Maciej Sobieraj

Lecture 2

Outline

1. **Advanced flow control and data aggregates**
2. Extending expressive power: pointers, functions and memory

Arrays

- Why?
  - It may be that we have to read, store, process, and finally, print dozens, maybe hundreds, perhaps thousands of numbers.

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

- store **five** values of type *int*
  - the elements in an array are numbered **starting from 0**

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

Arrays

- Assigning a value to a chosen element of an array

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

- A value stored in the **third** element of the array

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

Arrays

- The **sum of all values** stored in the *numbers* array

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

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

- Assigning the same value (e.g. 2012) to all elements of the array

```
int numbers[5];

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

Arrays

- What the code below does?

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

Array initialization

- The vector initiator is simply a list of values enclosed inside **curly brackets**.

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

- We didn't specify the size of the array but **provided an initiator**.

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

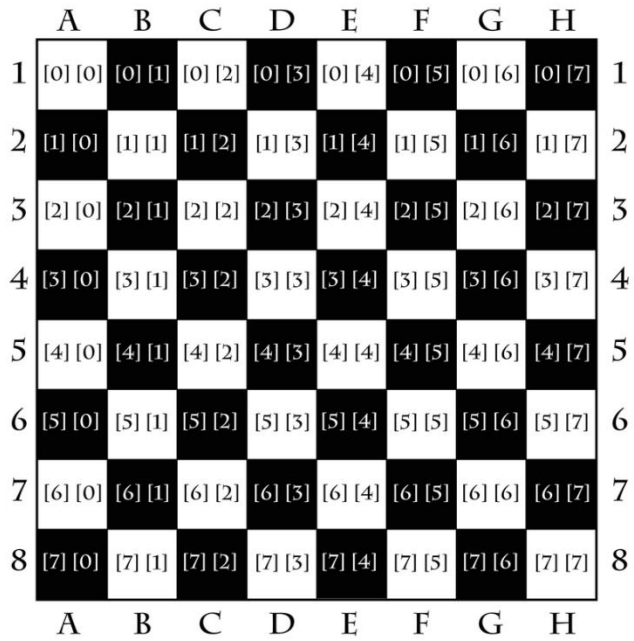
Not only ints

```
float FloatArr[10];

char surname[20];

bool votes[100];
```

Not only vectors

- **two dimensional array**

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



Not only vectors

- The device records the air temperature on an hourly basis and does it throughout the month.
- This gives us a total of 24 * 31 = 744 values.
  - **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;

Not only vectors

```
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;
```

Not only vectors

```
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.";
```

Not only vectors

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

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

Not only vectors

- The “C++” language **does not limit the number of the array's dimensions**.

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

Structures – why do we need them?

- A string is little more than a type
- Variables of type string may be assigned with the same operators as any other variable

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

- For example, suppose that the first registered student is Mr. Bond (James Bond).
  - student_name[0] = "Bond";

Structures – why do we need them?

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

- Mr. Bond has spent three hours and thirty minutes studying our course.
  - student_time[0] = 3.5;

Structures – why do we need them?

- The main issue here is that the data concerning the same object (a student) is **dispersed between three variables**, although it should logically exist as a consolidated unit.
- Can we use **an aggregate whose elements could be of different types**?
  - **A structure can contain any number of any elements of any type**.

Structures – why do we need them?

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

- The declaration of the structure →
  - the declaration of the structure always starts with the keyword **struct**
  - there is a so-called struct tag after the keyword (*STUDENT* in this case); it's the name of the structure itself; there is a widely accepted custom of composing structure tags with capital letters simply to distinguish them from ordinary variables
  - here comes the opening curly bracket - a signal that the declaration of fields begins at this point
  - our structure has three fields: the first is a *string* and is called *name*; the second is a *float* and is called *time*; the third is an *int* and it's called *recent_chapter*

Structures – why do we need them?

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

- The declaration of variable

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

- **selection operator** designed for structures and is denoted as a single character **.** (dot).
  - stdnt.time = 1.5;

Structures – why do we need them?

```
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;
```

Structures – why do we need them?

- We can also use the structure tag to declare an array of structures:
  - DATE Visits[100];
  - Visits[0].year = 2012;
    Visits[0].month = 1;
    Visits[0].day = 1;

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

Structures – why do we need them?

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

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

Structures – why do we need them?

- **A structure can be a field inside another structure**.

```
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;

Structures – a few important rules

- A structure's field names may overlap with the tag names and that's not a problem, although it may cause you some difficulty in reading and understanding the program.

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

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

Structures – a few important rules

- It may be the case that the particular compiler you're working with doesn't like it when a structure's tag name overlaps with the variable's name

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

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

Structures – a few important rules

- **Two structures can contain fields with the same names**

```
struct {
 int f1;
} str1;

struct {
 char f1;
} str2;

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

Initializing structures

- You can initialize your structures as early as **at the time of declaration**.
- The structure's initiator is enclosed in curly brackets and contains **a list of values assigned to the subsequent fields**, starting from the first.

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

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

Initializing structures

```
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;

Initializing structures

```
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. Advanced flow control and data aggregates
2. **Extending expressive power: pointers, functions and memory**

Pointers – the absolute basics

- **Pointers are used to store information about the location (address) of any other data**.
- Try to get this important difference:
  - the value of the variable is what the variable stores;
  - the address of the variable is information about where this variable is placed (where it lives)

Pointers – the absolute basics

- This declaration sets up a variable named *p*. It isn't an *int* - **the asterisk means that *p* is a pointer** and will be used to store information about the location of the data of type *int*.

```
int *p;
```

Pointers – the absolute basics

- A pointer that is assigned a value of zero is called **a null pointer**

```
p = 0;
```

- The **NULL** symbol is actually equal to zero. It looks like a variable but you can't modify its value. It's a so-called **macro**.

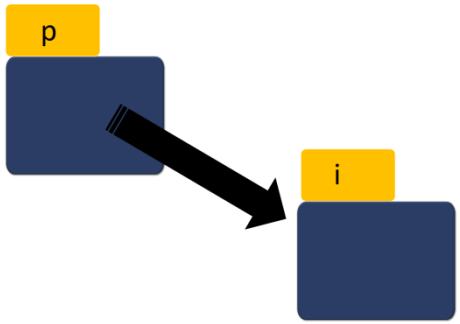
```
p = NULL;
```

How to assign a value?

- We can assign the pointer with the value which **points to any already existing variable**.

```
p = &i;
```

- After completing the assignment, the *p* variable will point to the place where the *i* variable is stored in the memory



How to get a value

- **Dereferencing is an operation where the pointer variable** (as we'll see later, it's not only a variable, but also an expression that yields a pointer) **becomes synonymous with the value it points to**.

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

How to get a value

- We assign the value of 2 to the *ivar* variable

```
ivar = 2;
```

- We make the *ptr* pointer point to the *ivar* variable

```
ptr = &ivar;
```

- The following invocation brings up 2 to the screen

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

How to set a value

- *How do we set a value pointed to by the pointer?*

```
*ptr = 4
```

sizeof operator

- The operator provides information on **how many bytes of memory its argument occupies**

sizeof

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

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

sizeof operator

- What will the result be?

```
char tab[10];

i = sizeof tab[1];
```

```
int i;

i = sizeof i;
```

sizeof operator

```
#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;
}
```

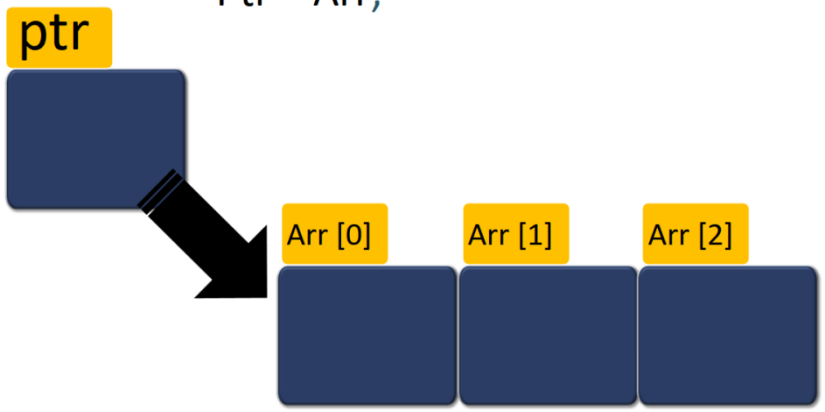
Pointers vs. arrays

- What do pointers and arrays have in common?

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

- The two assignments that follow the declaration will set *Ptr* to the same value.

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



The pointers' arithmetic

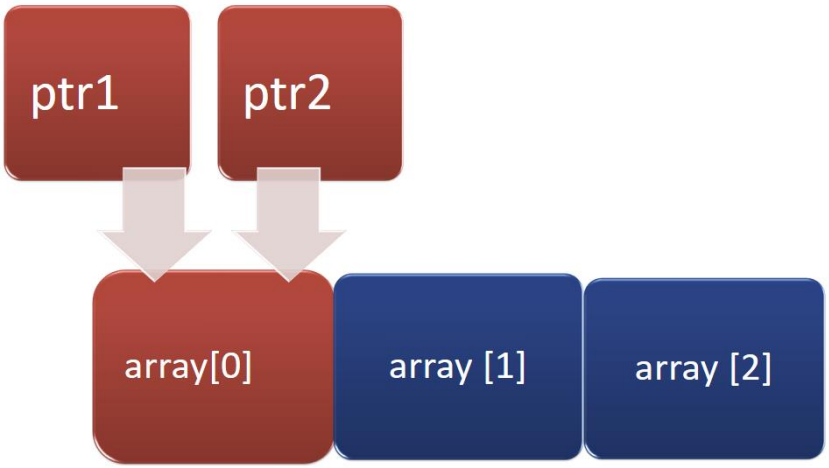
- The pointers' arithmetic is significantly different from the integers' arithmetic as it is relatively reduced and allows the following operations only:
  - **adding an integer** value to a pointer giving a pointer (*ptr* + *int* → *ptr*)
  - **subtracting an integer** value from a pointer giving a pointer (*ptr* – *int* → *ptr*)
  - **subtracting a pointer from a pointer** giving an integer (*ptr* – *ptr* → *int*)
  - **comparing the two pointers** for equality or inequality (such a comparison gives a value of type *int* representing true or false) (*ptr* == *ptr* → *int*; *ptr* != *ptr* → *int*)

The pointers' arithmetic

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

ptr1 = array;
```

- ***ptr2* points to the first element of the *array***

```
ptr2 = ptr1;
```



The pointers' arithmetic

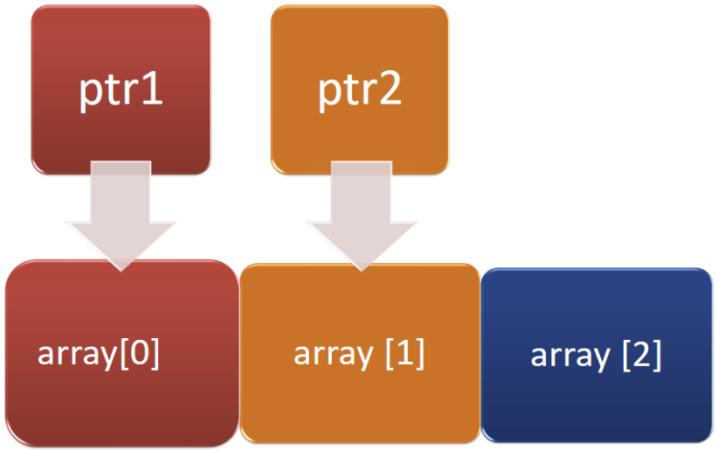
- We can check if the two pointers are equal

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

The pointers' arithmetic

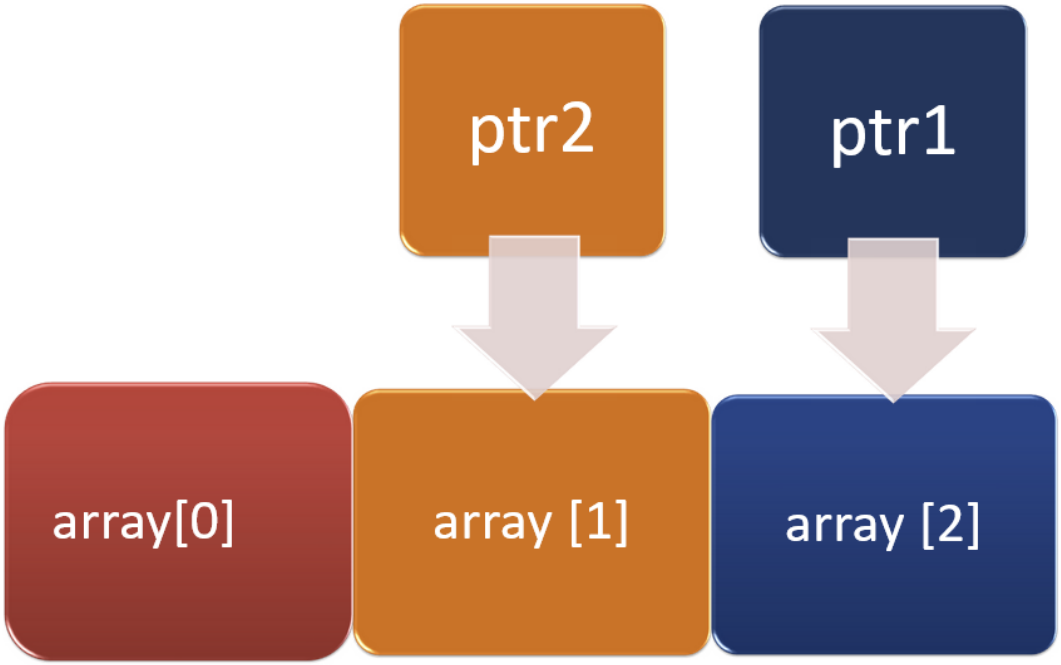
- It’s determined **how many bytes of memory the type occupies** (we use the *sizeof* operator for this purpose) - in our case it will be *sizeof* (*int*)
- the value we want to add to the pointer is multiplied by the given size

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



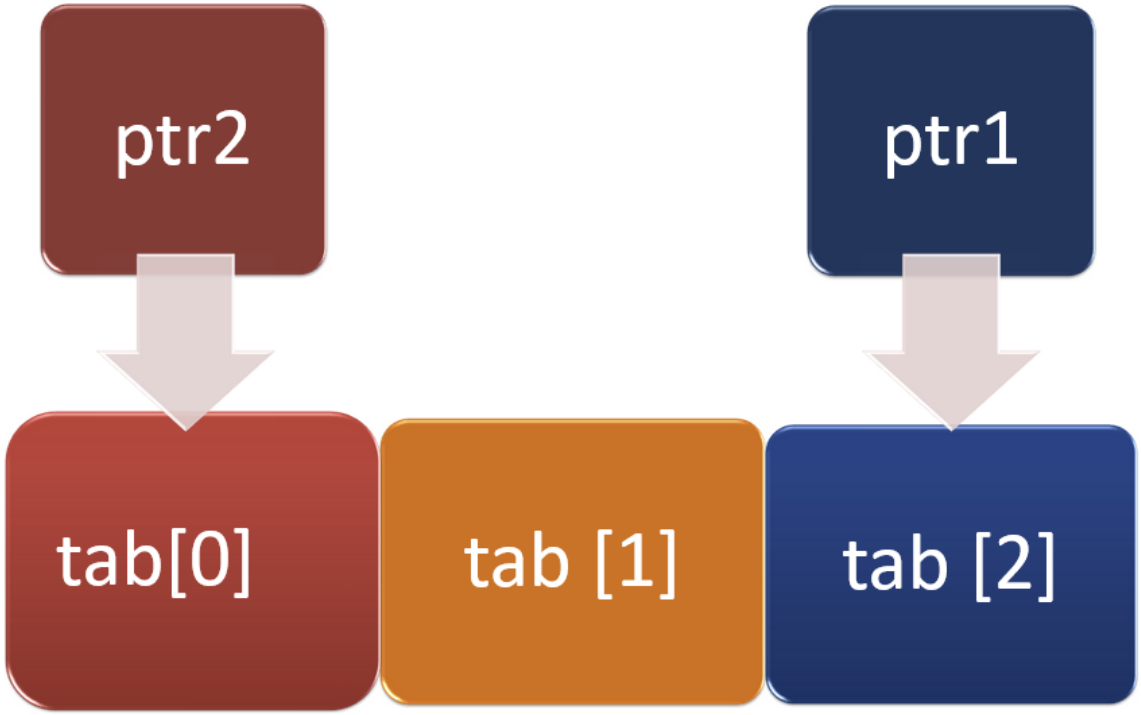
The pointers' arithmetic

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



The pointers' arithmetic

```
ptr2 = ptr2 – 1;
```



What is a function?

- A function is a kind of box (not always black) that can do something useful
- In general, we can divide functions into two groups:
  - functions written by someone else (not you) which are made available by the environment, sometimes called **predefined** or **library functions**
  - functions written by you

What is a function?

- Each function is characterized by the following traits:
  - name
  - parameters
  - type of result

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

What is a function?

- Transforming a declaration into a definition requires us to add a body

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

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

First function

```
#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;
}
```

First function

```
#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;
}
```

Defining functions

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

Defining functions

```
#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;
}
```

Example functions

```
#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;
}
```

Example functions

- We expect the program to produce the following output:
  - Fahrenheit 32 corresponds to 0 Centigrade
  - Fahrenheit 212 corresponds to 100 Centigrade
  - Fahrenheit 451 corresponds to 232.778 Centigrade

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

The invocation syntax

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

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

- The only acceptable form of the *VoidFunction* invocation looks like this:
  - VoidFunction(2);
- the *NonVoidFunction* can be invoked in the following two ways:
  - value = NonVoidFunction(2);
  - NonVoidFunction(2);

Side effects

- Any function needs to have the ability to communicate with its environment.
- We already know two kinds of communication like this:
  - **transferring data to a function using actual parameters** whose values are assigned to formal parameters
  - **transferring data from a function using the function's result**; note that only one value may be transferred by such means because the syntax of the *return* statement allows you to specify only one value

Using a global variable

```
#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;
}
```

Passing parameters by value

```
#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;
}
```

Passing parameters by reference

```
#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;
}
```

Passing parameters by reference

- You can mix parameters of both kinds if you find it useful.

```
#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;
}
```

Passing parameters by reference

- The “passing by reference” method has one important and obvious **limitation**.
- **If a parameter is declared as passed by reference (so it is preceded by the & sign) its corresponding actual parameter must be a variable**.
- An actual parameter referring to a “passed by value” formal parameter may be an expression in general, so we can use not only a variable, also a literal, or even a function invocation's result.

Passing parameters by reference

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

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

- All the following invocations are permitted:
  - ByVal(i);
  - ByVal(i + 2);
  - ByVal(intfun(0));

Passing parameters by value

- It is possible to utilize “passing by value” and be able to propagate the value outside the function

```
#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;
}
```

Parameters – cont.

- We’re now going to rewrite our *Greet* function to make it more flexible. We want it to:
  - be able to emit **any greeting**, not only the one predefined in the source code,
  - be able to emit the greeting **more than once**, on the invoker's demand.
- This means that our *NewGreet* has to have two parameters intended to:
  - store the greeting
  - store the number of greeting repetitions

Parameters – cont.

```
#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;
}
```

Default parameters – a simple example

```
#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;
}
```

Default parameters – a simple example

- The program will produce the following output:

```
Hello
Hello
Good morning
Hi
```

```
#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;
}
```

Default parameters – a simple example

- Is it possible to have more than one default parameter in one function?

```
#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;
}
```

Different tools for different tasks

- A function to find the larger of two float numbers

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

Max – extended version

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

- Previous function
  - x = max(max(a,b),c);

How to find the best candidate?

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

- Which of these two overloaded functions is the best candidate for the invocation?

How to find the best candidate?

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

- Which of these two overloaded functions is the best candidate for the invocation?
  - There is no good candidate
  - PlayWithNumber(1.0f);

A new operator: a three-argument one

- This operator works as follows:
  - **calculates** the value of the *expression1*
  - if the calculated value is **non-zero**, the operator returns the value of *expression2*, completely neglecting *expression3*
  - if the value calculated in step 1 is **zero**, the operator returns the value of *expression3*, omitting *expression2*.

```
expression1 ? expression2 : expression3
```

A new operator: a three-argument one

- i = i > 0 ? 1 : 0;

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

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

Sorting an array

Sorting an array

```
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;
   }
```

Sorting an array

```
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);
```

Final version

```
#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;
}
```

Memory on demand

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

delete [] array;
delete count;
```

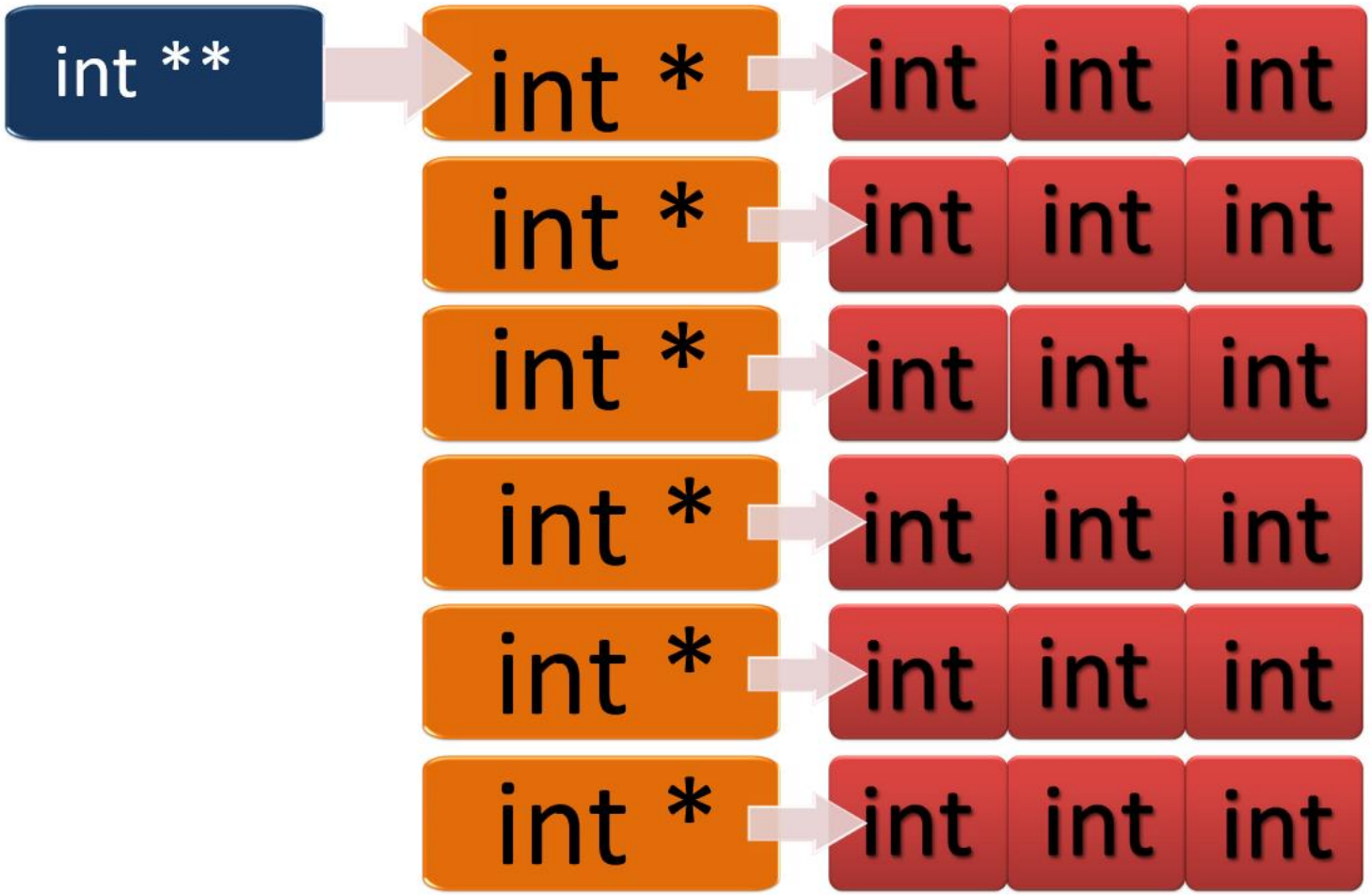
Memory on demand

- ***dynamic arrays***

```
#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;
}
```

Arrays of pointers



Arrays of pointers

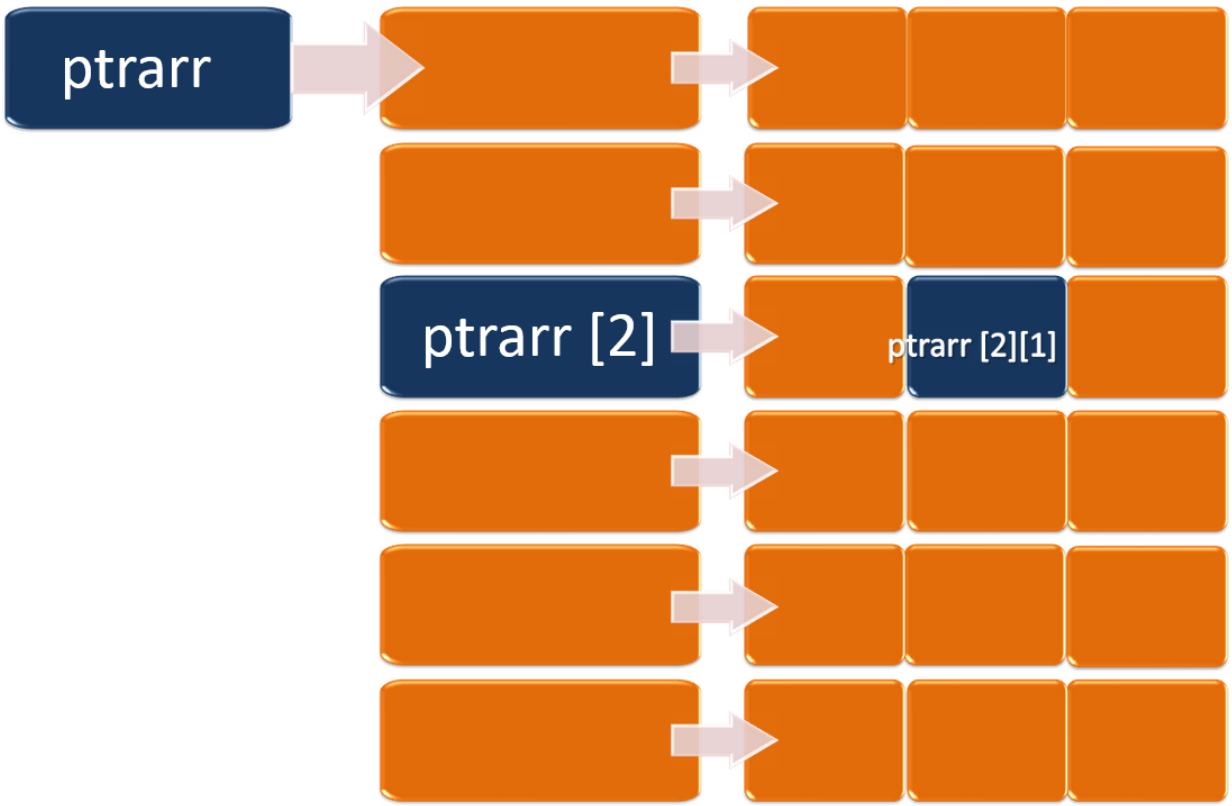
```
int **ptrarr;

ptrarr = new int * [rows];

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

Arrays of pointers

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

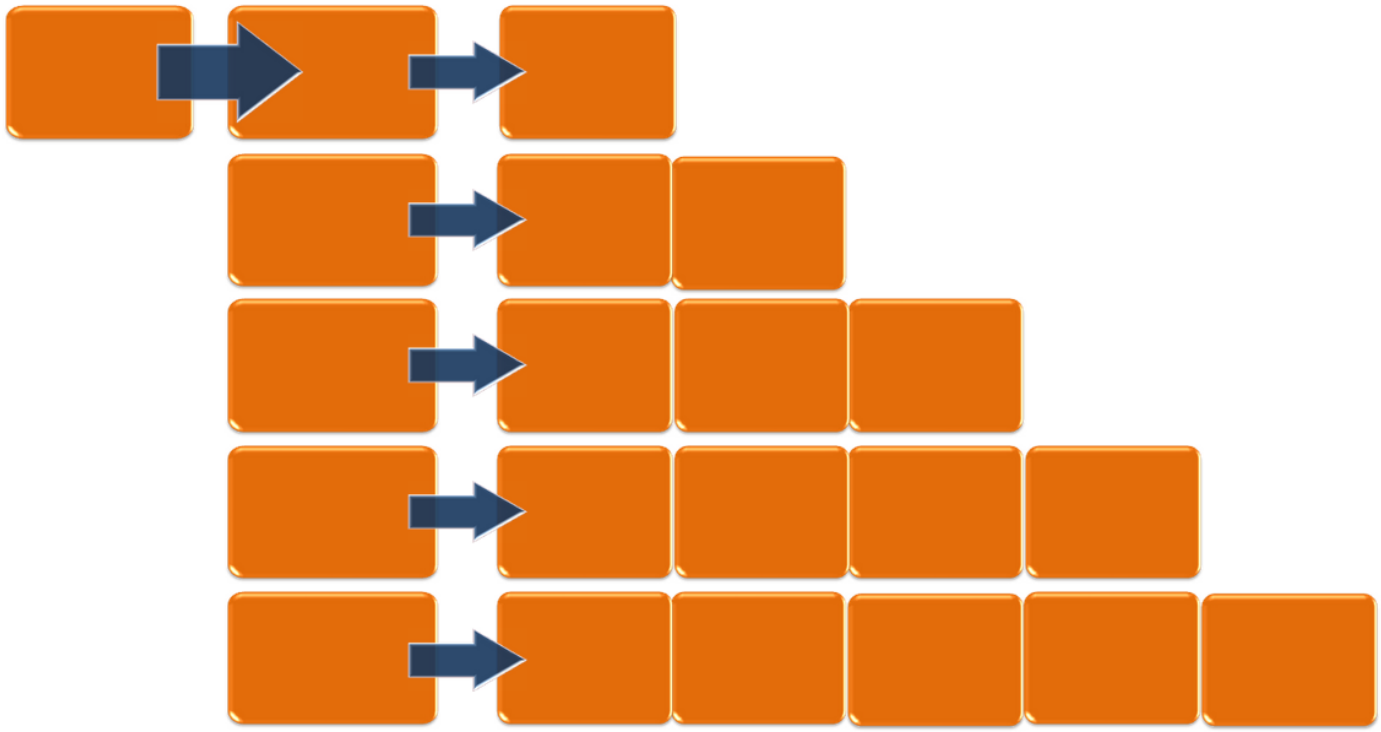
Arrays of pointers

- ptrarr[2][1]



Arrays of pointers

- every row may be of **a different length**



Arrays of pointers

```
#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;
}
```