwę funkcji z jej adresem w pamięci. ■
- Używając tylko nazwy funkcji bez nawiasów i argumentów, odnosimy się do adresu funkcji ■
- Aby wywołać funkcję potrzebujemy nawiasów() ■
- Adres można przechowywać w specjalnym wskaźniku ■



Wskaźnik do funkcji deklarujemy pisząc

```
int (*fptr)(int);
```

Do wskaźnika do funkcji możemy przypisać tylko nazwę funkcji której lista argumentów i typ zwracanego wyniku są zgodne z typem wskaźnika.

```
int f(int i){
 cout << "This is function f\n";
 return i+1;
}
int g(int i){
 cout << "This is function g\n";
 return i+2;
}
```

## Wskaźniki do funkcji <sup>39</sup>

```
int main(){
 int (*fptr)(int)=f;
 cout << (unsigned int)f << endl;
 cout << (unsigned int)fptr << endl;
 cout << (unsigned int)*fptr << endl;
 int i=3;
 // We call the function indicated by the pointer,
 // that is function f
 cout << fptr(i) << endl;
 cout << (*fptr)(i) << endl;
 // We set the pointer on function g
 // and we repeat the call
 fptr=g;
 cout << (unsigned int)fptr << endl;
 cout << fptr(i) << endl;
}
```

# Deklarowanie funkcji <sup>40</sup>

Jeśli chcemy umieścić funkcję w osobnym pliku, aby skompilować ją oddzielnie od programu głównego, musimy powiedzieć programowi głównemu, jaki jest nagłówek funkcji. Robimy to, pisząc deklarację funkcji. Składa się ona z nagłówka funkcji (typ zwracany, nazwa i lista argumentów ujęte w nawiasy), a następnie średnika zamiast treści funkcji.

Nagłówkiem funkcji

```
double square(double x){
 return x*x;
}
```

jest

```
double square(double x);
```

# Preprocesor <sub>3</sub>

**Preprocesor**: służy do wstępnego przetwarzania pliku źródłowego programu ■

Dyrektywy preprocesora są przetwarzane przed uruchomieniem kompilatora.

Składnia:

*#dyrektywa argumenty*

- brak średnika! ■
- ukośnik potrzebny do kontynuacji linii ■
- szeroko używany w C ■
- w C++ jego rola jest ograniczona. Wiele konstrukcji używanych w C w C++ jest traktowanych jako zaszłość ■

# Makrodefinicje <sup>4</sup>

syntax:

```
#define nazwa definicja
```

```
#define nazwa (argumenty_formalne) definicja
```

- name: dowolny identyfikator
- zwyczajowo: wielkie litery
- semantyka: wywołanie zostaje zastąpione definicją
- w przypadku argumentów: każde wystąpienie nazwy argumentu w definicji jest zastępowane tekstem podanym w wywołaniu makrodefinicji

# Makrodefinicje <sub>5</sub>

- W C++ w większości przypadków makra mogą i powinny być zastąpione przez bardziej sprytne **szablony**. ■
- Szablony są podobne do makr, ale są przetwarzane bezpośrednio przez kompilator, a nie przez preprocesor. ■
- W ten sposób można uniknąć wielu błędów spowodowanych brakiem właściwej współpracy między preprocesorem a kompilatorem.

# Zastąpienie argumentu łańcuchem znaków <sub>6</sub>

- literał łańcuchowy jest wstawiany w miejsce argumentu poprzedzonego # ■
- makro

```
#define SHOW(x) cout << #x << "=" << x << "\n" ;
int i=7;
SHOW(i)
```

efekt na standardowym wyjściu

```
i=7
```

■

**Ćwiczenie.** Napisz makro, którego można użyć do zadeklarowania zmiennej i jednocześnie wydrukować informację na standardowym wyjściu, że zmienna jest deklarowana.



# Klejenie identyfikatorów <sup>7</sup>

- Podwójny krzyżyk `##` jest konwertowany na pusty tekst ■
- zastosowanie: tworzenie identyfikatorów poprzez klejenie nazw ■
- przykład: zadeklarowanie zestawu powiązanych zmiennych

```
#define TEXT(a) char *a##_pointer; int a##_length;
TEXT (alpha);
TEXT (beta);
TEXT (gamma);
```

# Klejenie identyfikatorów <sub>8</sub>

- preprocesor nie sprawdza, czy argumenty są sensowne:

```
char* "alpha"_pointer; int "alpha"_length
```



**Ćwiczenie.** Napisz makro, które deklaruje dwie powiązane zmienne: tablicę i liczbę całkowitą do przechowywania liczby elementów w tablicy.

# Zastosowanie makrodefinicji w C++<sup>9</sup>

- aby zdefiniować puste nazwy do sterowania kompilatorem

```
#define INTEL
```

- dyrektywy kompilacji warunkowej

```
#if warunek
 // pomija kompilację tego fragmentu, jeżeli warunek
 nie jest spełniony
#endif
```

- warunek: decyzja musi być możliwa w czasie kompilacji ■

# Dyrektywy kompilacji warunkowej<sup>10</sup>

wariant z **else**

```
#if warunek
 // linie kompilowane gdy warunek zachodzi
#else
 // linie kompilowane gdy warunek nie zachodzi
#endif
```

# Testowanie obecności i usuwanie nazw makrodefinicji <sup>11</sup>

Można sprawdzić, czy makrodefinicja jest zdefiniowana przez

```
defined(nazwa)
```



Makrodefinicja może zostać usunięta przez

```
#undef nazwa
```

# Testowanie obecności i usuwanie nazw makrodefinicji <sup>12</sup>

Czasami chcemy skompilować część programu tylko wtedy, gdy nazwa nie jest zdefiniowana. Następnie możemy użyć dyrektywy **#if** w formularzu

```
#if !defined (nazwa)

#endif
```

albo możemy użyć dyrektywy **#ifndef**

```
#ifndef nazwa

#endif
```

## Przerwanie kompilacji <sup>13</sup>

Poniższa dyrektywa może być użyta do przerywania kompilacji i wyświetlenia komunikatu o błędzie

```
#error tekst
```

**Ćwiczenie.** Napisz program, który przerywa kompilację, jeśli zdefiniowane jest makro o nazwie BORLAND

```
//#define _BORLAND
int main(){
 int i=7,j=14,k=3;

 #if defined(_BORLAND)
 #error keyword bitand does not work with Borland C++
 k=i & j;
 #else
 k=i bitand j;
 #endif
 cout << "k=" << k << "\n";
}
```



# Dyrektywa `include` 15

Dyrektywa **`#include`** pozwala na włączenie do kompilowanego pliku zawartości innego pliku. ■

syntax

```
#include <file_name>
```

```
#include "file_name"
```



- te dwie formy różnią się kolejnością przeszukiwania pliku przez foldery ■
- Konkretna kolejność zależy od kompilatora ■

# Jednokrotne włączenie pliku <sup>16</sup>

- złożony zestaw relacji między dołączonymi plikami: jeden plik może zostać błędnie dołączony więcej niż raz podczas jednej kompilacji ■
- to może prowadzić do błędów ■
- trik

```
#if !defined(_FILE_)
#define _FILE_
 // zawartość pliku
#endif
```

## Predefiniowane nazwy <sup>17</sup>

- `_ _ LINE _ _` — Numer linii, przy której aktualnie działa preprocesor ■
  - `_ _ NAME _ _` — nazwa aktualnie kompilowanego pliku ■
  - `_ _ DATE _ _` — data kompilacji ■
  - `_ _ TIME _ _` — czas kompilacji ■
- 

**Ćwiczenie.** Napisz program, który wypisze na standardowym wejściu datę i godzinę jego skompilowania.

**Ćwiczenie.** Napisz makro, które deklaruje zmienną i wypisuje linię, w której zmienna została zadeklarowana.