Maciej Sobieraj

Wykład 2

Outline

1. **Kontrola przepływu i typy danych cd.**
2. Rozszerzanie ekspresyjnej mocy: wskaźniki, funkcje i pamięć

Tablice

- Dlaczego?
  - Być może musimy czytać, przechowywać, przetwarzać i na koniec drukować dziesiątki, a może setki, a może tysiące liczb.

```
int var1, var2, var3, var4, var5, var5, var7, var8, var9;
```

- zapisz **pięć** wartości typu *int*
  - elementy w tablicy są ponumerowane **od 0**

```
int numbers[5];
```

Tablice

- Przypisywanie wartości do wybranego elementu tablicy

```
numbers[0] = 111;
```

- Wartość przechowywana w trzecim elemencie tablicy

```
i = numbers[2];
```

Tablice

- Suma **wszystkich wartości** przechowywanych w tablicy liczb

```
int numbers[5], sum = 0;

for(int i = 0; i < 5; i++)
   sum += numbers[i];
```

- Przypisywanie tej samej wartości (np. 2012) do wszystkich elementów tablicy

```
int numbers[5];

for(int i = 0; i < 5; i++)
   numbers[i] = 2012;
```

Tablice

- Co robi poniższy kod?

```
for(int i = 0; i < 2; i++) {
   auxiliary = numbers[i];
   numbers[i] = numbers[4 – i];
   numbers[4 – i] = auxiliary;
}
```

Inicjalizacja tablicy

- Inicjator wektorowy jest po prostu listą wartości zamkniętych w **nawiasach klamrowych**.

```
int vector[5] = { 0,1,2,3,4 };
```

- Nie określiliśmy rozmiaru tablicy, ale **dostarczyliśmy inicjatora**.

```
int vector[] = { 0,1,2,3,4,5,6 };
```

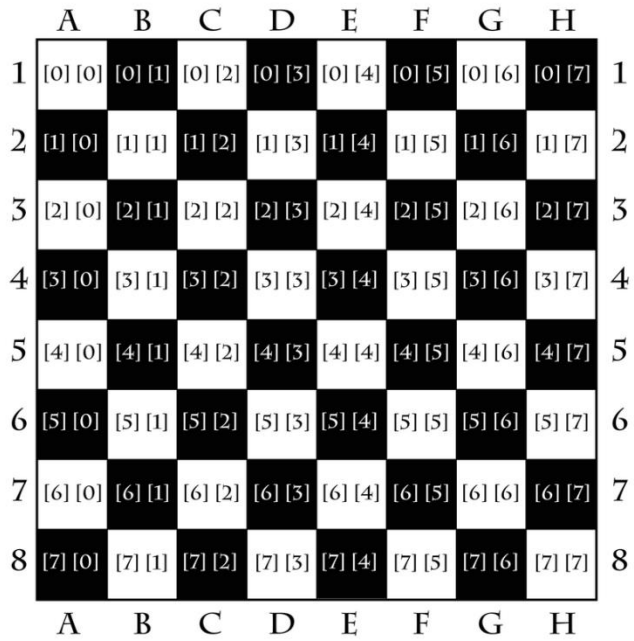
Nie tylko **int**

```
float FloatArr[10];

char surname[20];

bool votes[100];
```

Nie tylko wektory

- dwuwymiarowa tablica

```
int chessboard[8][8];
```



Nie tylko wektory

- Urządzenie rejestruje temperaturę powietrza co godzinę i robi to przez cały miesiąc.
- Daje nam to w sumie 24 * 31 = 744 wartości.
  - **float** temp[31][24];
  - **float** sum = 0.0, average;
  - **for**(**int** day = 0; day < 31; day++)
  - sum += temp[day][11];
  - average = sum / 31;
  - cout << "Average temperature at noon: " << average << endl;

Nie tylko wektory

```
float temp[31][24];
float max = -100.0;

for(int day = 0; day < 31; day++)
   for(int hour = 0; hour < 24; hour++)
      if(temp[day][hour] > max)
         max = temp[day][hour];
cout << "The highest temperature was " << max << endl;
```

Nie tylko wektory

```
float temp[31][24];
int hotdays = 0;

for(int day = 0; day < 31; day++)
    if(temp[day][11] >= 20.0)
        hotdays++;
cout << hotdays <<" days were hot.";
```

Nie tylko wektory

```
float temp[31][24];
int d,h;

for(d = 0; d < 31; d++)
    for(h = 0; h < 24; h++)
        temp[d][h] = 0.0;
```

Nie tylko wektory

- Język "C ++" nie ogranicza liczby wymiarów tablicy.

```
int guests[3][15][20];
```

Struktury - dlaczego ich potrzebujemy?

- **string** to trochę więcej niż typ
- Zmienne typu **string** mogą być przypisane do tych samych operatorów, co każda inna zmienna

```
string student_name[100000];
```

- Załóżmy na przykład, że pierwszym zarejestrowanym uczniem jest Mr. Bond (James Bond).
  - student_name[0] = "Bond";

Struktury - dlaczego ich potrzebujemy?

```
float student_time[100000];
```

- Mr. Bond spędził trzy godziny i trzydzieści minut na studiowaniu naszego kursu.
  - student_time[0] = 3.5;

Struktury - dlaczego ich potrzebujemy?

- Głównym problemem jest to, że dane dotyczące tego samego obiektu (studenta) są **rozproszone między trzema zmiennymi**, chociaż logicznie powinny istnieć jako skonsolidowana jednostka.
- Czy możemy użyć agregatu, którego elementy mogą być różnych typów?
  - Struktura może zawierać dowolną liczbę dowolnych elementów dowolnego typu.

Struktury - dlaczego ich potrzebujemy?

```
struct STUDENT {
    string  name;
    float   time;
    int     recent_chapter;
};
```

- Deklaracja struktury →
  - deklaracja struktury zawsze zaczyna się od słowa kluczowego **struct**
  - po słowie kluczowym znajduje się tak zwany znacznik struktury (STUDENT w tym przypadku); to nazwa samej struktury; istnieje powszechnie przyjęty zwyczaj tworzenia znaczników strukturalnych wielkimi literami, aby odróżnić je od zwykłych zmiennych
  - tutaj pojawia się nawias klamrowy otwierający - sygnał, że deklaracja pól zaczyna się w tym momencie
  - nasza struktura ma trzy pola: pierwsze typu **string** i o nazwie **name**; druge typu **double** o nazwie **time**; trzecie to **int** o nazywie **recent_chapter**

Struktury - dlaczego ich potrzebujemy?

```
struct STUDENT {
   string  name;
   float   time;
   int     recent_chapter;
};
```

- Deklaracja zmiennej

```
struct STUDENT stdnt;
STUDENT stdnt2;
```

- **operator selekcji** przeznaczony dla struktur i jest oznaczany jako pojedynczy znak **.** (kropka).
  - stdnt.time = 1.5;

Struktury - dlaczego ich potrzebujemy?

```
struct STUDENT {
   string  name;
   float   time;
   int     recent_chapter;
};
```

STUDENT STDNTS[100000];

```
stndts[0].name = "Bond";
stndts[0].time = 3.5;
stdnts[0].recent_chapter = 4;
```

Struktury - dlaczego ich potrzebujemy?

- Możemy również użyć znacznika strukt zadeklarowania tablicy struktur:
  - DATE Visits[100];
  - Visits[0].year = 2012;
    Visits[0].month = 1;
    Visits[0].day = 1;

```
struct DATE {
    int year;
    int month;
    int day;
};
```

Struktury - dlaczego ich potrzebujemy?

- struct DATE {
  int year, month, day;
  } DateOfBirth, Visits[100];
- DATE current_date;

```
struct DATE {
   int year;
   int month;
   int day;
};
```

Struktury - dlaczego ich potrzebujemy?

- Struktura może być polem wewnątrz innej struktury.

```
struct STUDENT {
   string  name;
   float time;
   int   recent_chapter;
   struct DATE last_visit;
} HarryPotter;
```

- HarryPotter.last_visit.year = 2012;
  HarryPotter.last_visit.month = 12;
  HarryPotter.last_visit.day = 21;

Struktury - kilka ważnych zasad

- Nazwy pól struktury mogą nakładać się z nazwami znaczników i nie stanowi to problemu, chociaż może to powodować trudności w czytaniu i rozumieniu programu.

```
struct STRUCT {
    int STRUCT;
} Structure;

Structure.STRUCT = 0; /* STRUCT is a field name here */
```

Struktury - kilka ważnych zasad

- Może się zdarzyć, że danemu kompilatorowi, z którym pracujesz, nie podoba się, gdy nazwa znacznika struktury pokrywa się z nazwą zmiennej

```
struct STR {
 int field;
} Structure;
int STR;

Structure.field = 0;
STR = 1;
```

Struktury - kilka ważnych zasad

- Dwie struktury mogą zawierać pola o tych samych nazwach

```
struct {
 int f1;
} str1;

struct {
 char f1;
} str2;

str1.f1 = 32;
str2.f1 = str1.f1;
```

Inicjowanie struktur

- Możesz inicjować swoje struktury już **w momencie deklaracji**.
- Inicjator struktury jest ujęty w nawiasy klamrowe i **zawiera listę wartości przypisanych do kolejnych pól**, zaczynając od pierwszego.

```
struct DATE date = { 2012, 12, 21 };
```

  - date.year = 2012;
  - date.month = 12;
  - date.day = 21;

Inicjowanie struktur

```
struct STUDENT he = { "Bond", 3.5, 4, { 2012, 12, 21 }};
```

- he.name = "Bond";
- he.time = 3.5;
- he.recent_chapter = 4;
- he.last_visit.year = 2012
- he.last_visit.month = 12;
- he.last_visit.day = 21;

Inicjowanie struktur

```
STUDENT nobody = { };
```

- nobody.name = "";
- nobody.time = 0.0;
- nobody.recent_chapter = 0;
- nobody.last_visit.year = 0
- nobody.last_visit.month = 0;
- nobody.last_visit.day = 0;

Outline

1. Kontrola przepływu i typy danych cd.
2. **Rozszerzanie ekspresyjnej mocy: wskaźniki, funkcje i pamięć**

Wskaźniki - absolutne podstawy

- **Wskaźniki służą do przechowywania informacji o lokalizacji (adresie) wszelkich innych danych**.
- Postaraj się dostrzec tę istotną różnicę :
  - wartość zmiennej jest tym, co przechowuje zmienna;
  - adres zmiennej to informacja o miejscu, w którym ta zmienna jest umieszczona (gdzie się znajduje)

Wskaźniki - absolutne podstawy

- Ta deklaracja ustawia zmienną o nazwie **p**. To nie jest int - **gwiazdka oznacza, że p jest wskaźnikiem** i będzie używana do przechowywania informacji o lokalizacji danych typu **int**.

```
int *p;
```

Wskaźniki - absolutne podstawy

- Wskaźnik, któremu przypisano wartość zero, nazywany jest wskaźnikiem zerowym (**null pointer)**

```
p = 0;
```

- Symbol **NULL** jest faktycznie równy zeru. Wygląda jak zmienna, ale nie możesz zmienić jego wartości. To tak zwane **makro**.

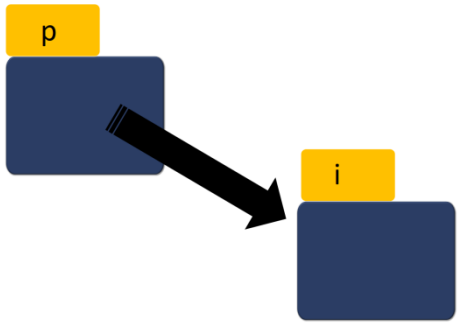
```
p = NULL;
```

Jak przypisać wartość?

- Możemy przypisać wskaźnik wartością, która **wskazuje na dowolną już istniejącą zmienną**.

```
p = &i;
```

- Po zakończeniu przypisania zmienna **p** wskaże miejsce, w którym zmienna **i** jest zapisana w pamięci



Jak uzyskać wartość?

- **Dereferencja jest operacją, w której zmienna wskaźnika** (jak zobaczymy później, to nie tylko zmienna, ale także wyrażenie, które daje wskaźnik) **staje się synonimem wartości, którą wskazuje**.

```
int ivar, *ptr;
```

Jak uzyskać wartość?

- Przypisujemy wartość 2 do zmiennej **ivar**

```
ivar = 2;
```

- Sprawiamy, że wskaźnik **ptr** wskazuje na zmienną **ivar**

```
ptr = &ivar;
```

- Następujące wywołanie powoduje wyświetlenie 2 na ekranie

```
cout << *ptr;
```

Jak ustawić wartość?

- Jak ustawić wartość wskazywaną przez wskaźnik?

$$*ptr = 4$$

Operator *sizeof*

- Operator podaje informacje o tym, ile bajtów pamięci zajmuje jego argument

sizeof

```
int i; char c;     char tab[10];

i = sizeof c;      i = sizeof tab;
```

Operator *sizeof*

- Jaki będzie wynik?

```
char tab[10];

i = sizeof tab[1];
```

```
int i;

i = sizeof i;
```

Operator *sizeof*

- Jaki będzie wynik?

```
char tab[10];

i = sizeof tab[1];
```

4 bajty

```
int i;

i = sizeof i;
```

4 bajty

Operator *sizeof*

```
#include <iostream>

using namespace std;

int main(void) {
   cout << "This computing environment uses:" << endl;
   cout << sizeof(char) << " bytes for chars" << endl;
   cout << sizeof(short int) << " bytes for shorts" << endl;
   cout << sizeof(int) << " bytes for ints" << endl;
   cout << sizeof(long int) << " bytes for longs" << endl;
   cout << sizeof(float) << " bytes for floats"  << endl;
   cout << sizeof(double) << " bytes for doubles"  << endl;
   cout << sizeof(bool) << " byte for bools" << endl;
   cout << sizeof(int *) << " bytes for pointers" << endl;
   return 0;
}
```

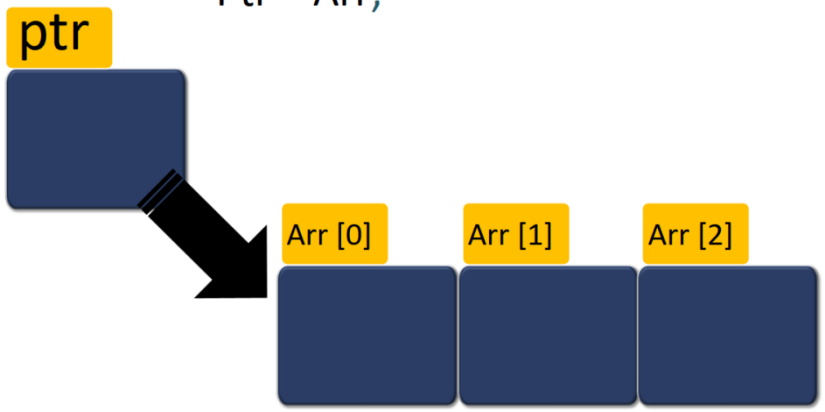
Wskaźniki vs. tablice

- Co mają wspólnego wskaźniki i tablice?

```
int *Ptr, Arr[3];
```

- Dwa przypisania, które następują po deklaracji, ustawią **Ptr** na tę samą wartość.

```
int *Ptr, Arr[3];
Ptr = &Arr[0];
Ptr = Arr;
```



Arytmetyka wskaźników

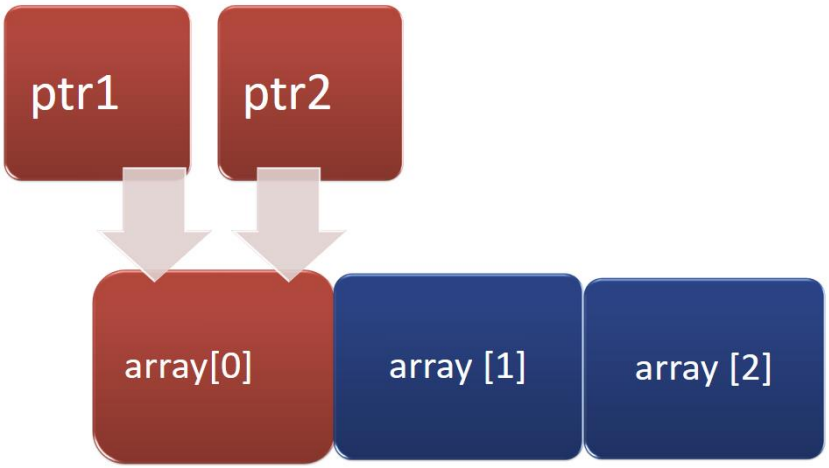
- Arytmetyka wskaźników jest znacząco różna od arytmetyki liczb całkowitych, ponieważ jest względnie zredukowana i pozwala tylko na następujące operacje :
  - **dodanie wartości całkowitej** do wskaźnika, który daje wskaźnik (*ptr + int → ptr*)
  - **odejmowanie wartości całkowitej** od wskaźnika dającego wskaźnik (*ptr – int → ptr*)
  - **odjęcie wskaźnika od wskaźnika**, który daje liczbę całkowitą (*ptr – ptr → int*)
  - **porównanie dwóch wskaźników** dla równości lub nierówności (takie porównanie daje wartość typu int reprezentującą prawdę lub fałsz) (*ptr == ptr → int*; *ptr* != *ptr → int*)

Arytmetyka wskaźników

```
int *ptr1, *ptr2, array[3], i;

ptr1 = array;
```

- ptr2 wskazuje na pierwszy element tablicy

```
ptr2 = ptr1;
```



Arytmetyka wskaźników

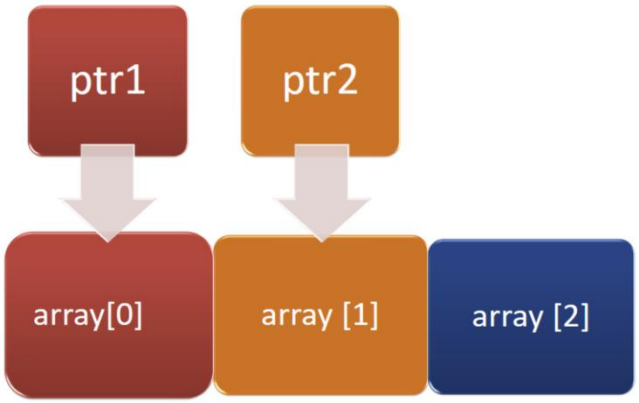
- Możemy sprawdzić, czy te dwa wskaźniki są równe

```
if(ptr2 == ptr1) {
:
:
}
```

Arytmetyka wskaźników

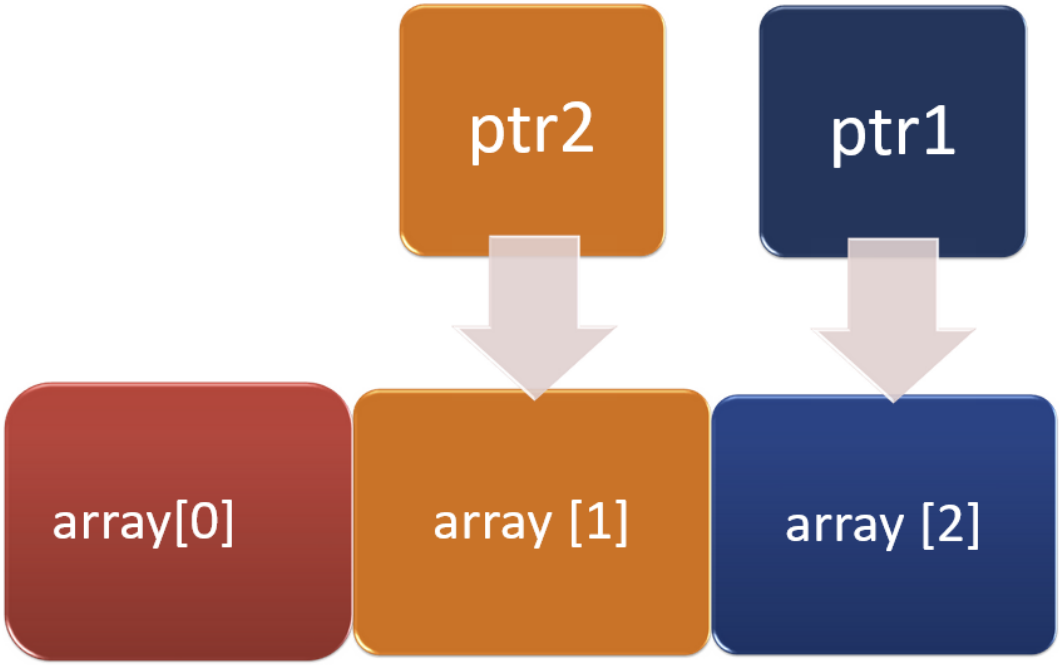
- Określa, **ile bajtów pamięci zajmuje dany typ** (w tym celu używamy operatora **sizeof**) - w naszym przypadku będzie to **sizeof (int)**
- wartość, którą chcemy dodać do wskaźnika, jest mnożona przez podany rozmiar

```
ptr2 = ptr2 + 1;
ptr2++;
```



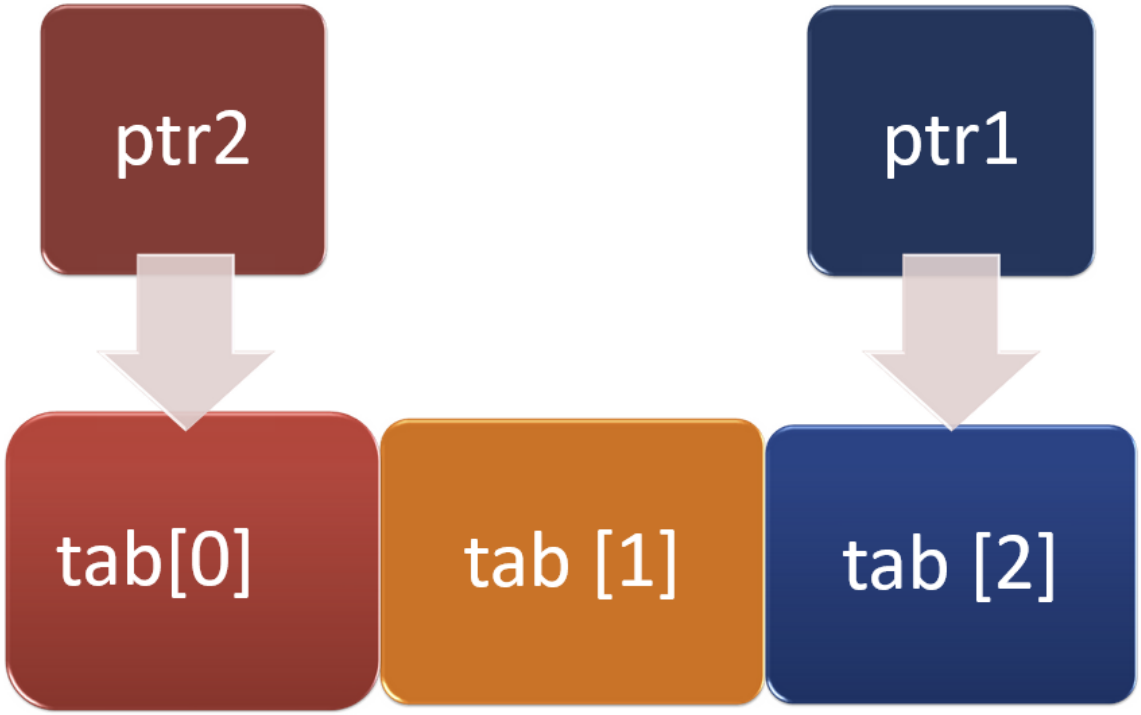
Arytmetyka wskaźników

```
ptr1 = ptr1 + 2;
```



Arytmetyka wskaźników

```
ptr2 = ptr2 – 1;
```



Co to jest funkcja?

- Funkcja jest rodzajem pudełka (nie zawsze czarnego), które może zrobić coś pożytecznego
- Ogólnie możemy podzielić funkcje na dwie grupy:
  - funkcje napisane przez kogoś innego (nie ty), które są udostępniane przez środowisko, czasami nazywane funkcjami **predefiniowanymi** lub **bibliotecznymi**
  - funkcje napisane przez ciebie

Co to jest funkcja?

- Każda funkcja charakteryzuje się następującymi cechami:
  - nazwa
  - parametry
  - typ zwracanego wyniku

```
float square(float x);
```

Co to jest funkcja?

- Przekształcenie deklaracji w definicję wymaga od nas dodania ciała

```
float square(float x)
{
    float result;

    result = x * x;
    return result;
}
```

Pierwsza funkcja

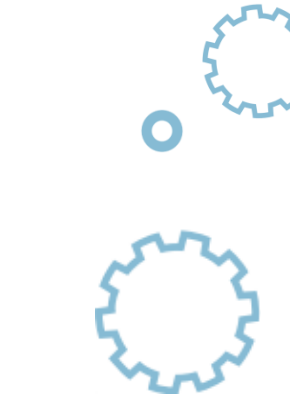
```
#include  <iostream>
using namespace std;
float square(float x)
{
	float result;
	result = x * x;
	return result;
}
int main(void) {
	float arg = 2.0;
	cout << "The second power of " << arg << " is " << square(arg) << endl;
	return 0;
}
```

Pierwsza funkcja

```
#include  <iostream>
using namespace std;
float square(float);
int main(void) {
	float arg = 2.0;
	cout << "The second power of " << arg << " is " << square(arg) << endl;
	return 0;
}

float square(float x)
{
	float result;
	result = x * x;
	return result;
}
```

Definiowanie funkcji

```
return_type function_name (parameters_list)
{
        function_body;
}
```



Definiowanie funkcji

```
#include <iostream>

using namespace std;

void Greet(void)
{
        cout << "Ave user!" << endl;
}

void GreetManyTimes(int howmanytimes)
{
    while(howmanytimes > 0)
    {
        Greet();
        howmanytimes--;
    }
}

int main(void)
{
        int sizeofego;

        cout << "How big is your ego? [km]" << endl;
        cin >> sizeofego;
        GreetManyTimes(1 + sizeofego);
        return 0;
}
```

Definiowanie funkcji

```
#include <iostream>

using namespace std;

float FahrenheitToCelsius(float temp)
{
        return ((temp - 32.0) * 5.0) / 9.0;
}

void TestTheFunction(float temp) {
        cout << "Fahrenheit " << temp << " corresponds to " <<
FahrenheitToCelsius(temp) << " Centigrade" << endl;
}

int main(void)
{
        TestTheFunction(32.0);
        TestTheFunction(212.0);
        TestTheFunction(451.0);
        return 0;
}
```

Przykłady funkcji

- Oczekujemy, że program wygeneruje następujący wynik :
  - 32 stopnie Fahrenheita odpowiadają 0 stopniom Celsjusza
  - 212 stopni Fahrenheita odpowiada 100 stopniom Celsjusza
  - 451 stopni Fahrenheita odpowiada 232.778 stopniom Celsjusza

The invocation syntax

- A function may:
  - **return a value** when it has a type name in front of its name or it doesn’t have the type name there (in this case the function is considered as returning an *int* value); such a function has a result and may have an effect, too
  - **return nothing** when the *void* keyword is in front of its name; such a function doesn't have a result and we can expect that it has an effect

Składnia wywołania

```
void VoidFunction(int par) { ... ; return; }

int NonVoidFunction(int par) { ... ; return par * par; }
```

- Jedynie akceptowalna forma wywołania funkcji **VoidFunction** wygląda następująco :
  - VoidFunction(2);
- **NonVoidFunction** można wywołać na dwa następujące sposoby :
  - value = NonVoidFunction(2);
  - NonVoidFunction(2);

Skutki uboczne

- Każda funkcja musi mieć zdolność komunikowania się ze swoim środowiskiem.
- Znamy dwa rodzaje komunikacji takie jak:
  - **przenoszenie danych do funkcji za pomocą rzeczywistych parametrów**, których wartości są przypisane do parametrów formalnych
  - **przesyłanie danych z funkcji za pomocą wyniku funkcji**; pamiętaj, że tylko jedna wartość może zostać przekazana w ten sposób, ponieważ składnia instrukcji **return** pozwala określić tylko jedną wartość.

Używanie zmiennej globalnej

```
#include <iostream>

using namespace std;

int globvar = 0;

void func(void)
{
	cout << "Thank you for invoking me :)" << endl;
	globvar++;
}

int main(void)
{
	for(int i = 0; i < 5; i++)
		func();
	cout << endl << "The function enjoyed " << globvar <<
		" times" << endl;
	return 0;
}
```

Przekazywanie parametrów przez wartość

```
#include <iostream>

using namespace std;

void AmIAbleToChangeMyParameter(int param)
{
	cout << "----------" << endl;
	cout << "I have got: " << param << endl;
	param++;
	cout << "I'm about to give back: " << param << endl;
	cout << "----------" << endl;
}

int main(void)
{
	int var = 1;

	cout << "var = " << var << endl;
	AmIAbleToChangeMyParameter(var);
	cout << "var = " << var << endl;
	return 0;
}
```

Przekazywanie parametrów przez referencję

```
#include <iostream>

using namespace std;

void AmIAbleToChangeMyParameter(int &param)
{
	cout << "----------" << endl;
	cout << "I have got: " << param << endl;
	param++;
	cout << "I'm about to give back: " << param << endl;
	cout << "----------" << endl;
}

int main(void)
{
	int var = 1;

	cout << "var = " << var << endl;
	AmIAbleToChangeMyParameter(var);
	cout << "var = " << var << endl;
	return 0;
}
```

Przekazywanie parametrów przez referencję

- Możesz mieszać parametry obu rodzajów, jeśli uznasz to za przydatne.

```
#include <iostream>

using namespace std;

void MixedStyles(int bval, int &bref)
{
	bref = bval + 1;
}

int main(void)
{
	int var1 = 1, var2;

	MixedStyles(var1, var2);
	cout << "var1 = " << var1 << ", var2 = " << var2 << endl;
	return 0;
}
```

Przekazywanie parametrów przez referencję

- Metoda "przekazywania przez referencję" ma jedno ważne i oczywiste **ograniczenie**.
- **Jeśli parametr jest zadeklarowany jako przekazany przez referencję (dlatego poprzedza go znak &), jego odpowiedni faktyczny parametr musi być zmienną**.
- Rzeczywisty parametr odnoszący się do formalnego parametru "przekazywanego przez wartość" może być ogólnie wyrażeniem, więc możemy użyć nie tylko wartości zmiennej, ale także wyniku wywołania funkcji.

Przekazywanie parametrów przez referencję

```
void ByRef(int &par)
{
        par = 0;
}

void ByVal(int par)
{
        par = 0;
}
```

- Dopuszczalne są wszystkie następujące wywołania:
  - ByVal(i);
  - ByVal(i + 2);
  - ByVal(intfun(0));

Przekazywanie parametrów przez wartość

- Możliwe jest wykorzystanie "przekazywania przez wartość" i przekazanie wartości poza funkcję

```
#include <iostream>

using namespace std;

void ByPtr(int *ptr)
{
	*ptr = *ptr + 1;
}

int main(void)
{
	int  variable = 1;
	int *pointer  = &variable;

	ByPtr(pointer);
	cout << "variable = " << variable << endl;
	return 0;
}
```

Parametry

- Zamierzamy teraz przepisać funkcję **Greet**, aby była bardziej elastyczna. Chcemy tego by:
  - być w stanie generować dowolne powitanie, nie tylko te predefiniowane w kodzie źródłowym,
  - być w stanie generować powitanie więcej niż jeden raz, na żądanie inwokatora.
- Oznacza to, że nasz **NewGreet** musi mieć dwa parametry :
  - powitanie
  - liczba powtórzeń powitania

Parametry

```
#include <iostream>
using namespace std;
void NewGreet(string greet, int repeats)
{
	for(int i = 0; i < repeats; i++)
		cout << greet << endl;
}
int main(void)
{
	NewGreet("Hi!", 5);
	return 0;
}
```

Domyślne parametry - prosty przykład

```
#include <iostream>
using namespace std;
void NewGreet(string greet, int repeats = 1)
{
        for(int i = 0; i < repeats; i++)
            cout << greet << endl;
}

int main(void)
{
    NewGreet("Hello", 2);
    NewGreet("Good morning");
    NewGreet("Hi", 1);
    return 0;
}
```

Domyślne parametry - prosty przykład

- Program wygeneruje następujące wyniki :

```
#include <iostream>
using namespace std;
void NewGreet(string greet, int repeats = 1)
{
        for(int i = 0; i < repeats; i++)
            cout << greet << endl;
}

int main(void)
{
    NewGreet("Hello", 2);
    NewGreet("Good morning");
    NewGreet("Hi", 1);
    return 0;
}
```

Hello
Hello
Good morning
Hi

Domyślne parametry - prosty przykład

- Czy możliwe jest posiadanie więcej niż jednego parametru domyślnego w jednej funkcji?

```
#include <iostream>
using namespace std;
void NewGreet(string greet = "Good morning", int repeats = 1)
{
        for(int i = 0; i < repeats; i++)
            cout << greet << endl;
}
int main(void)
{
        NewGreet("Hello", 2);
        NewGreet("Hi");
        NewGreet();
        return 0;
}
```

Różne narzędzia do różnych zadań

- Funkcja znajdowania większej z dwóch liczb zmiennoprzecinkowych

```
float max(float a, float b)
{
        if(a > b)
            return a;
        else
            return b;
}
```

Max – wersja rozszerzona

```
float max(float a, float b, float c)
{
        int m = a;
        if(b > m)
            m = b;
        if(c > m)
            m = c;
        return m;
}
```

- Poprzednia funkcja
  - x = max(max(a,b),c);

Jak znaleźć najlepszego kandydata?

```
void PlayWithNumber(int x) { ... }
void PlayWithNumber(float x) { ... }
:
PlayWithNumber(1);
:
```

- Która z tych dwóch przeciążonych funkcji jest najlepszym kandydatem do wywołania?

Jak znaleźć najlepszego kandydata?

```
void PlayWithNumber(int x) { ... }
void PlayWithNumber(float x) { ... }
:
PlayWithNumber(1.0);
:
```

- Która z tych dwóch przeciążonych funkcji jest najlepszym kandydatem do wywołania?
  - There is no good candidate
  - PlayWithNumber(1.0f);

Nowy operator: trzyargumentowy

- Ten operator działa w następujący sposób :
  - oblicza wartość ***expression*1**
  - jeśli obliczona wartość jest różna od zera, operator zwraca wartość ***expression*2**, całkowicie pomijając ***expression*3**
  - jeśli wartość obliczona w kroku 1 wynosi zero, operator zwraca wartość ***expression*3**, pomijając ***expression*2**.

```
expression1 ? expression2 : expression3
```

Nowy operator: trzyargumentowy

- i = i > 0 ? 1 : 0;

- **if**(i > 0)
- i = 1;
- **else**
- i = 0

```
float max(float a, float b)
{
        return a > b ? a : b;
}
```

Sortowanie tablicy

Sortowanie tablicy

```
int numbers[5];    // array to be sorted
int aux;           // auxiliary variable for swaps

// we need 5 – 1 comparisons – why?
for(int i = 0; i < 4; i++) {
   // compare adjacent elements
   if( numbers[i] > numbers[i + 1]) {
      /* if we went here it means that we have to swap the elements */
      aux = numbers[i];
      numbers[i] = numbers[i + 1];
      numbers[i + 1] = aux;
   }
```

Sortowanie tablicy

```
int  numbers[5];
int  aux;
bool swapped;

do {  // we will decide if we need to continue this loop
   swapped = false;  // no swap occured yet

   for(int i = 0; i < 4; i++)
      if(numbers[i] > numbers[i + 1]) {
         swapped = true;
         aux = numbers[i];
         numbers[i] = numbers[i + 1];
         numbers[i + 1] = aux;
      }
} while(swapped);
```

Wersja końcowa

```
#include <iostream>

using namespace std;

int main(void) {
	int  numbers[5];
	int  aux;
	bool swapped;
	// ask the user to enter 5 values
	for(int i = 0; i < 5; i++) {
		cout << endl << "Enter value #" << i + 1 << ": ";
		cin >> numbers[i];
	}
	// sort them
	do {
		swapped = false;
		for(int i = 0; i < 4; i++) {
			if(numbers[i] > numbers[i + 1]) {
				swapped = true;
				aux = numbers[i];
				numbers[i] = numbers[i + 1];
				numbers[i + 1] = aux;
			}
		}
	} while(swapped);
	// print results
	cout << endl << "Sorted array: " << endl;
	for(int i = 0; i < 5; i++)
		cout << numbers[i] << " ";
	cout << endl;
	return 0;
}
```

Pamięć na żądanie

```
float *array = new float[20];
int    count = new int;

delete [] array;
delete count;
```

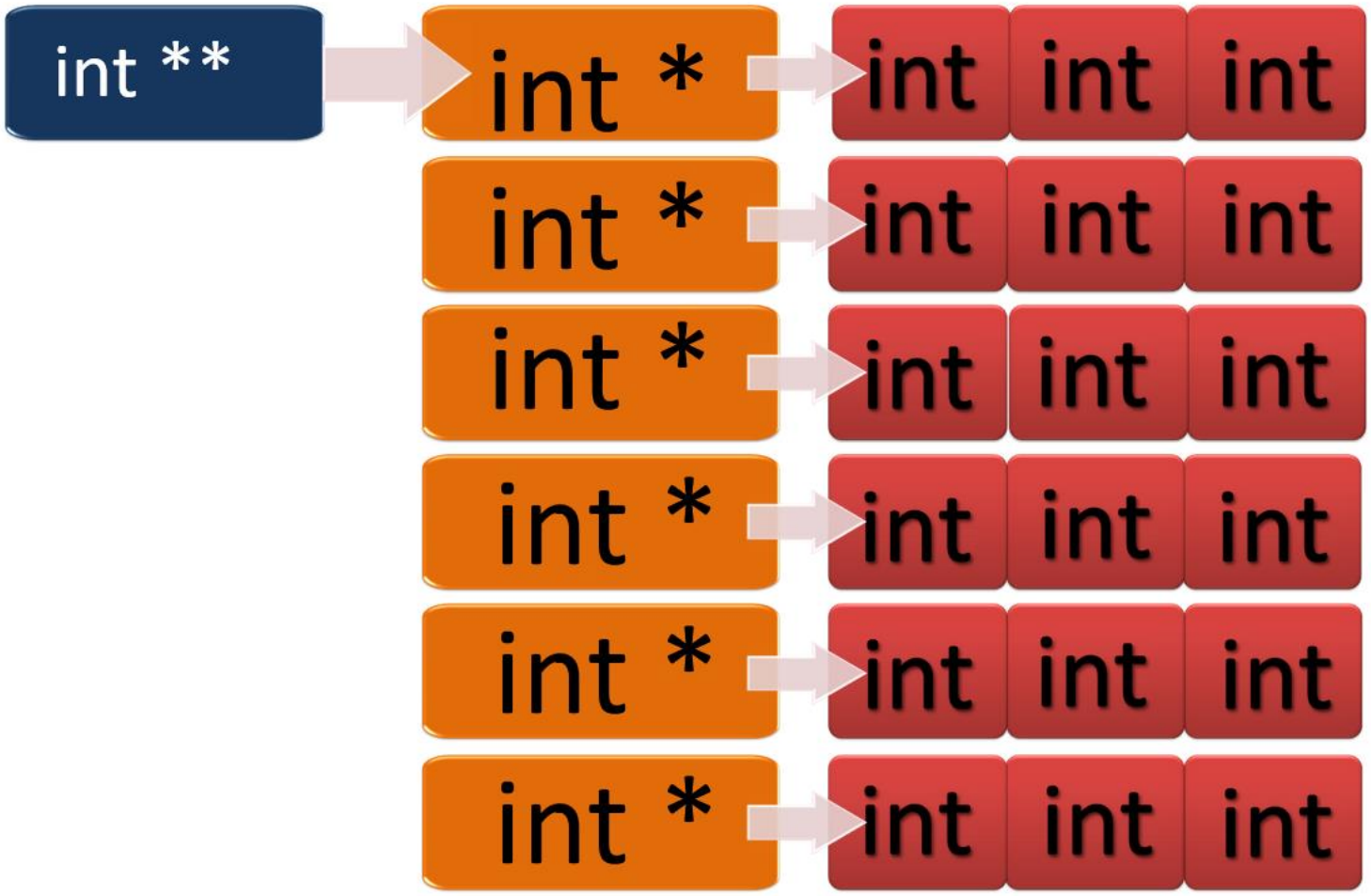
Pamięć na żądanie

- ***Tablice dynamiczne***

```
#include <iostream>

using namespace std;

int main(void) {
   float *arr;

   arr  = new float[5];
   for(int i = 0; i < 5; i++)
      arr[i] = i * i;
   for(int i = 0; i < 5; i++)
      cout << arr[i] << endl;
   delete [] arr;
   return 0;
}
```

```
#include <iostream>
using namespace std;
int main(void) {
   int *numbers, how_many_numbers;
   int  aux;
   bool swapped;

   cout << "How many numbers are you going to sort? ";
   cin >> how_many_numbers;
   if( how_many_numbers <= 0 || how_many_numbers > 1000000) {
      cout << "Are you kidding?" << endl;
      return 1;
   }
   numbers = new int[how_many_numbers];
   for(int i = 0; i < how_many_numbers; i++) {
      cout << "\nEnter the number #" << i + 1 << ": ";
      cin >> numbers[i];
   }
  do {
     swapped = false;
     for(int i = 0; i < how_many_numbers - 1; i++)
        if(numbers[i] > numbers[i + 1]) {
           swapped = true;
           aux = numbers[i];
           numbers[i] = numbers[i + 1];
           numbers[i + 1] = aux;
        }
  } while(swapped);
  cout << endl << "The sorted array:" << endl;
  for(int i = 0; i < how_many_numbers; i++)
     cout << numbers[i] << " ";
  cout << endl;
  delete [] numbers;
  return 0;
}
```

Tablice wskaźników



Tablice wskaźników

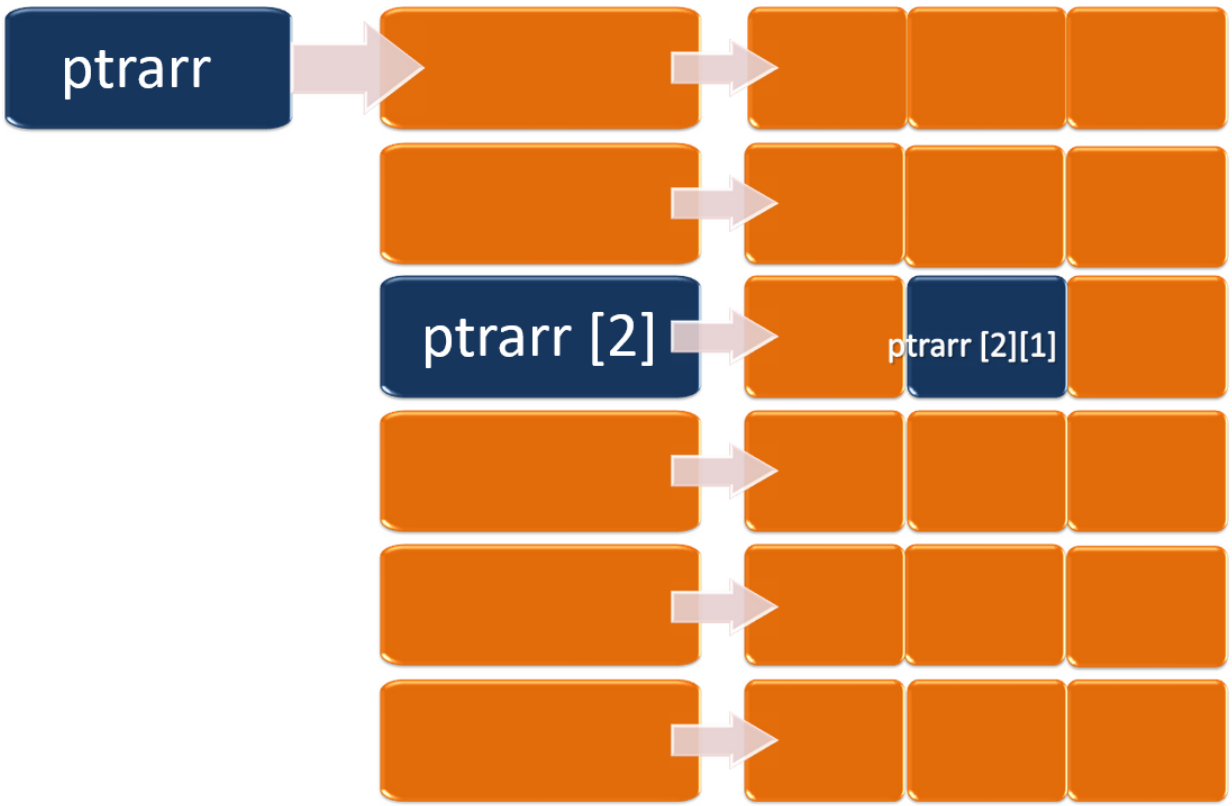
```
int **ptrarr;

ptrarr = new int * [rows];

for(int r = 0; r <rows; r++)
    ptrarr[r] = new int[columns];
```

Tablice wskaźników

```
ptrarr[r][c] = 0;
```

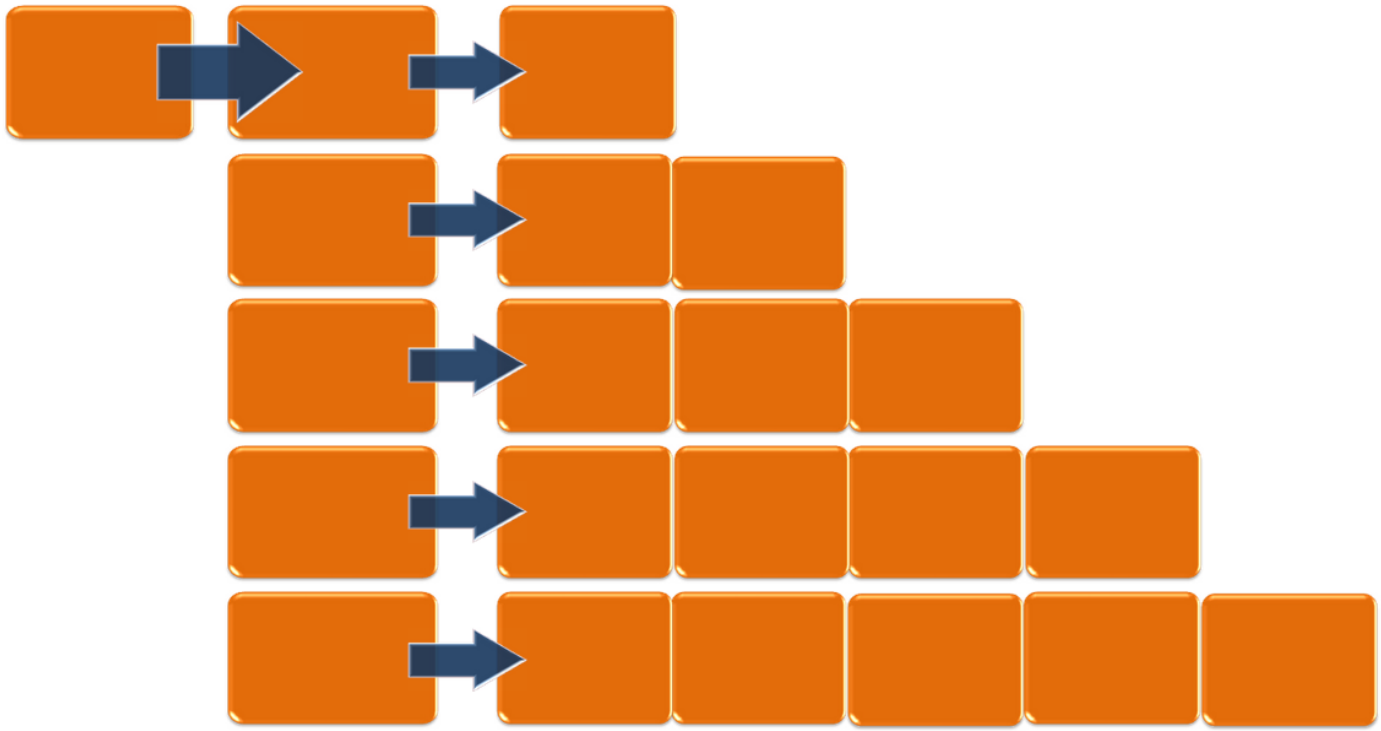
Tablice wskaźników

- ptrarr[2][1]



Tablice wskaźników

- każdy wiersz może być innej długości



Tablice wskaźników

```
#include <iostream>

using namespace std;

int main(void)
{
	int rows = 5, cols = 5;
	int **arr;
	// allocate and initialize the array
	arr = new int * [rows];
	for (int r = 0; r < rows; r++) {
		arr[r] = new int[r + 1];
		for(int c = 0; c <= r; c++)
			arr[r][c] = (r + 1) * 10 + c + 1;
	}
	// print the array
	for(int r = 0; r < rows; r++) {
		for(int c = 0; c <= r; c++)
			cout << arr[r][c] << " ";
		cout << endl;
	}
	// free the array
	for(int r = 0; r < rows; r++)
		delete [] arr[r];
	delete [] arr;
	return 0;
}
```