Arrays and Objects

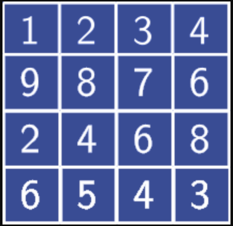
Ramses van Zon

PHY1610, Winter 2024

What is an array?

- A number of elements of the same type
- These elements are organized into a 1d, 2d, 3d, . . . rectangular shape or grid.
- 1-dimensional array = vector: grid is trivial.
- 2-dimensional array = matrix: grid represent rows and columns
- 3-and-higher-dimensional sometimes called a tensor, generalization of matrix.

Let's look at the case of a matrix in particular:

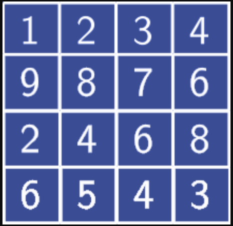


1	2	3	4
9	8	7	6
2	4	6	8
6	5	4	3

- 16 integers
- 4 rows
- 4 columns
- shape or grid: 4x4
- two dimensions, both are 4

Automatic arrays (spoiler: don't use)

```
int main()
{
    int matrix[4][4] = {{1,2,3,4},
                        {9,8,7,6},
                        {2,4,6,8},
                        {6,5,4,3}};
    // ...
}
```



1	2	3	4
9	8	7	6
2	4	6	8
6	5	4	3

You've seen automatic (a.k.a. static) arrays already.

They are only useful if you know the size of your array ahead of time and they have modest dimensions.

What we also saw that array expressions are equivalent to pointers to the first element.

This has dire consequences.

But first, let's consider what is good about this when calling functions.

Passing arrays to functions

- C++ functions pass arguments by value.

  If arrays were first-class types, the whole array would need to be copied to the function.

- Due to the pointer conversion, the function takes a copy of the pointer to the first element of the array, and is able to manipulate the original array in memory, without making a copy of all the elements.

- Different sized arrays are equivalent to the same pointer type, so you can use the function for arrays of different sizes.
  As well as for dynamic arrays.

- But we must pass the size as an argument to make up for this flexibility.

```cpp
// cookies.cpp
#include <iostream>
int sum_arr(const int* arr, int n) {
   int total = 0;
   for (int i = 0; i < n; i++)
      total += arr[i];
   return total;
}
int main() {
   int cookies[] = {1, 2, 4, 8, 16, 32, 64, 128};
   int sum = sum_arr(cookies, 8);
   std::cout<<"Cookies eaten: "<<sum<<"\n";
   int* mc = new int[5] {1, 2, 4, 8, 16};
   int sum2 = sum_arr(mc, 5);
   std::cout<<"More Cookies eaten: "<<sum2<<"\n";
   delete[] mc;
}
```

```
$ g++ -std=c++17 -O2 cookies.cpp -o cookies
$ ./cookies
Cookies eaten: 255
More cookies eaten: 31
```

How about a multidimensional array?

- Computer memory addresses are **linear**.
- So a bunch of numbers has no shape.
- We could represent the array linearly by storing row after row:

1	2	3	4
9	8	7	6
2	4	6	8
6	5	4	3

```
int linear(int row, int col, int nrow, int ncol) {
   return row*ncol + col;
}
int main() {
    int nrows=4, ncols=4;
    int linarr[nrows*ncols]
    = {1,2,3,4,9,8,7,6,2,4,6,8,6,5,4,3};
    int r = 1, c = 2;
    int i = linear(r,c,nrows,ncols);
    linarr[i] = 5.0;
}
```

But C++ natively has automatic multidimensional arrays. *(spoiler: don't use)*

```
int main() {
    int nrows=4, ncols=4;
    int arr[nrows][ncols]
    = {{1,2,3,4},{9,8,7,6},{2,4,6,8},{6,5,4,3}};
    int r = 1, c = 2;
    arr[r][c] = 5.0;
}
```

- Stores elements row by row: row major
- When passing arrays to functions, these get converted to pointers to the first element.

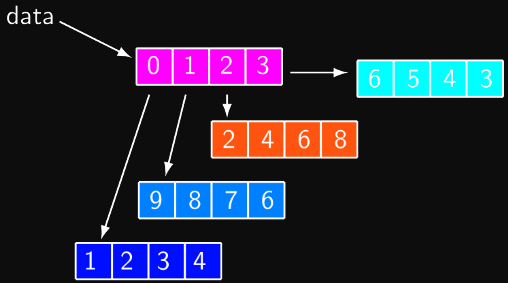
  Which is of type `int[][4]`.

Structure of 2D arrays

If a 1D array is actually a pointer to a block of memory, a 2D array is a pointer to a block of memory which contains pointers to other blocks of memory.

```
int data[4][4] = {{1,2,3,4}, {9,8,7,6}, {2,4,6,8}, {6,5,4,3}};
```

C++ arrays are stored in memory in blocks of rows: row major.



The compiler considers this an array of 4 `int[4]`'s.
Automatic arrays: compiler creates row pointers.
Dynamic arrays, must construct array of pointers.

If you pass a native multidimensional array to a function, you must specify the dimensions of the array as part of the type, so that C++ knows how to index things properly:

```
int sum_arr(int data[][4], int size);
```

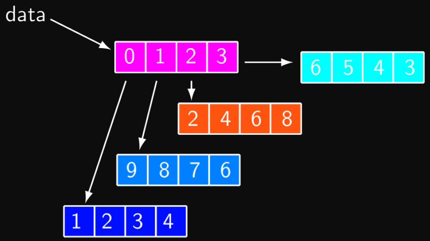
You'd need a separate function for every possible column dimension.

2D dynamic arrays, two ways

If we allocate 2d arrays ("matrix") dynamically, we can write functions that can take any size 2d array.

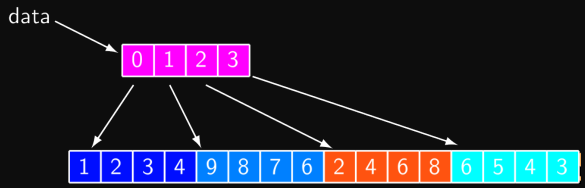
One can think of different ways to allocate a matrix:

Allocate a matrix, method 1 (textbook)



```
int** allocate_matrix1(int n, int m) {
   int** a = new int*[n];
   for (int i = 0; i < n; i++)
      a[i] = new int[m];
   return a;
}
```

Allocate a matrix, method 2



```
int** allocate_matrix2(int n, int m) {
   int** a = new int*[n];
   a[0] = new int[n * m];
   for (int i = 1; i < n; i++)
       a[i] = &a[0][i * m];
   return a;
}
```

It turns out many libraries need contiguous memory! So use the second one

Deallocating 2D arrays

The second method allocates continuous memory, while the first does not.

This affects how you should deallocate.

The discontinuous block can be deleted like this:

```
void deallocate_matrix1(int** a, int numrows) {
  for (int i = 0; i < numrows; i++)
      delete [] a[i];
  delete [] a;
}
```

The continuous block is deleted like this:

```
void deallocate_matrix2(int** a) {
   delete[] a[0];
   delete[] a;
}
```

Note that `delete[]` must be called as many times as `new ...[]` was called.

Use pointers for dynamic arrays function arguments

```
int sum_arr(int** p, int n, int m) {
   int total = 0;
   for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++)
         total += p[i][j];
   return total;
}
```

```
int** allocate_matrix2(int n, int m) {
   int** a = new int*[n];
   a[0] = new int[n * m];
   for (int i = 1; i < n; i++)
       a[i] = &a[0][i * m];
   return a;
}
```

```
void deallocate_matrix2(int** p) {
   delete [] p[0];
   delete [] p;
}
```

```
#include <iostream>
int main()
{
   int numrows = 3, numcols = 4;
   int** p = allocate_matrix2(numrows, numcols);
   for (int i = 0; i < numrows; i++) {
      for (int j = 0; j < numcols; j++) {
         p[i][j] = i + j;
         std::cout << p[i][j] << " ";
      }
      std::cout << std::endl;
   }
   std::cout << "Total="
             << sum_arr(p,numrows,numcols)
             << std::endl;
   deallocate_matrix2(p);
}
```

By the way: Accessing memory efficiently

Which is faster?

```
int sum_arr1(int** p, int numrows, int numcols) {
   int total = 0;
   for (int i = 0; i < numrows; i++)
      for (int j = 0; j < numcols; j++)
          total += p[i][j];
   return total;
}
```

or

```
int sum_arr2(int** p, int numrows, int numcols) {
   int total = 0;
   for (int j = 0; j < numcols; j++)
      for (int i = 0; i < numrows; i++)
         total += p[i][j];
   return total;
}
```

Why?

First one traverses p linearly in contiguous memory. That's what memory subsystems are best at.
The second case jumps around in memory, and makes the memory subsystem work harder (look up "cache").

When looping over blocks, arrange your arrays so that you are looping over the last index for row major languages.

There are several object oriented C++ packages available to allow you to deal with multi-dimensional arrays. Let's look at what "object oriented" means first.

Object Oriented programming

Object Oriented Programming (OOP)

- Complexity can be hidden inside each component.
- Can separate interface from the implementation.
- Can have multiple instances of the same type of component.
- Reuse of components.
- Same interface can be implemented by different objects.
- Objects' functionality can be extended using inheritance.

At a low level, OOP may need to be broken for best performance.

C objects: Structs

C already had objects, sort of; it calls them structs.

```
struct Point2D {
  int j;
  double x, y;
};

void print1(struct Point2D a) {
  std::cout << a.j << ' ' << a.x << ' ' << a.y;
}

void print2(struct Point2D* a) {
  std::cout << a->j << ' ' << a->x << ' ' << a->y;
}
```

- Calling `print1` make a copy of the content of a.
- Calling `print2` only copies *address* of a.
- Memory copies are not cheap!
- If we do this with classes, a special function called the *constructor* is called every time a copy takes place, i.e., every time `print1` is called.

C++ way of avoiding copies:

```
void print3(const Point2D& a) {
  std::cout << a.j << ' ' << a.x << ' ' << a.y;
}
```

- & makes this a call by reference; no copy!
- In C++, we can omit the `struct` keyword; `Point2D` becomes a new *data type*.

Encapsulation in Objects

- In true OOP, data is encapsulated and accessed using methods specific for that (kind of) data.
- The interface (collection of methods) should be designed around the meaning of the actions: abstraction.
- Programs typically contain multiple objects of the same type, called instances.

Classes and Objects

- Objects in C++ are made using ‘classes’.
- A class is a type of object.
- From a class, one creates 1 or more instances.
- These are the objects.
- Syntactically, classes are structs with member functions.

How do we add these member functions?

Declaration:

```
class Point2D {
public:
  int j;
  double x,y;
  void set(int aj, double ax, double ay);
};
```

Definition:

```
void Point2D::set(int aj, double ax, double ay)
{
    j = aj;
    x = ax;
    y = ay;
}
```

How do we use the class?

```
int main() {
    Point2D myobject;
    myobject.set(1, -0.5, 3.14);
    std::cout << myobject.j << std::endl;
}
```

Data hiding

- Good components hide implementation details
- Each member function or data member can be
  1. private: only member functions of class have access
  2. public: accessible from anywhere
- These are specified as sections within the class.

Example (Declaration)

```
class Point2D {
private:
    int j;
    double x,y;
public:
    void set(int aj,double ax,double ay);
    int get_j();
    double get_x();
    double get_y();
};
```

Data hiding

Example (Definition)

```
int Point2D::get_j() {
    return j;
}
double Point2D::get_x() {
    return x;
}
double Point2D::get_y() {
    return y;
}
```

Example (Usage)

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.get_j() << std::endl;
```

WARNING:

When hiding the data through these kinds on accessor functions, now, each time the data is needed, a function has to be called, and there's an overhead associate with that.

- The overhead of calling this function can sometimes be optimized away by the compiler, but often it cannot.
- Considering making data is that is needed often by an algorithm just public, or use a `friend` .

Class > Struct

- A class defines a type, and when an instance of that type is declared, memory is allocated for that struct.
- A class is more than just a chunk of memory.

  For example, arrays may have to be allocated (`new ...`) when the object is created.
- When the object ceases to exist, some clean-up may be required (`delete ...`).
- Constructor

  is called when an object is created.
- Destructor

  is called when an object is destroyed.

Constructors

Example

```
class Point2D {

private:
    int j;
    double x,y;

public:
    Point2D(int aj,double ax,double ay);
    int get_j();
    double get_x();
    double get_y();
};

Point2D::Point2D(int aj,double ax,double ay) {
    j = aj;
    x = ax;
    y = ay;
}

Point2D myobject(1,-0.5,3.14);
```

Destructors

- ... are called when an object is destroyed.
- This occurs when a non-static object goes out-of-scope, or when delete is used.
- Good opportunity to release memory.
- Declare destructor as a member functions of the class with no return type, with a name which is the class name plus a $\sim$ attached to the left.
- A destructor cannot have arguments.

And this is just the beginning of OOP with C++.

- There's far more that one can learn about objects in C++.

  For example: inheritance, operating overloading, friends, automatic conversions, ...
- These are getting to an advanced level. You can look them up as you please.
- A note about using C++ objects with HPC: objects can be very elegant, and simplify coding considerably, especially given the right context, such as data analysis. However, objects come with a fair amount of overhead, and can slow down your code considerably under certain circumstances.

Better array alternative: the rarray library

- https://github.com/vanzonr/rarray
- Fast, arbitrary rank, same access syntax as C arrays, row major.
- Header-only library:
  - https://raw.githubusercontent.com/vanzonr/rarray/main/rarray

Example:

```cpp
// rarrayex.cc
#include <rarray>
#include <iostream>
int main() {
    rarray<double,2> a(3,3);
    a.fill(4);
    for (int i=0; i<3; i++)
        for (int j=0; j<i; j++)
            a[i][j] = i+j;
    std::cout << a << std::endl;
}
```

```
$ g++ -std=c++17 -O2 rarrayex.cc -o rarrayex
$./rarrayex
{
{4,4,4},
{1,4,4},
{2,3,4}
}
```

Note: compile with optimization (at least -O2), otherwise, the abstraction layers of rarray make things unnecessarily slow.

About rarray

- Written specifically as an optimized way to circumvent the weirdness of C++ arrays.
- Similar syntax as builtin C++ arrays (repeated square brackets to access elements)
- Rarrays behave much like shared pointers: If you copy rarray a to rarray b, both share the same data.
- But rarrays are objects that remember their shape.
- Automatically deletes memory, but uses reference counters to avoid dangling references from copies.
- rarrays use heap memory, thus allowing for large arrays
- Data is contiguous, so rarrays are compatible with common numerical libraries (BLAS, FFTW, ...) (the libaries will expect pointers, which you get through `.data()`)
- General type is `rarray<TYPENAME,RANK>`, where rank is a positive integer. Shorthands:
  - `rarray<TYPENAME,1> = rvector<TYPENAME>`
  - `rarray<TYPENAME,2> = rmatrix<TYPENAME>`
  - `rarray<TYPENAME,3> = rtensor<TYPENAME>`

Rarray in a Nutshell

Define a n×m×k array of doubles:	`rarray<double,3> b(n,m,k);`
Define it with preallocated memory:	`rarray<double,3> c(ptr,n,m,k);`
Element i,j,k of the array b:	`b[i][j][k]`
Pointer to the contiguous data in b:	`b.data()`
Total number of elements in b:	`b.size()`
Extent in the ith dimension of b:	`b.extent(i)`
Array of all extents of b:	`b.shape()`
Define an array with same shape as b:	`rarray<double,3> b2(b.shape());`
Reference-counter copy of the array:	`rarray<double,3> d=b;`
Deep copy of the array:	`rarray<double,3> e=b.copy();`
Output a rarray:	`std::cout << h << endl;`
Read in a rarray:	`std::cin >> h;`

From C++ arrays to rarrays

Consider this code using C++ arrays:

```
#include <iostream>
int main()
{
   double a[] = {0, 1, 2, 3, 4};
   double *b = new double[5];

   for (int i = 0; i < 5; i++)
      b[i] = i;

   for (int i = 0; i < 7; i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;

   for (int i = 0; i < 7; i++)
      std::cout << b[i] << " ";
   std::cout << std::endl;

   delete [] b;
}
```

We can do the same with rarrays:

```
#include <rarray>
int main()
{
   rvector<double> a;
   a.form({0, 1, 2, 3, 4});

   rvector<double> b(5);
   for (int i = 0; i < 5; i++)
      b[i] = i;

   rvector<double> c = linspace<double>(0,4,5);

   std::cout << a << std::endl;
   std::cout << b << std::endl;
   std::cout << c << std::endl;
}
```

Passing an rarray to a function

With plain C++ arrays, we needed pass the size:

```
#include <iostream>
int sum_arr(int arr[], int n) {
   int total = 0;
   for (int i = 0; i < n; i++)
      total += arr[i];
   return total;
}
int main() {
   const int arrsize = 8;
   int cookies[arrsize] = {1,2,4,8,16,32,64,128};
   int sum = sum_arr(cookies, arrsize);
   std::cout << "Cookies eaten: " << sum << "\n";
}
```

or, with dynamic arrays:

```
int main() {
   const int arrsize = 8;
   int* cookies=new int[arrsize]{1,2,4,8,16,32,64,12
   int sum = sum_arr(cookies,arrsize);
   std::cout << "Cookies eaten: " << sum << "\n";
   delete[] cookies;
}
```

With rarrays, the size comes with the array, accessible through the `size()` or `extent()` method:

```
#include <iostream>
#include <rarray>

int sum_arr(rvector<int> arr) {
   int total = 0;
   for (int i = 0; i < arr.size(); i++)
      total += arr[i];
   return total;
}

int main() {
   rvector<int> cookies;
   cookies.form({1, 2, 4, 8, 16, 32, 64, 128});
   int sum = sum_arr(cookies);
   std::cout << "Cookies eaten: " << sum << "\n";
}
```

Comparing passing 2d arrays

```cpp
#include <iostream>
int sum_arr(int** p, int nrows, int ncols) {
  int total = 0;
  for (int i = 0; i < nrows; i++)
    for (int j = 0; j < ncols; j++)
      total += p[i][j];
  return total;
}
int** allocate_matrix2(int nrows, int ncols) {
  ....
}
void deallocate_matrix2(int** p) {
  ....
}
int main() {
  int nrows=3,ncols=4
  int** p = allocate_matrix2(nrols, ncols)
  for (int i = 0; i < nrows; i++) {
    for (int j = 0; j < ncols; j++) {
      p[i][j] = i + j;
      std::cout << p[i][j] << " ";
    }
    std::cout << std::endl;
  }
  std::cout<<"Total="<<sum_arr(p,nrows,ncols)<<"\n";
  deallocate_matrix2(p, nrows);
}
```

```cpp
#include <iostream>
#include <rarray>
int sum_arr(rmatrix<int> p) {
  int total = 0;
  for (int i = 0; i < p.extent(0); i++)
    for (int j = 0; j < p.extent(1); j++)
      total += p[i][j];
  return total;
}

int main() {
  int nrows=3,ncols=4;
  rmatrix<int> p(nrows,ncols);
  for (int i = 0; i < p.extent(0); i++) {
    for (int j = 0; j < p.extent(1); j++) {
      p[i][j] = i + j;
      std::cout << p[i][j] << " ";
    }
    std::cout << std::endl;
  }
  std::cout << "Total=" << sum_arr(p) << "\n";
}
```

Returning a rarray from a function

No problem:

```
rmatrix<double> zeros(int n, int m) {
   rmatrix<double> r(n,m);
   r.fill(0.0);
   return r;
}

int main() {
   rmatrix<double> s = zeros(100,100);
   return s[99][99];
}
```

No hidden copies (rarray fully supports C++11 move semantics)

No need to deallocate s, done automatically by destructor.

For comparison: The vector class (don't do this!)

A quick warning about the vector class, which you will bump into if you start poking around. The vector class has advantages, and is quite satisfactory for 1D arrays.

However, generalizing the vector class to higher dimensions quickly becomes extremely ugly and non-contiguous.

```
using std::vector;
int n = 256;                // size per dimension
vector<vector<vector<float>>> v(n);// allocate for top dimension
for (int i=0;i<n;i++) {
   v[i].reserve(n);         // allocate vectors for middle dimension
   for (int j=0;j<n;j++)
      v[i][j].reserve(n);   // allocate elements in last dimension
}
```

Note on the future:

C++20 allows to write `a[i,j]` instead of `a[i][j]`.

Before C++20, `a[i,j]` meant `a[j]`.

C++23 has a "non-owning" multidimensional view on a pointer called mdspan.

No compiler support, but a reference implementation exists.

Non-owning: Still much allocate and deallocate your own memory first.

C++26 is rumoured that it might include an owning multidimensional array type.