CS32: Data Structures + Algorithms

Objective

Data abstraction

C++ Classes

Pointers, Dynamic Arrays, Resource Management

Linked lists

Stacks and Queues

Inheritance and Polymorphism

OOP

Recursion

Templates, Iterators, STL

Algorithmic Efficiency

Sorting

Trees

Tree-based tables, Hash tables

Priority Queues, Heaps

Graphs

CS 31 Review

Pointers

- Another way to implement pass by reference
- Traverse arrays
- Manipulate dynamic storage
- Represent relationships in data structures

double	is a number
double &	is a reference to a double (another name for a pre-existing object)
double*	is a pointer to a double.

&x	generate a pointer to x
*p	the object that p points to

Reference parameters

→ used to make a function return multiple values

```
void polarToCartesian ( double rho, double theta,
                        double xx. double yy )

{
    xx = rho * cos(theta);
    yy = rho * sin(theta);
}

int main()
{
    double r;
    double angle;

    .. .. get r and angle ...

    double x;
    double y;

    polarToCartesian ( r, angle, x, y )
```

while x and y obtain the correct value, they poof when the function poofs.

What I'd like to do: change variable values OUTSIDE the function.

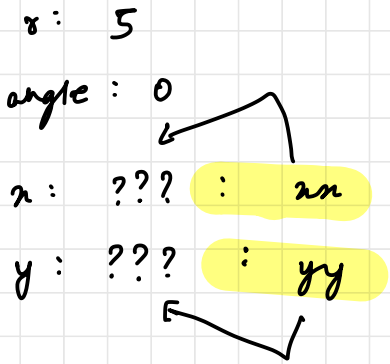
```
void polarToCartesian(double rho, double theta,
                      double& xx, double& yy)
```

double rho and double theta create brand new doubles that the values are copied into.

But double& xx and double& yy are references — essentially, they point to the same double as x/y.

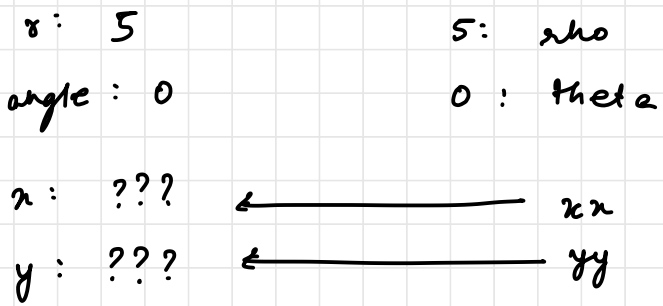
So if I change the value of x, even xx changes.

i.e. the parameter is just another name for the original argument. NO COPY IS MADE.



Upon ending the function, xx and yy go away - but they do their job, i.e. alter the values of x and y.

Pointers




```
void polarToCartesian ( double rho, double theta,
                        double* xx, double* yy )
{
    *xx = rho * cos(theta);
    *yy = rho * sin(theta);
}

int main()
{
    double r;
    double angle;

    .. .. get r and angle ...
```

```
    double x;
    double y;

    polarToCartesian(r, angle, &x, &y)
}
```

The pointer xx points to the main routine's x

" " yy " " " " y

Pointers are arrows pointing to something.

They don't actually hold a double, but tell you how to locate one.

To create an arrow / pointer to x, we append & before the variable name.

&x generates a pointer to x.

double * xx	→	data type for pointer
&x	→	pointer itself (address of x)
*p	→	the object that p points to (follow the pointer)

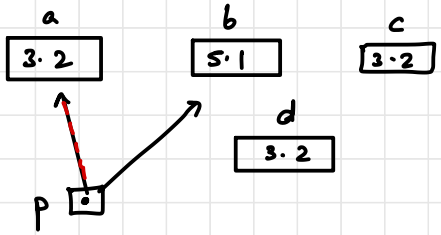
Mechanics of pointers

```
double a = 3.2;
double b = 5.1;

double* p = &a;

double c = a;

double d = *p;   // Now p points to b
```




```
p = &b;
```

make p point to b.

```
*p = b;
```

take b and store it where p points to.

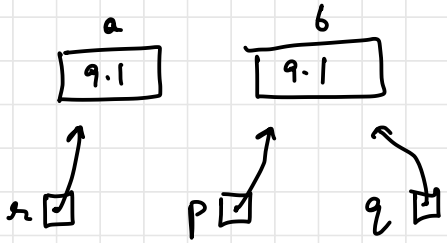
```
int k = 2;
p = &k;   // different type, cannot happen.
```

pointer types cannot be converted.

Cannot multiply a pointer with any data type, obviously. cannot perform numerical operations.

Following an uninitialized pointer throws an error. Randomly replaces some bit pattern in/out your system might point anywhere and can be potentially very dangerous.

When we compare two pointers, we compare the address. While the value of the pointer might be the same, the address might not.



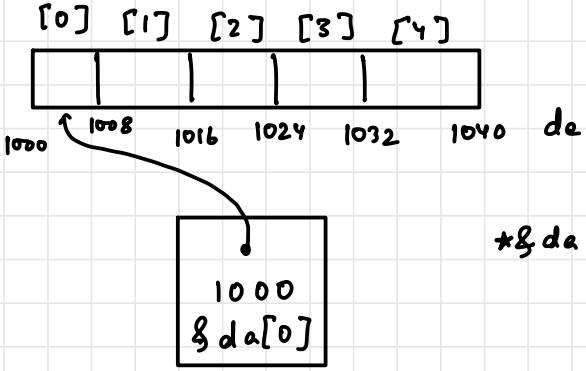

```
(p == r)        // false
 ↓   ↳ &a
&b

(p == q)        // true

(*p == *r)      // true
```

Pointers to traverse arrays

```
for (double *dp = &da[0]; dp < &da[5]; dp++)
    *dp = 3.6;
```



*&da[0] = da[0].

Algebra rules

→ &a[i] ± j = &a[i ± j]

Integer + pointer = pointer [sub + integer]

→ C++ standard allows position 1040 (just pass the end) but that pointer cannot be followed

→ &a[i] < &a[j] ⇒ i < j

>
<=
>=
==
!=

These compare positions in an array

→ In an expression, writing the name of an array without the sub is a pointer to element 0.

a ⟷ &a[0]

&da[5]
&da[0 + 5]
&da[0] + 5
a + 5

Template to traverse an array

```
for (double * dp = da; dp < da + MAXSIZE; dp++)
```

→ double b[] is practically equivalent to double* b.

→ p[i] ⟷ *(p + i) because p[] is a pointer to the beginning of the array.

→ well-defined pointer that does not point to any object → nullptr. Used for error.

NULLPTR

1. `int * ip = nullptr;`
2. `ip = nullptr;`
3. `if (ip == / != nullptr)`

```
...
```

Undefined :	*ip = 42	(if ip = nullptr)

An integer constant 0 in a context where a pointer is required is a nullptr.

Structs

Define your own datatype. Instead of having multiple corresponding arrays, use structs.

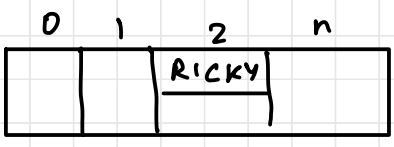
> Structs : common convention
> → start with a capital letter

```
struct Employ
{
    string name;
    double salary;
    int age;
};   ← important trailing semicolon

int main()
{
    Employee e1;

    e1.name = "Fred"
    e1.salary = 50000
    e1.age = 51;

    Employee company[100]
    company[2].name = "Ricky"
}
```



Passing to functions : structs

use objectOfStructType . nameOfMemberOfStructType to use attributes of the struct.

```
void printPayCheck (const Employee& e)
{
  cout << e.name << endl;
}
```

guarantee no change to e

Based on size of a struct,

large size - pass by reference (cheap)

small size - pass by value (expensive)

```
void f(Blah b);
 or
void f (const Blah b);
```

To change a value, obviously use pass by reference or pointers.

Pointer approach

`*ep.name` ← dot operator takes precedence

`(*ep).name` ← ugly

`ep->name` ← solution

a pointer of an object of some struct → the name of a member of the struct

Member Functions

Necessary to impose constraints within a struct.

```
struct Target
{
    void init();
    bool move(char dir);
    void replayHistory();

    int pos;
    string history;
};

void Target::init()
{
    this->pos = 0;
    this->history = "";
}
```

member functions

data members

:: implies a member function

'this' is used by an object to access itself

The `this` arrow can be left off since the usage is implicit.

```
bool Target::move(char dir)
{
    switch(dir)
    {   case 'R':
        case 'r':
            pos++;
            break;
        case 'L':
        case 'l':
            pos--;
            break;
        default:
            return false;
    }
    history += toupper(dir);
    return true;
}
```

Access modifiers - enforcement mechanism

public / private - impose your own functions

If I want my private variables to be used only by the public functions and not otherwise, I use access modifiers.

```
struct Target
{ public:
    void init();
    bool move(char dir);
    void replayHistory();
  private:
    int pos;
    string history;
}
```

Public can be accessed by other classes and functions too.

Private can only be accessed by member functions inside the class, or indirectly by using public member functions.

Private functions cannot even be viewed outside the public functions. Allows changing internal implementation without changing external operation.

Simply add a public

```
int Target::position()
{
    return pos;
}
```

Common convention while naming private elements is to use a common prefix [to identify data members]

For eg.

```
private:
    int    m_health;
    string m_name;
```

Name of some struct :: Name of a member of that type

If not specifically listed, all data members are public. This is due to C integration.

STRUCT vs. **CLASS**

- Starts off public if not specified
- Starts off private if not specified

Otherwise, they are literally the same.

To be safe, list public and private members independently and clearly → use interchangeably.

const member functions

C++ does not allow, for eg.

```
void report (const Target& x)
{ cout << x.position() << endl; }
```

because this might change values, and the compiler cannot see the implementation of the function

Therefore, the correct approach is:

```
int position() const;
```

which promises not to modify the object it is called on.

const also needs to be appended to :: implementation of a member function.

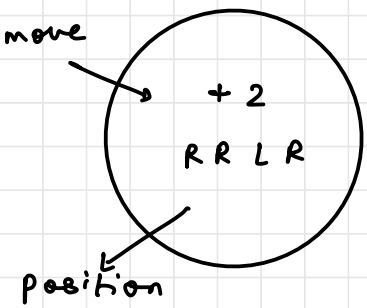
We need to make sure that the Target is initialized immediately. If some functions are called without initializing, that can throw errors.

We use a constructor for this. It is a special member function that is automatically called everytime an object is created.

```
class Target
{ public:
    Target();          // constructor
                       // no return type
}
```

```
Target :: Target ()
{
    pos = 0;
    history = " "
}
```

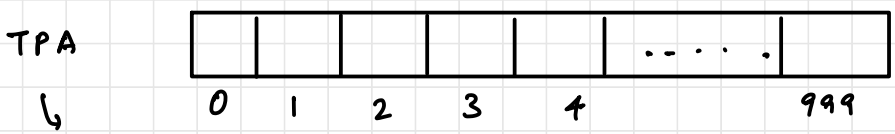
// replaces init
// initializes Target
// constructed immediately



flow of information

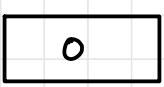
Pointers and Dynamic storage

It doesn't make sense to initialize a Target till it is required. Initializing a 1000 targets when only 100 are needed would be a waste of processing. Therefore, we use pointers as such:



target pointer array

nTargets



# of targets being initially used

```
Target *  tpa [1000];
int  nTargets  =  0;
```

Syntax to construct a brand new target and give me a pointer assigned to a new array position:

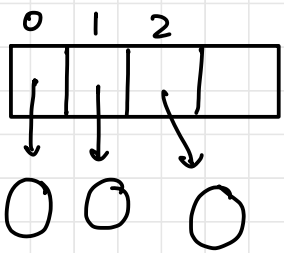
```
tpa [nTargets] = new Target;
                 └─┘
                 built into C++
                 └──────────┘
                 calls constructor
└──────────────────────────────┘
assigns a new pointer to
tpa [nTargets]
nTargets ++ ;
```

```
tpa [1] → move ('R');   // since this is a pointer we use →
// I could potentially use (*tpa[1]).move('R') but verbose
```

Pointers - Deleting a target

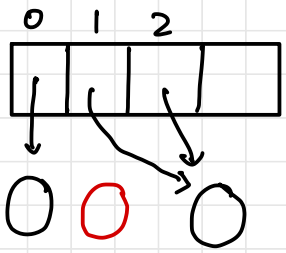
Theoretically, I can:

```
tpa [1] = tpa [2];
nTargets --
```



thereby reassigning [2] to [1] and deleting what was present in [1] initially. But this would involve two pointers pointing to the

same object, albeit briefly. However, I would NOT be looking at the second pointer, since



my nTargets would not include the second pointer element in the array.

A better way of looking at this would be:

```
tpa[1] = tpa[2];
nTargets--
tpa[2] = nullptr; // unnecessary
```

It is necessary to return storage to the operating system upon deletion of a certain element.

Here, for instance, the red blurb is inaccesible, because the pointer pointing to it is lost. So we call it garbage.

A program with too much garbage leads to a **memory leak.** Memory leaks usually do not show up unless the program runtime is high.

Deleting the object that a pointer points to results in a **dangling pointer.** This may point to the previously stored information, or may point to new information. Make sure to reassign the pointer.

Deleting the object that a pointer points to only makes sense if New is used. Otherwise, it causes an error.

In dynamic allocation, deleting a pointer deletes the object that the pointer points to

```
p = new Target;
delete p;
```

// deletes the new target

Double delete is also a common error. Avoid deleting deleted memory.

```
int k;
int * p = &k;
delete p;   // error, p cannot delete a local variable
```

- Named local variables live on 'the stack'. They go away upon deletion.
- Variables declared outside any function live in the global storage area / static storage area
- Dynamically allocated objects live on 'the heap.' They need to be deleted after usage.

> The array of pointers lives on the stack. Upon leaving the function, they get deleted. This results in dynamically allocated objects becoming garbage.

Therefore, it is essential that we delete dynamically allocated objects.

```
for (int k = 0; k < nTargets; k++)
    delete tpa[k];
```

To initiate a class member with set attributes, give constructors arguments.

```
public:
  Pet (string nm, int initialHealth);

int main
{
    Pet p1("Tommy", 10);
    Pet p2;              // Error
}
```

Constructor Caveat

```
Blah b3(30, 20, 10);
Pet  p2("Tommy", 20);
Sleep s1(20);
```

```
Target t();   // error: this is a function
Target t;     // Ditch parantheses, works.
```

Lecture 1 Part 1 – January 3

Undefined Behaviour

a[k] where k is out of bounds.

i/j where j = 0.

p→ [element undefined]

Uninitialized variable assumed to be 0.

Implementation-dependent behaviour

17/-5	17%-5
-3 -4	2 -3

If there's no return statement in a function, (except void), it is undefined behaviour.

```
double f
{
    if ——
      return 0;
    if ——
      return 1;
}
```

probably will not give an error, although it will give a warning. Ensure return in the general function code.

int	-2 billion	to	2 billion
unsigned int (size_t)	0	to	4 billion

0 - 2 billion ⟶ treated the same way

```
for (int k = 0;  k < sting.size() - 1;  k++)
                    ↓                   ↓
                 unsigned            signed
                [size_t]
```

an expression containing a signed and an unsigned int always converts the signed to unsigned.

Therefore, if string.size = 0, the value returned is 4 billion.

```
for (int k = 0; k+1 < s.size(); k++)
```

↳ fixed version

Alternatively,

```
int ssize = sting.size()
```

CONVERT BOTH POTENTIAL CASES TO INT AND ACT UPON THAT.

Lecture 1 Part 2 — January 3