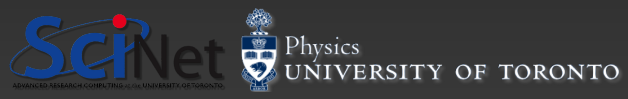
Arrays and other Data Structures

(PHY1610 – Lecture 5)

Daniel Gruner, Ramses van Zon,
Marcelo Ponce, Scott Northrup

SciNet HPC Consortium/Physics Department
University of Toronto

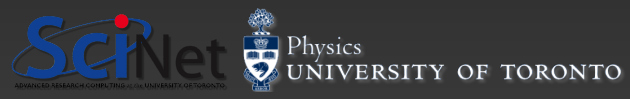
January 24th, 2017



Today's class

Today we will discuss the following topics:

- Arrays: general use
- Arrays: are actually pointers
- Arrays: multidimensional
- Arrays: memory layout
- Arrays: existing library functionality
- Data Structures
- Dynamical Data Structures
- Linear Data Structures: list
- The C++ STL
- Other Types of Data Structures: trees and maps



Arrays must be dealt with carefully

Most scientific programming depends on arrays in one form or another.
They show up everywhere:

- Fields.
- Grid information.
- Discretization.
- Linear algebra.

However, C++ was not designed with arrays in mind. You need to understand how they work to avoid the various pitfalls that can show up.

Initializing static arrays

You've seen static arrays already. They are only useful if you know the size of your array ahead of time and they have modest dimensions.
A few points about initialization:

```
int cards[4] = {3, 6, 8, 10}; // Okay.
int cards[4] {3, 6, 8, 10}; // Okay.
int hand[4]; // Okay.
hand[4] = {3, 6, 8, 10}; // Not allowed. Needs a variable type.
hand = cards; // Not allowed.
int hand[500] = {0}; // Okay. Sets all values to 0.
int cards[] = {3, 6, 8, 10}; // Okay, and encouraged
```

Beware the end of the array

```
include <iostream> // myarray.cc
int main() {
   int a[] = {0, 1, 2, 3, 4};
   int *b = new int[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;

   return 0;
}
```

Beware the end of the array

```cpp
include <iostream> // myarray.cc
int main() {
   int a[] = {0, 1, 2, 3, 4};
   int *b = new int[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;

   return 0;
}
```

```
$ g++ -std=c++11 myarray.cc \
      -o myarray
$ ./myarray
0 1 2 3 4 0 9420816
0 1 2 3 4 0 135137
```

Passing arrays to functions

```
#include <iostream> // myarray2.cc

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 << "Total␣cookies␣
       eaten:␣" << sum << std::
       endl;
    return 0;
}
```

Passing arrays to functions

```
#include <iostream> // myarray2.cc

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 << "Total␣cookies␣
        eaten:␣" << sum << std::
        endl;
    return 0;
}
```

```
$ g++ -std=c++11 myarray2.cc \
      -o myarray2
$ ./myarray2
Total cookies eaten: 255
```

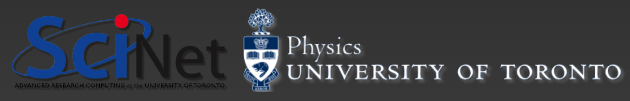
Passing arrays to functions, cont.

```
#include <iostream> // myarray3.cc
int sum_arr(int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    arr[0] += 10; 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
        );
    int sum2 = sum_arr(cookies,
        arrsize);

    std::cout << "Total␣cookies␣eaten:
        ␣" << sum << std::endl;
    std::cout << "Total␣cookies␣eaten:
        ␣" << sum2 << std::endl;
    return 0;
}
```

Passing arrays to functions, cont.

```cpp
#include <iostream> // myarray3.cc
int sum_arr(int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    arr[0] += 10; 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
        );
    int sum2 = sum_arr(cookies,
        arrsize);

    std::cout << "Total␣cookies␣eaten:
        ␣" << sum << std::endl;
    std::cout << "Total␣cookies␣eaten:
        ␣" << sum2 << std::endl;
    return 0;
}
```

```
$ g++ -std=c++11 myarray3.cc \
      -o myarray3
$ ./myarray3
Total cookies eaten: 255
Total cookies eaten: 265
```



Arrays are actually pointers

The function behaves the way it does because

- C++ functions pass arguments by value, not by reference;
- but an array variable is actually a pointer to the first element of the array, not the whole array itself;
- thus the function takes a copy of the pointer to the array, and is able to manipulate the original array in memory, without making a copy of it.
- This saves on memory, and is faster, since the array isn't being copied.
- If you want your array to be protected from being modified by a function, include the `const` flag in the function prototype.

Arrays are actually pointers, cont.

```
#include <iostream> // myarray4.cc
int main() {{
   float a[] = {10.0, 20.0, 30.0};
   float *b = new float[3];
   float *p = a;
   for (int i = 0; i < 3; i++)
      *(b + i) = a[i];
   std::cout << "p = " << p << " *p = "
      " << *p << std::endl;
   p = p + 1;
   std::cout << "p = " << p << ", *p =
      " << *p << std::endl;
   std::cout << "b = " << b << ", *b =
      " << *b << std::endl;
   std::cout <<"b[1]="<<b[1]<<",b[2]="
      <<*(b+2) << std::endl;
   return 0;
}
```

Arrays are actually pointers, cont.

```
#include <iostream> // myarray4.cc
int main() {{
   float a[] = {10.0, 20.0, 30.0};
   float *b = new float[3];
   float *p = a;
   for (int i = 0; i < 3; i++)
      *(b + i) = a[i];
   std::cout << "p = " << p << " *p = "
      " << *p << std::endl;
   p = p + 1;
   std::cout << "p = " << p << ", *p =
      " << *p << std::endl;
   std::cout << "b = " << b << ", *b =
      " << *b << std::endl;
   std::cout <<"b[1]="<<b[1]<<",b[2]="
      <<*(b+2) << std::endl;
   return 0;
}
```

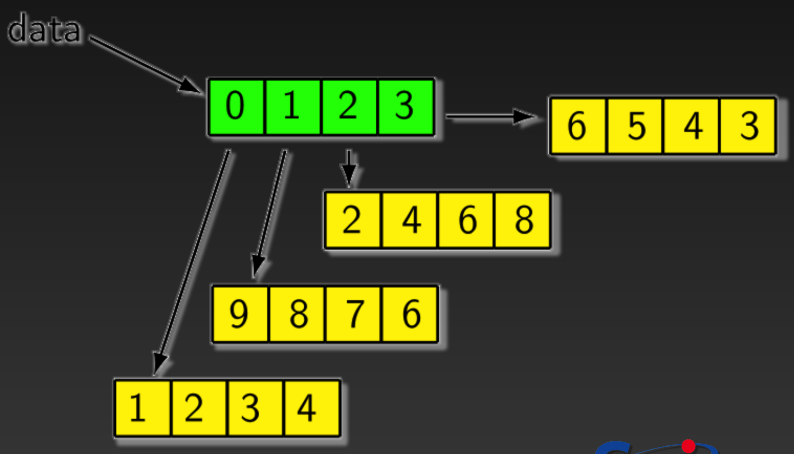
```
$ g++ -std=c++11 myarray4.cc \
      -o myarray4
$ ./myarray4
p = 0x7fff67e23b30, *p = 10
p = 0x7fff67e23b34, *p = 20
b = 0x999010, *b = 10
b[1] = 20, b[2] = 30
$
```

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}};
```

Conceptually:



2D arrays

Arrays are actually pointers; 2D arrays are pointers to arrays of pointers.

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

Note that C++ is **row major**, versus column major (as in Fortran), meaning that this array is stored in memory as written above, and is normally written

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 9 & 8 & 7 & 6 \\ 2 & 4 & 6 & 8 \end{bmatrix}$$

If you pass an array to a function, you must specify the size of the arrays being pointed to, so that C++ knows how to index things properly:

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

Use pointers for passing dynamic arrays

```
#include <iostream> //myarray7.cc
int sum_arr(int **p, const int numrows, const int numcols);
void deallocate_mem(float **p, const int numrows);

int main() {
   int numrows = 3, numcols = 4;
   int **p = new int *[numrows];

   for (int i = 0; i < numrows; i++) {
        p[i] = new int[numcols];
        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_mem(p, numrows);
    return 0;
}
```

Use pointers for passing dynamic arrays

```
// myarray7.cc, continued
int sum_arr(int **p, const int numrows,
    const 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;
}

void deallocate_mem(float **p, const int
    numrows) {
     for (int i = 0; i < numrows; i++)
          delete [] p[i];
     delete [] p;
}
```

```
$ g++ -std=c++11 \
  myarray7.cc  -o myarray7
$ ./myarray7
p is
0 1 2 3
1 2 3 4
2 3 4 5
Total = 30
```

Allocating 2D arrays

Do you understand the difference between these two functions?

```
float **allocate_matrix1(int n, int m) {
   float **a = new float *[n];

   for (int i = 0; i < n; i++)
        a[i] = new float[m];
   return a;
```

Allocating 2D arrays

Do you understand the difference between these two functions?

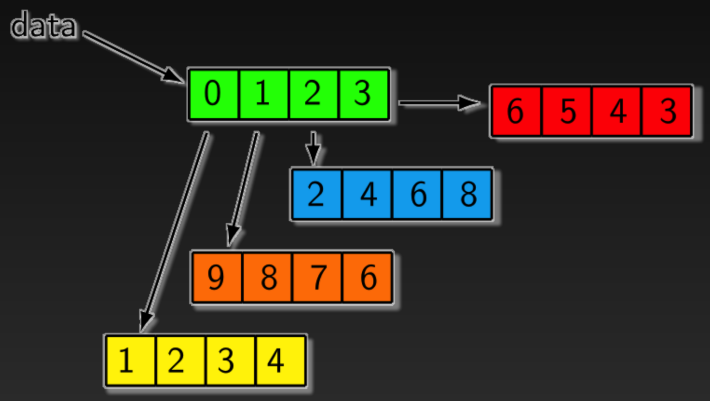
```
float **allocate_matrix1(int n, int m) {
   float **a = new float *[n];

   for (int i = 0; i < n; i++)
        a[i] = new float[m];
   return a;
```

```
float **allocate_matrix2(int n, int m) {
   float **a = new float *[n];

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

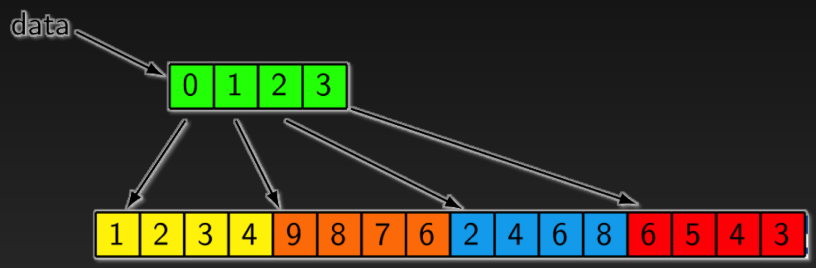
What is the advantage of the second over the first?

2D arrays

allocate_matrix1



allocate_matrix2



Deallocating 2D arrays

The second code allocates continuous memory, while the first does not. The discontinuous block can be deleted like this:

```
void deallocate_matrix1(float **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(float **a) {
   delete[] a[0];
   delete[] a;
}
```

Note that delete must be called as many times as new was called during the allocation.

Accessing memory quickly

Which is faster?

```
int sum_arr1(int **p, const int numrows, const 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;
}
```

```
int sum_arr2(int **p, const int numrows, const 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?

C++ is row major

Arrays are stored in memory in blocks of rows:

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

But those blocks aren't necessarily continuous in memory if you declare your blocks like this:

```
    float **a = new float *[n]; // First array is an array of pointer
    for (int i = 0; i < n; i++) a[i] = new float[m];
    return a;
}
```

When looping over blocks, arrange your arrays so that you are looping over the last index for row major languages, and the first index for column-major languages. Make sure this is correct for the language you are using!

Existing C++ matrix packages

There are several C++ packages available to allow you to do matrix algebra (Armadillo, Eigen, Blitz++, boost, rarray). These packages come with built-in functionality that you may need:

- All the usual matrix multiplication operations.
- Matrix inversion, solving systems of equations.
- Decompositions, and factorizations.
- Eigenvalue calculations.

Many operations have already been parallelized.
These are useful, and should be used first before trying to speed things up by building your own.
There are also the BLAS and LAPACK libraries, which are the classic libraries for doing linear algebra.

First example: Eigen

Fast vectors and matrices, column major.

http://eigen.tuxfamily.org/

```cpp
// eig1.cc
#include <iostream>
#include <Eigen/Dense>
using std::cout;
using std::endl;
using namespace Eigen;

int main() {
     // 2 x 2 matrix, of type float.
     Eigen::Matrix<float,2,2> A, B;
     A << 2, -1, -1, 3;
     B << 1, 2, 3, 1;
     cout << "A:"    <<A     << endl;
     cout << "B:"    <<A     << endl;
     cout << "A+B:" <<A+B   << endl;
     cout << "A-B:" <<A-B   << endl;
     cout << "1,6*A"<<1.6*A<< endl;
     return 0;
}
```

First example: Eigen

Fast vectors and matrices, column major.

http://eigen.tuxfamily.org/

```cpp
// eig1.cc
#include <iostream>
#include <Eigen/Dense>
using std::cout;
using std::endl;
using namespace Eigen;

int main() {
     // 2 x 2 matrix, of type float.
     Eigen::Matrix<float,2,2> A, B;
     A << 2, -1, -1, 3;
     B << 1, 2, 3, 1;
     cout << "A:"    <<A     << endl;
     cout << "B:"    <<A     << endl;
     cout << "A+B:" <<A+B   << endl;
     cout << "A-B:" <<A-B   << endl;
     cout << "1,6*A"<<1.6*A<< endl;
     return 0;
}
```

```
$ g++ -std=c++11 eig1.cc -o eig1
$ ./eig1
A:
2 -1
-1 3
B:
1 2
3 1
A + B:
3 1
2 4
A - B:
1 -3
-4 2
1.6 * A:
3.2 -1.6
-1.6 4.8
$
```

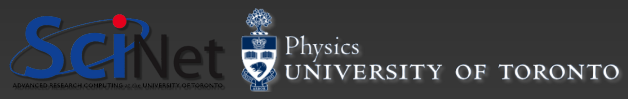
Eigen: matrix manipulation

```cpp
// eig2.cc
#include <iostream>
#include <Eigen/Dense>
using std::cout;
using std::endl;
using namespace Eigen;

int main() {
   Matrix<float, 2, 2> mat;
   mat << 1, 2, 3, 4;
   Matrix<float, 2, 1> u(-1, 1);
   Matrix<float, 2, 1> v(2, 0);
   cout << "mat␣*␣mat:" <<
      mat * mat << endl;
   cout << "mat␣*␣u:" <<
      mat * u << endl;
   cout << "u^T␣*␣mat:" <<
   u.transpose() * mat << endl;
   cout << "u^T␣*␣v:" <<
   u.transpose() * v << endl;
   return 0;
}
```

Eigen: matrix manipulation

```cpp
// eig2.cc
#include <iostream>
#include <Eigen/Dense>
using std::cout;
using std::endl;
using namespace Eigen;

int main() {
   Matrix<float, 2, 2> mat;
   mat << 1, 2, 3, 4;
   Matrix<float, 2, 1> u(-1, 1);
   Matrix<float, 2, 1> v(2, 0);
   cout << "mat * mat:" <<
      mat * mat << endl;
   cout << "mat * u:" <<
      mat * u << endl;
   cout << "u^T * mat:" <<
   u.transpose() * mat << endl;
   cout << "u^T * v:" <<
   u.transpose() * v << endl;
   return 0;
}
```

```
$ g++ -std=c++11 eig2.cc -o eig2
$ ./eig2
mat * mat:
7 10
15 22
mat * u:
1
1
u^T * mat:
2 2
u^T * v:
-2
$
```



Eigen: systems of equations

```cpp
// eig3.cc
#include <iostream>
#include <Eigen/Dense>
using std::cout;
using std::endl;
using namespace Eigen;

int main() {
   Matrix<float,2,2> A, b, x;
   A << 2, -1, -1, 1;
   A(3,3) = 3;
   b << 1, 2, 3, 1;
   cout << "A:" << A << endl;
   cout << "b:" << b << endl;
   // Solve with LU decomposition x
      = A.lu().solve(b);
   cout << "The solution is:"
        << x << endl;
   cout << "Eigenvalues of A:"
        << A.eigenvalues()<<endl;
   return 0;
}
```

```
$ g++ -std=c++11 eig3.cc -o eig3
$ ./eig3
A:
2 -1
-1 3
b:
1 2
3 1
The solution is:
1.2 1.4
1.4 0.8
Eigenvalues of A:
(1.38917,0)
(3.61803,0)
$
```

Second example: rarray

https://github.com/vanzonr/rarray

Fast, arbitrary rank, same access syntax as C arrays, row major.

```cpp
// rex.cc
#include <rarray>
#include <rarrayio>
#include <iostream>

int main() {
  const int n = 3;
  rarray<double,3> a(n,n,n);
  a.fill(4);
  for (int i=0; i<n; i++)
    for (int j=0; j<i; j++)
      for (int k=0; k<j; k++)
        a[i][j][k] =i+j+k;
  std::cout << a << std::endl;
}
```

```
$ g++ -std=c++11 -O2 rex.cc -o rex
$ ./rex
{{{4,4,4},{4,4,4},{4,4,4}},
 {{4,4,4},{4,4,4},{4,4,4}},
 {{4,4,4},{3,4,4},{4,4,4}}}
```

Third example: The vector class (don't)

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
}
```

So we aren't going to discuss it further.

DATA STRUCTURES

- Basic Types
- structures
- linked List, Stacks/Queues
- binary Trees
- Hash tables (maps)

Basic Types in C++

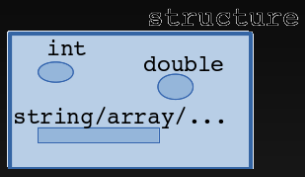
"Primitive" types

* boolean: `bool`
* integer type:
  [`unsigned`] `short`, `int`, `long`, `long long`
* floating point type:
  `float`, `double`, `long double`, std::`complex<float>`, ...
* character or string of characters:
  `char`, `char*`, `std::string`

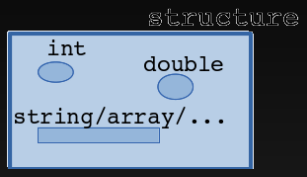
"Composite" types / "Abstract" Data Structures

* `array`, `pointer`
* `class`, `structure`, `enumerated type`, `union`

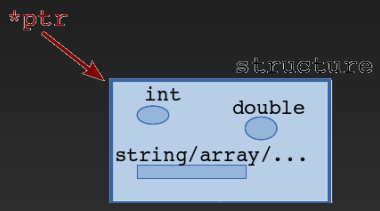
Structures:



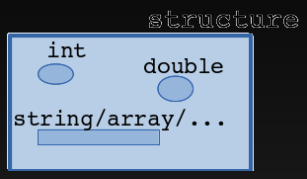
Structures:



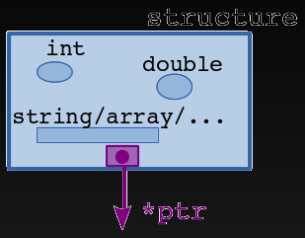
Pointer to a structure:



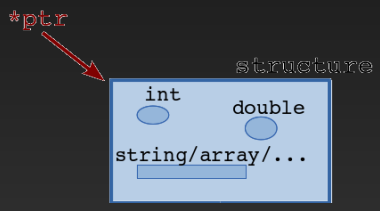
Structures:



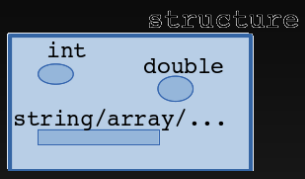
Structure w/pointer:



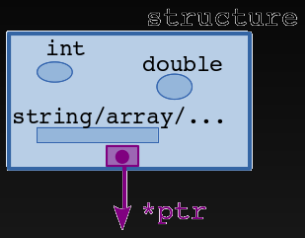
Pointer to a structure:



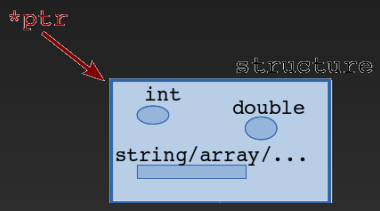
Structures:



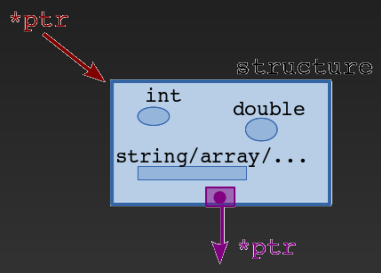
Structure w/pointer:



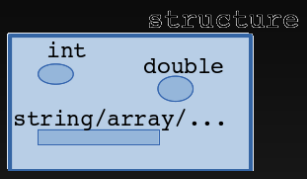
Pointer to a structure:



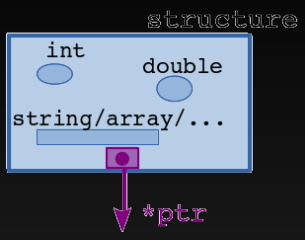
Ptr to a structure w/ptr:

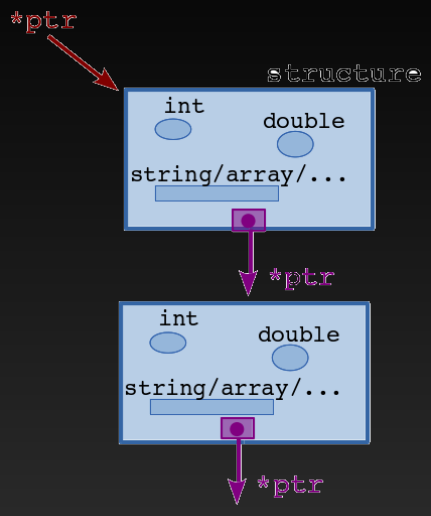


Structures:



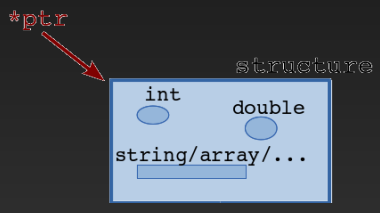
Structure w/pointer:



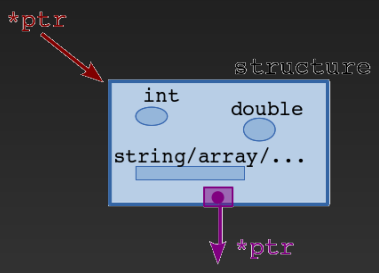


List

Pointer to a structure:



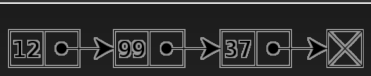
Ptr to a structure w/ptr:



Linear Data Structures

Linked Lists

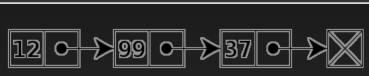
```
struct node
{
   int val;
   struct node *next;
};


class Node
{
   int val;
   Node *next;
};
```



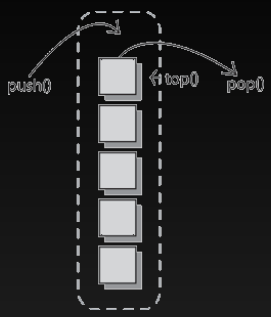
Linear Data Structures

Linked Lists

```
struct node
{
   int val;
   struct node *next;
};


class Node
{
   int val;
   Node *next;
};
```



Stack

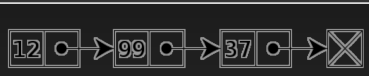


- LIFO

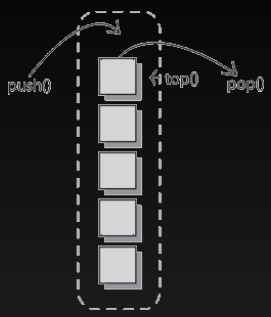
Linear Data Structures

Linked Lists

```
struct node
{
   int val;
   struct node *next;
};


class Node
{
   int val;
   Node *next;
};
```

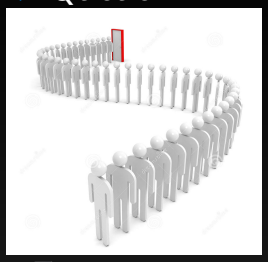


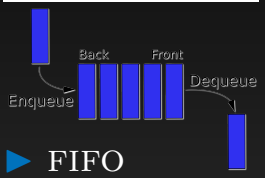
Stack



- LIFO

Queue



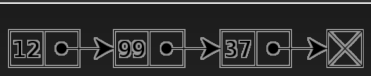


- FIFO

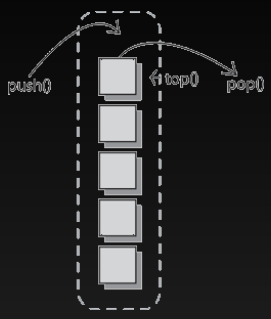
Linear Data Structures

Linked Lists

```
struct node
{
    int val;
    struct node *next;
};


class Node
{
    int val;
    Node *next;
};
```

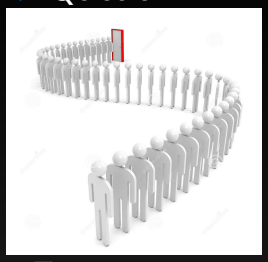


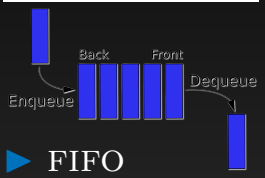
Stack



- LIFO

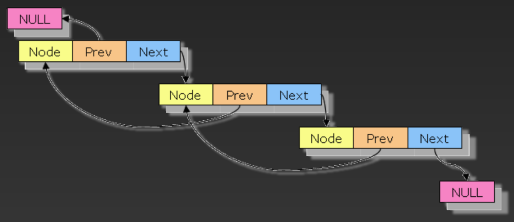
Queue

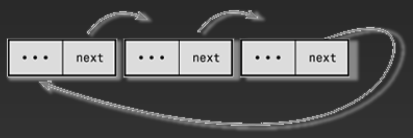


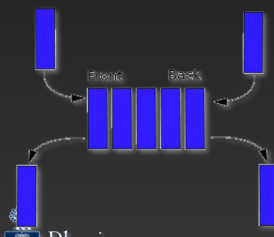


- FIFO

Other examples: double linked lists, circular lists, double-ended queues, ...







stack/queue – implementation, using the C++ STL

```cpp
template<class T, class C = deque<T> >
class std::stack {
protected:
C c;
public:
typedef typename C::value_type value_type;
typedef typename C::size_type size_type;
typedef C container_type;
explicit stack(const C& a = C()) : c(a){} // Inherit the constructor
bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
value_type& top() const { return c.back(); }
const value_type& top () const { return c.back(); }
void push(const value_type& n) { c.push_back(n); }
void pop() { c.pop_back(); }
};
```

```cpp
std::queue<myclass*> my_queue;


class myclass{
   string s;
};

myclass *p = new myclass();

my_queue.push(p);

// something ...

p = my_queue.front();
my_queue.pop();

std::cout << p->s;
```

The C++ Standard Template Library

generic collection of class templates and algorithms

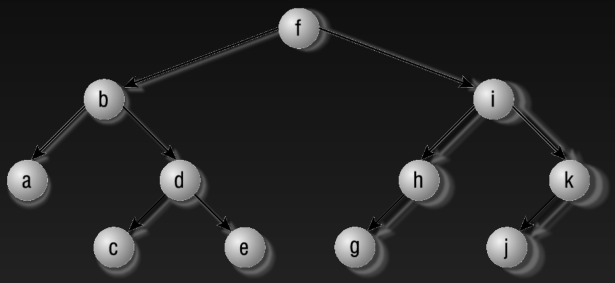
▼ Container class templates:

- `std::deque`, double-ended queue
- `std::forward_list`, singly linked list
- `std::list`, doubly linked list
- `std::map/multimap`, sorted associative array and multimap
- `std::queue`, single-ended queue (`std::priority_queue`)
- `std::stack`, stack
- `std::unordered_map/multimap`, unordered map/multimap, hash tables
- `std::vector`, resizable array

- `std::stack<int> myStack;`
- Functions (`stack`: `top`, `push`, `pop`, `size`)
- Algorithms
- Modifiers

Binary Trees

- abstract data structure, such that each node has at most two "children"



- *linked* abstract data structure composed of nodes
- each node contains a *field* and one or more *pointers* to other nodes
- recursive structures
- a linear list is trivially a *tree*
- graph theory

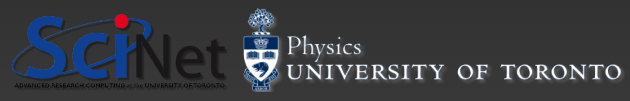
Example of a **tree** implementation...

```
#include <iostream>

using namespace std;

#define YES 1
#define NO 0

class tree
{
  private:
    struct leaf
    {
      int data;
      leaf *l;
      leaf *r;
    };
    struct leaf *p;

  public:
    tree();
    ~tree();
    void destruct(leaf *q);
    tree(tree& a);
    void findparent(int n,int &found,leaf* &parent);
    void findfordel(int n,int &found,leaf *&parent,leaf* &x);
    void add(int n);
    void transverse();
    void in(leaf *q);
    void pre(leaf *q);
    void post(leaf *q);
    void del(int n);
};
```

Example of a **tree** implementation...

```
#include <iostream>

using namespace std;

#define YES 1
#define NO 0

class tree
{
  private:
    struct leaf
    {
      int data;
      leaf *l;
      leaf *r;
    };
    struct leaf *p;

  public:
    tree();
    ~tree();
    void destruct(leaf *q);
    tree(tree& a);
    void findparent(int n,int &found,leaf* &parent);
    void findfordel(int n,int &found,leaf *&parent,leaf* &x);
    void add(int n);
    void transverse();
    void in(leaf *q);
    void pre(leaf *q);
    void post(leaf *q);
    void del(int n);
};
```

```
void tree::add(int n)
{
  int found;
  leaf *t,*parent;
  findparent(n,found,parent);
  if(found==YES)
    cout<<"\nSuch a Node Exists";
  else
  {
    t=new leaf;
    t->data=n;
    t->l=NULL;
    t->r=NULL;

    if(parent==NULL)
      p=t;
    else
      parent->data > n ? parent->l=t :
            parent->r=t;
  }
}
```



Example of a **tree** implementation...

```
#include <iostream>

using namespace std;

#define YES 1
#define NO 0

class tree
{
  private:
    struct leaf
    {
      int data;
      leaf *l;
      leaf *r;
    };
    struct leaf *p;

  public:
    tree();
    ~tree();
    void destruct(leaf *q);
    tree(tree& a);
    void findparent(int n,int &found,leaf* &parent);
    void findfordel(int n,int &found,leaf *&parent,leaf* &x);
    void add(int n);
    void transverse();
    void in(leaf *q);
    void pre(leaf *q);
    void post(leaf *q);
    void del(int n);
};
```

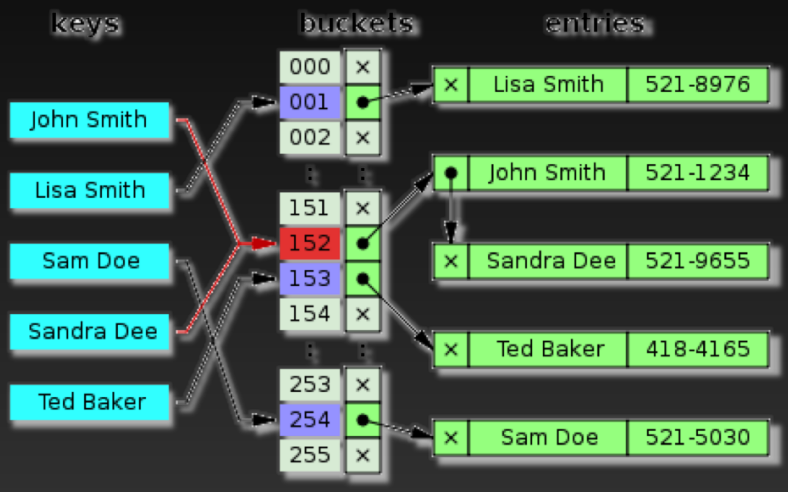
```
void tree::add(int n)
{
  int found;
  leaf *t,*parent;
  findparent(n,found,parent);
  if(found==YES)
    cout<<"\nSuch a Node Exists";
  else
  {
    t=new leaf;
    t->data=n;
    t->l=NULL;
    t->r=NULL;

    if(parent==NULL)
      p=t;
    else
      parent->data > n ? parent->l=t :
            parent->r=t;
  }
}
```

```
int main()
{
  tree t;
  int data[]={32,16,34,1,87,13,7,18,14,19,23,24,
  for(int iter=0 ; iter < 15 ; i++)
    t.add(data[iter]);

  t.transverse();
  t.del(16);
  t.transverse();
  t.del(41);
  t.transverse();
  return 0;
}
```

Hash Table (map)

- abstract data structure used to implement an associative array

  - map keys to values
  - uses a *hash function* to compute an index into an array of buckets or slots, from which the correct value can be found
  - aka dictionary or map



SciNet Physics UNIVERSITY OF TORONTO

Extra Material...

A bit more on rarrays

Amittedly, this is the lecture's own package.
But he's used it for a long time.
Its aims are:

- Header-only solution for dynamically allocated arrays for any rank
- Same syntax as builtin C++ arrays
- And just as fast (when compiled with optimization)
- No hidden copies
- Compatible with linear algebra packages BLAS and LAPACK

Rarray in a Nutshell

Define a n×m×k array of floats:	`rarray<float,3> b(n,m,k);`
Define it with preallocated memory:	`rarray<float,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()`
Extent in the ith dimension in b:	`b.extent(i)`
Shallow copy of the array:	`rarray<float,3> d=b;`
Deep copy of the array:	`rarray<float,3> e=b.copy();`
A rarray re-using an automatic array:	`float f[10][20][8]={...};`
.	`rarray<float,3> g=RARRAY(f);`
Output a rarray to screen:	`std::cout << h << endl;`
Read a rarray from keyboard:	`std::cin >> h;`

Returning a rarray from a function

No problem:

```
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
}
```

No hidden copies.

Optional Bounds checking

If the preprocessor constant `RA_BOUNDSCHECK` is defined, an out of bounds exception is thrown if

- an index is too small or too large;
- the size of dimension is requested that does not exist (in a call to `extent(int i)`);
- a constructor is called with a zero pointer for the buffer or for the dimensions array;
- a constructor is called with too few or too many arguments (for rank $<=$ 11).

`RA_BOUNDSCHECK` can be defined by adding the `-DRA_BOUNDSCHECK` argument to the compilation command, or by `#define RA_BOUNDSCHECK` before the `#include <rarray>` in the source.

I/O

- In the header rarrayio
- Only ascii for now: not great
- Prints it out as you would initialize an automatic array, i.e., with curly braces.
- Doing so means it can read it back in as well

Implementation of a map structure using C++ STL

```
std::map <key_type, data_type, [
    comparison_function]>
```

Implementation of a map structure using C++ STL

```
std::map <key_type, data_type, [
    comparison_function]>
```

```
std::map <string, char> grade_list;

grade_list["John"] = 'B';
// John's grade improves
grade_list["John"] = 'A';
```

Implementation of a map structure using C++ STL

```
std::map <key_type, data_type, [
    comparison_function]>
```

```
std::map <string, char> grade_list;

grade_list["John"] = 'B';
// John's grade improves
grade_list["John"] = 'A';
```

```
// functions assoc.with the map class
// int size(), bool empty((), clear()
grade_list.erase("John");

std::cout<<"The class is size "<<grade_list.size()<<std::endl;

grade_list.clear();
```

Implementation of a map structure using C++ STL

```
std::map <key_type, data_type, [
    comparison_function]>
```

```
std::map <string, char> grade_list;

grade_list["John"] = 'B';
// John's grade improves
grade_list["John"] = 'A';
```

```
// functions assoc.with the map class
// int size(), bool empty((), clear()
grade_list.erase("John");

std::cout<<"The class is size "<<grade_list.size()<<std::endl;

grade_list.clear();
```

```
std::map <string, char> grade_list;
grade_list["John"] = 'A';
if(grade_list.find("Tim") == grade_list.end())
{
    std::cout<<"Tim is not in the map!"<<endl;
}
```

Implementation of a map structure using C++ STL

```
std::map <key_type, data_type, [
    comparison_function]>
```

```
std::map <string, char> grade_list;

grade_list["John"] = 'B';
// John's grade improves
grade_list["John"] = 'A';
```

```
// functions assoc.with the map class
// int size(), bool empty((), clear()
grade_list.erase("John");

std::cout<<"The class is size "<<grade_list.size()<<std::endl;

grade_list.clear();
```

```
std::map <string, char> grade_list;
grade_list["John"] = 'A';
if(grade_list.find("Tim") == grade_list.end())
{
    std::cout<<"Tim is not in the map!"<<endl;
}
```

```
#include <iostream>
#include <map>
#include <string>
#include <utility>
using namespace std;

int main()
{
 typedef map<string, int> mapType;
 mapType populationMap;

 populationMap.insert(pair<string, int>("China", 1339))
      ;
 populationMap.insert(pair<string, int>("India", 1187))
      ;
 populationMap.insert(mapType::value_type("US", 310));
 populationMap.insert(mapType::value_type("Indonesia",
     234));
 populationMap.insert(make_pair("Brasil", 193));
 populationMap.insert(make_pair("Pakistan", 170));
```

Implementation of a map structure using C++ STL

```
std::map <key_type, data_type, [
    comparison_function]>
```

```
std::map <string, char> grade_list;

grade_list["John"] = 'B';
// John's grade improves
grade_list["John"] = 'A';
```

```
// functions assoc.with the map class
// int size(), bool empty((), clear()
grade_list.erase("John");

std::cout<<"The class is size "<<grade_list.size()<<std::endl;

grade_list.clear();
```

```
std::map <string, char> grade_list;
grade_list["John"] = 'A';
if(grade_list.find("Tim") == grade_list.end())
{
    std::cout<<"Tim is not in the map!"<<endl;
}
```

```
#include <iostream>
#include <map>
#include <string>
#include <utility>
using namespace std;

int main()
{
 typedef map<string, int> mapType;
 mapType populationMap;

 populationMap.insert(pair<string, int>("China", 1339))
      ;
 populationMap.insert(pair<string, int>("India", 1187))
      ;
 populationMap.insert(mapType::value_type("US", 310));
 populationMap.insert(mapType::value_type("Indonesia",
      234));
 populationMap.insert(make_pair("Brasil", 193));
 populationMap.insert(make_pair("Pakistan", 170));
```

```
mapType::iterator iter = --populationMap.end();
populationMap.erase(iter);

// output the size of the map
cout << "Size of populationMap: " << populationMap.size()
     << '\n';

for (iter = populationMap.begin(); iter != populationMap.
     end(); ++iter) {
 cout << iter->first << ": "
  << iter->second << " million\n";
}
```

```
string country("Indonesia");
iter = populationMap.find(country);
if( iter != populationMap.end() )
 cout << country <<"'s populations is "
  << iter->second << " million\n";
else
 cout << "Key is not in populationMap" << '\n';
```

```
 populationMap.clear();
}
```

Data Structures & Algorithms

	Insert	Delete	Balance	Get at index	Search	Find minimum	Find maximum	Space usage
Unsorted array	O(*n*)	O(*n*)	N/A	O(1)	O(*n*)	O(*n*)	O(*n*)	O(*n*)
Sorted array	O(*n*)	O(*n*)	N/A	O(1)	O(log *n*)	O(1)	O(1)	O(*n*)
Unsorted linked list	O(1)	O(1)	N/A	O(*n*)	O(*n*)	O(*n*)	O(*n*)	O(*n*)
Sorted linked list	O(1)	O(1)	N/A	O(*n*)	O(*n*)	O(1)	O(*n*)	O(*n*)
Self-balancing binary tree	O(1)	O(1)	O(log *n*)	N/A	O(log *n*)	O(log *n*)	O(log *n*)	O(*n*)
Heap	O(1)	O(1)	O(log *n*)	N/A	O(*n*)	O(1)	O(*n*)	O(*n*)
Hash table	O(1)	O(1)	O(*n*)	N/A	O(1)	O(*n*)	O(*n*)	O(*n*)
Trie	O(*m*)	O(*m*)	N/A	O(*n*)	O(*m*)	O(*m*)	O(*m*)	O(*n*)

	Linked list	Array	Dynamic array	Balanced tree	Random access list
Indexing	Θ(*n*)	Θ(1)	Θ(1)	Θ(log n)	Θ(log n)
Insert/delete at beginning	Θ(1)	N/A	Θ(*n*)	Θ(log n)	Θ(1)
Insert/delete at end	Θ(*n*) when last element is unknown; Θ(1) when last element is known	N/A	Θ(1) amortized	Θ(log *n*)	Θ(log *n*) updating
Insert/delete in middle	search time + Θ(1)[1][2][3]	N/A	Θ(*n*)	Θ(log *n*)	Θ(log *n*) updating
Wasted space (average)	Θ(*n*)	0	Θ(*n*)[4]	Θ(*n*)	Θ(*n*)

SRC: wikipedia