IP of the Max-Planck-Institute for Physics of Complex Systems (mpipks),
Nöthnitzer Straße 38, 01187 Dresden

MPI PKS C++ Lecture

Bennet Becker

2019-09-09

MPI PKS

Compound data types

Pointers

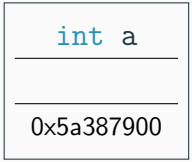
Pointers

- earlier, variables have been explained as locations in memory which can be accessed by their identifier (their name)
- This way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable
- For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address
- memory cells are ordered in a way that data representations larger than one byte to occupy memory cells that have consecutive addresses.

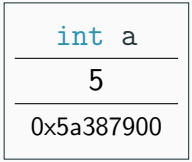
Pointers - operators

- The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator. For example: `foo = &myvar;`
- pointers can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (*). The operator itself can be read as "value pointed to by". `baz = *foo;`

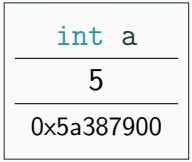
Pointers i

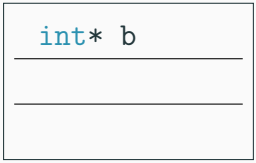


Pointers i

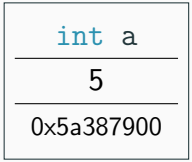


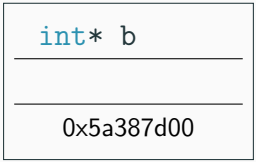
Pointers i



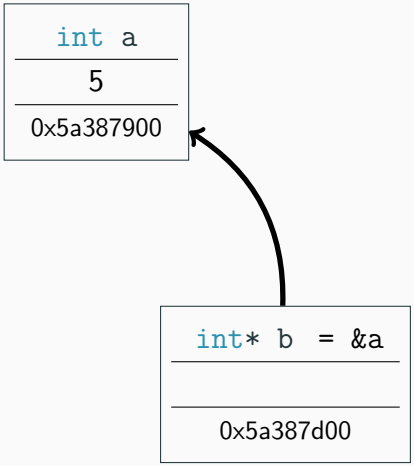


Pointers i

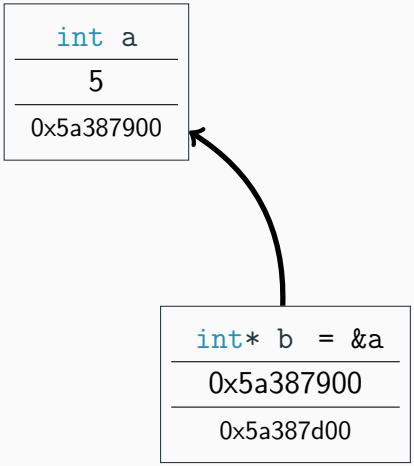




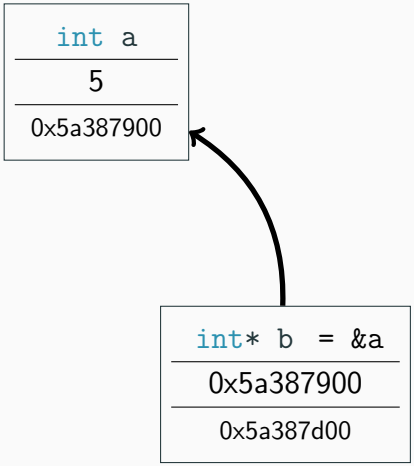
Pointers i



Pointers i

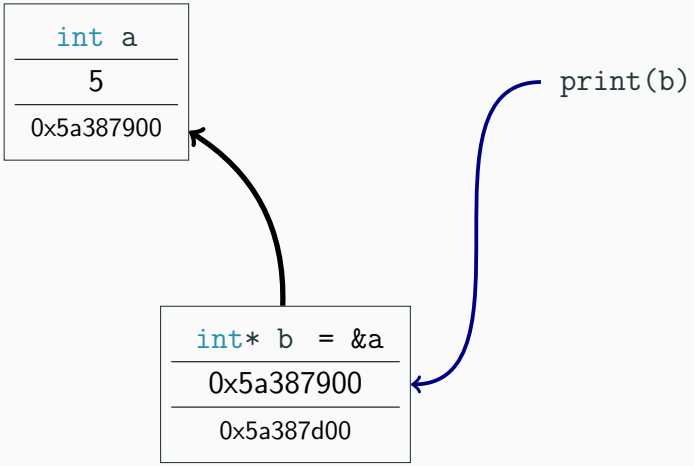


Pointers i

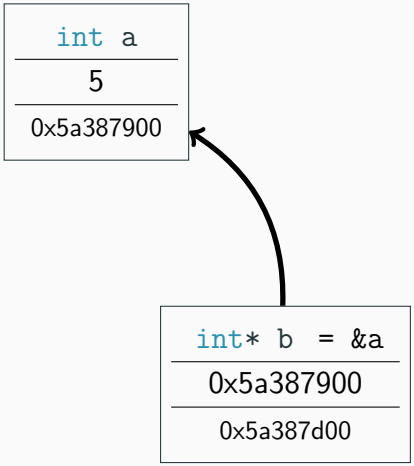



```
print(b)
```

Pointers i

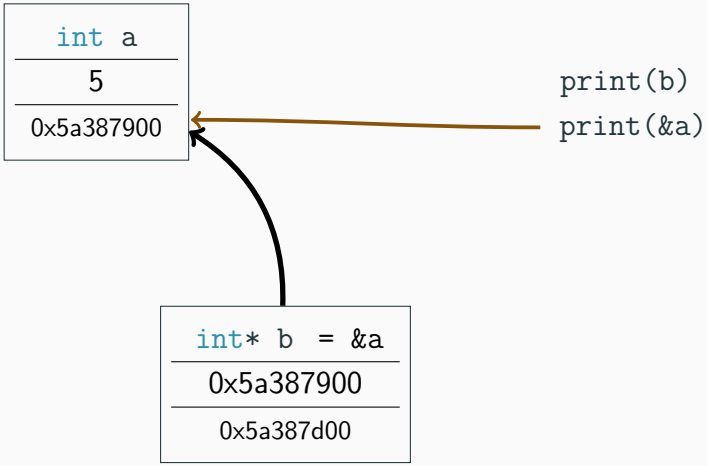


Pointers i

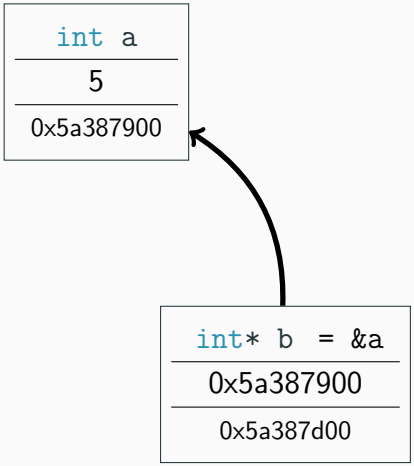



```
print(b)
print(&a)
```

Pointers i

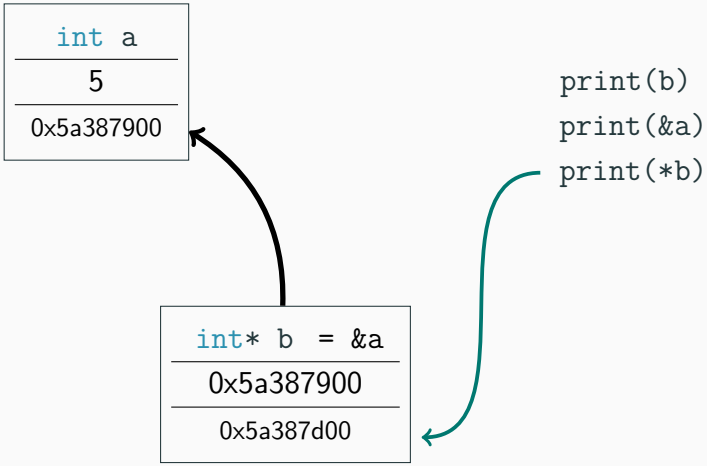

int a
5
0x5a387900
print(b)
print(&a)
int* b = &a
0x5a387900
0x5a387d00

Pointers i

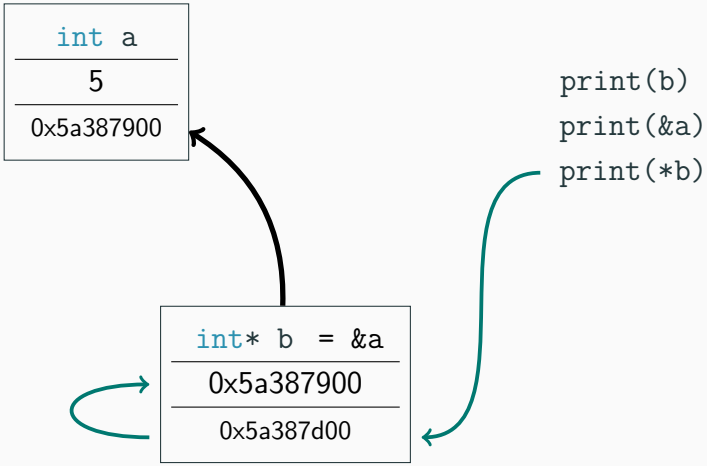



```
print(b)
print(&a)
print(*b)
```

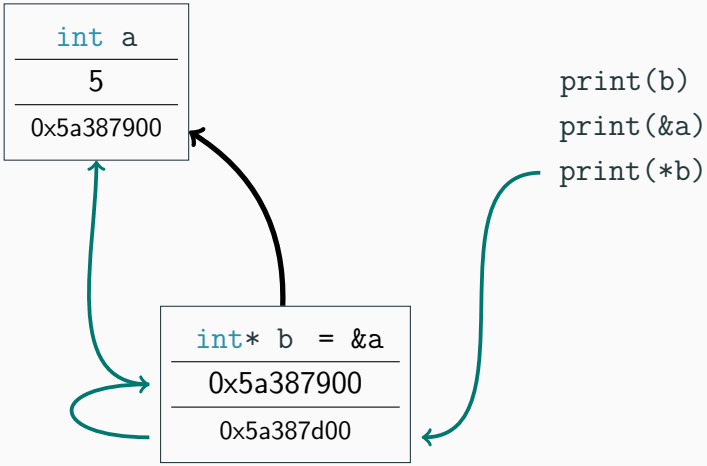
Pointers i


int a
5
0x5a387900
print(b)
print(&a)
print(*b)
int* b = &a
0x5a387900
0x5a387d00

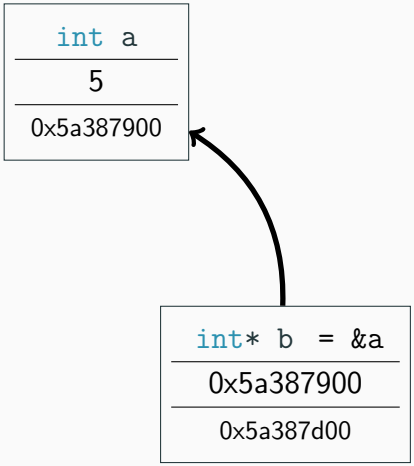
Pointers i


int a
5
0x5a387900
print(b)
print(&a)
print(*b)
int* b = &a
0x5a387900
0x5a387d00

Pointers i

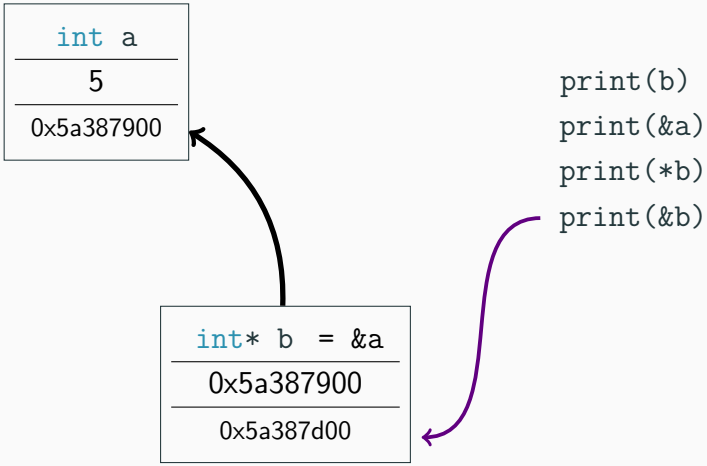

int a
5
0x5a387900
int* b = &a
0x5a387900
0x5a387d00
print(b)
print(&a)
print(*b)

Pointers i

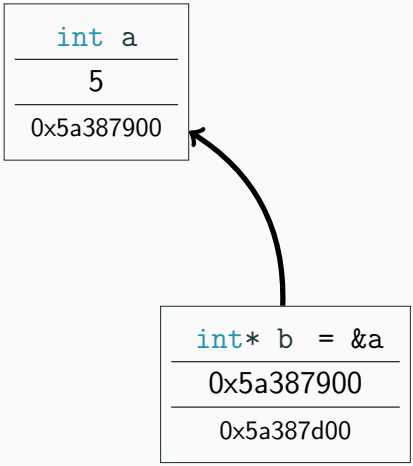



```
print(b)
print(&a)
print(*b)
print(&b)
```

Pointers i

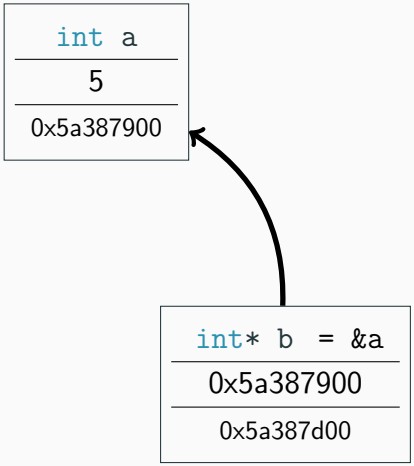

int a
5
0x5a387900
int* b = &a
0x5a387900
0x5a387d00
print(b)
print(&a)
print(*b)
print(&b)

Pointers ii



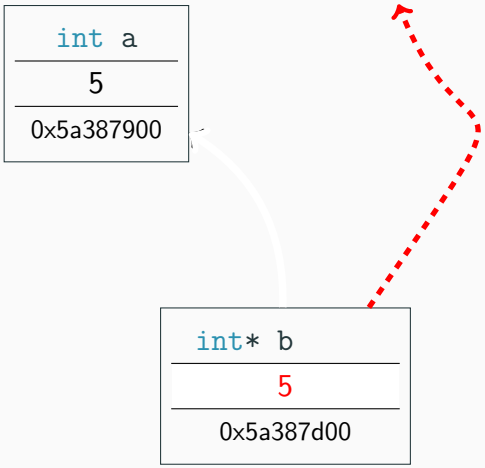

```
b = &a
```

Pointers ii



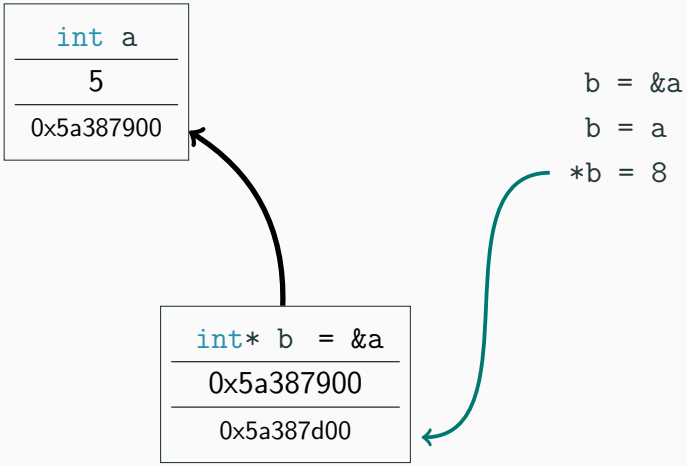

```
b = &a
b = a
```

Pointers ii

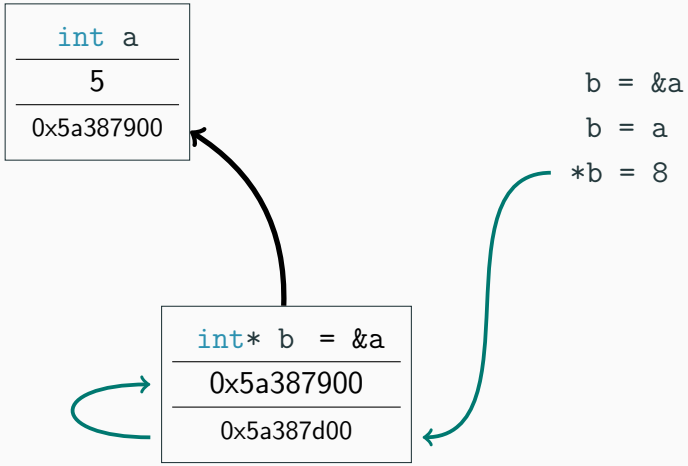



```
b = &a
b = a
```

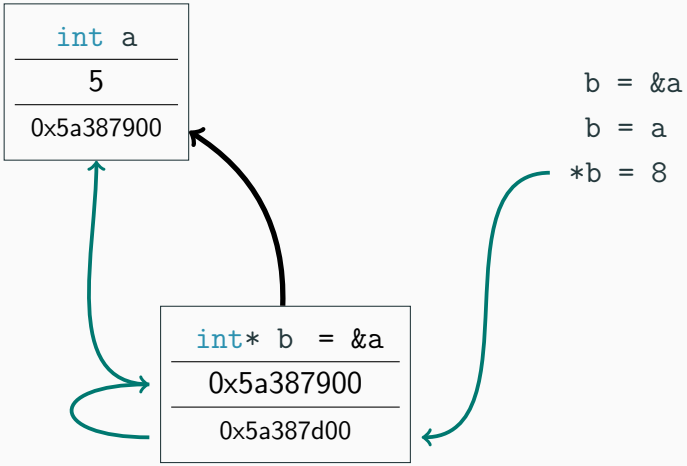
Pointers ii



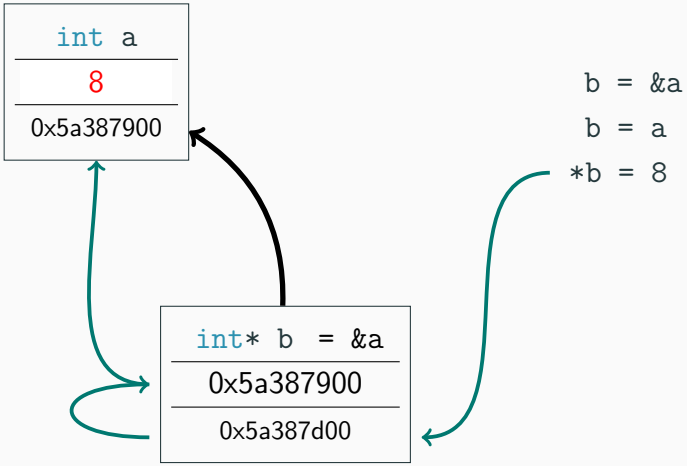
Pointers ii



Pointers ii



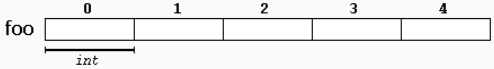
Pointers ii



Arrays and Library arrays

Arrays i

- series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier
- For example, an array containing 5 integer values of type int called foo could be represented as:



- each blank panel represents an element of the array
- elements are numbered from 0 to 4, being 0 the first and 4 the last

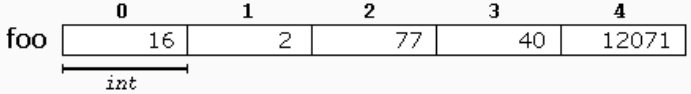
Arrays ii

- Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is: `type name [elements];`
- where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.
- Therefore, the foo array, with five elements of type int, can be declared as:
```
int foo [5]
```
- **NOTE**: The elements field within square brackets [], representing the number of elements in the array, must be a constant expression, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

array initialization i

- regular arrays of local scope are left uninitialized
- elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}

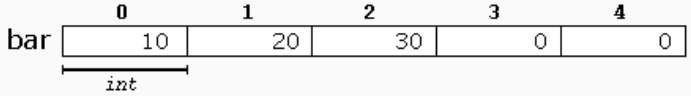
```
int foo [5] = { 16, 2, 77, 40, 12071 };
```



array initialization ii

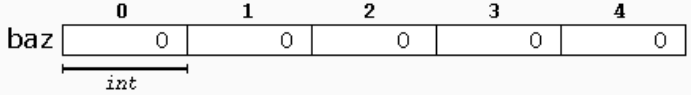
- number of values between braces shall not be greater than the number of elements in the array.
- If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```



- The initializer can even have no values, just the braces:

```
int baz [5] = { };
```



array initialization iii

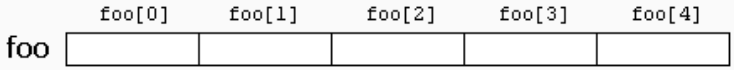
- When an initialization of values is provided it is possible to leave the square brackets empty []. the compiler will assume automatically a size that matches the number of values included between the braces

```
int foo [] = { 0, 8, 15, 42, 1337 };
```

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values

Accessing Array Values

- straightforward name, followed by the index in `[]` : `name[index]`



- index starts at 0 ends at index - 1
- out-of-bounds access is not checked on compile-time, it is an runtime-error

TASK: dot product

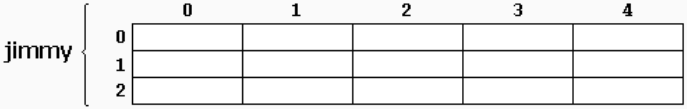
Assume two equally sized array to be two vectors. Write a program that calculates the dot product of the vectors.

You can initialize the arrays with static values, random values (`srand()` *and* `rand()` *) or read the values from the terminal.*

Bonus Task: calculate the cross product of the vectors too.

Multidimensional arrays i

- can be described as "arrays of arrays". E.g. bidimensional array imagined two-dimensional table



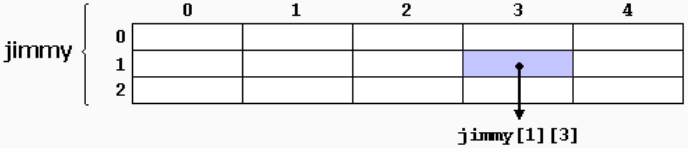
- jimmy represents a bidimensional array of 3 per 5 elements of type int.

```
int jimmy [3][5];
```

Multidimensional arrays ii

- reference the second element vertically and fourth horizontally:

  `jimmy[1][3]`



Multidimensional arrays iii

- not limited to two indices (i.e., two dimensions). can contain as many indices as needed.
- Although be careful: the amount of memory needed for an array increases exponentially with each dimension
  `char century [100][365][24][60][60];` is an array with 3 billion chars, consuming 3Gib of memory

Multidimensional Illusion i

- multidimensional arrays are just an abstraction for programmers
- same result can be achieved with a simple array

```
int jimmy [3][5];    // is equivalent to
int jimmy [15];      // (3 * 5 = 15)
```

- With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension

Multidimensional Illusion ii

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main () {
  for (n=0; n<HEIGHT; n++) {
    for (m=0; m<WIDTH; m++) {
      jimmy[n][m]=(n+1)*(m+1);
    }
  }
}
```

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main () {
  for (n=0; n<HEIGHT; n++) {
    for (m=0; m<WIDTH; m++) {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
  }
}
```

TASK: matrix multiplication

Assume 3 bidimensional arrays to be 3 matrices. Write a Program that for matrices $A_{n\times n}$, $B_{n\times n}$ and $C_{n\times n}$ calculates $C = A \times B$.

Use the ijk-algorithm and the ikj-algorithm and compare runtime. Can you image why one is faster than the other?

Arrays as Parameters i

- At some point, we may need to pass an array to a function as a parameter
- But it is not possible to pass the entire block of memory to a function directly as an argument
- what can be passed instead is its address

Arrays as Parameters ii

- To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. `void procedure (int arg[])`
- function accepts a parameter of type "array of int" called arg. to pass an array declared as `int myarray [40];`
  it would be enough to write a call like this: `procedure (myarray);`
- But the function accepts any int array of any size, so we would need to pass the size of the array too.

Arrays as Parameters iii

- In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:
  `base_type[][depth][depth]`
- Notice that the first brackets [] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension
- passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer.

Library arrays i

- arrays explained above are directly implemented as a language feature, inherited from the C language
- great feature, but by restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization
- C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header <array>

Library arrays ii

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
        array<int,3> myarray {10,20,30};

        for (int i=0; i<myarray.size(); ++i)
                ++myarray[i];

        for (int elem : myarray)
                cout << elem << '\n';
}
```

Vectors

- Container of dynamic size
- same benefits as library arrays

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
        vector<int> vec {0, 1, 2};

        vec.push_back(10);
        vec.push_back(20);

        for (int i=0; i < vec.size(); ++i)
                ++myarray[i];

        for (int elem : vec)
                cout << elem << '\n';
}
```

Character Sequences

Character sequences i

- strings are sequences of characters. they can be represented as plain arrays of elements of a character type
- `char foo [20];` is an array that can store up to 20 elements of type char
- capacity to store sequences of up to 20 characters. But this capacity does not need to be fully exhausted: the array can also accommodate shorter sequences

Character sequences ii

- By convention, the end of strings represented in character sequences is signaled by a special character: the *null character* `\0`
- `char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };`
- `char myword[] = "Hello";`
- `const char* mystring = "Hello";`

structs, unions, enums

Data structures - struct i

- group of data elements grouped together under one name

```
struct type_name {
        member_type1 member_name1;
        member_type2 member_name2;
        member_type3 member_name3;
        //...
        //...
} object_names;
```

Data structures - struct ii

```
struct product {
        int weight;
        double price;
} apple, banana, melon;
```

Data structures - struct iii

```
struct product {
        int weight;
        double price;
} ;

product apple;
product banana, melon;
```

Data structures - struct iv

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Pointers to structures i

```
int main(int argc, const char** argv){
        product apple;
        product* p_apple = & apple;

        std::cout << apple.weight << apple.price;
        std::cout << p_apple->weight << p_apple->price;
        std::cout << (*p_apple).weight << (*p_apple).price;
}
```

Unions i

- Unions allow one portion of memory to be accessed as different data types

```
union type_name {
        member_type1 member_name1;
        member_type2 member_name2;
        member_type3 member_name3;
        //...
        //...
} object_names;
```

Unions ii

- creates a new union type, identified by `type_name`
- all its member elements occupy the **same physical space in memory**
- size of this type is the one of the largest member element

```
union mytypes_t {
        char c;
        int i;
        float f;
} mytypes;
```

- Each of the members are usually of a different data type
- since all of them are referring to the same location in memory, the modification of one of the members will affect the value of all of them

Enumerated types - enum i

- Enumerated types are types that are defined with a set of custom identifiers, known as enumerators, as possible values
- Objects of these enumerated types can take any of these enumerators as value

Enumerated types - enum ii

```
enum type_name {
        value1,
        value2,
        value3,
        //.
        //.
} object_names;
```

Enumerated types - enum iii

```
enum colors_t {black, blue, green, cyan, red, purple, yellow,
↪  white};

colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

- Values of enumerated types declared with enum are implicitly convertible to an integer type, and vice versa

Enumerated types - enum iv

- elements of such an enum are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to or from
- If it is not specified otherwise, the integer value equivalent to the first possible value is 0, the equivalent to the second is 1, to the third is 2, and so on...

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december}
                 ↪  y2k;
```

Enumerated types with enum class

- real enum types that are neither implicitly convertible to int and that neither have enumerator values of type int
- they are of the enum type itself, thus preserving type safety

```
enum class Colors {black, blue, green, cyan, red, purple,
↪  yellow, white};

Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```