

CS 32: 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 *f* 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 x;
 double angle;

 ... get r and angle ...

 double x;
 double y;

 polarToCartesian (r, angle, x, y)

while x and y obtain the correct value,
they pass when the function passes.

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.

x: 5

angle: 0

x: ??? : xx

y: ??? : yy

5: rho

0: theta

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

Pointers

$r : 5$

angle : 0

$x : ???$

$y : ???$

$5 : \text{rho}$

0 : theta

$\xleftarrow{\hspace{1cm}} xx$

$\xleftarrow{\hspace{1cm}} yy$

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 `&x` 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)

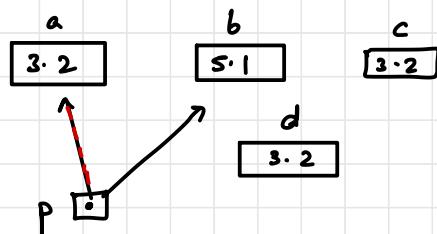
Mechanics of pointers

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

```
double* p = &a;
```

```
double c = a;
```

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



$$p = b$$

$p = \&b;$

make p point to b.

$*p = b;$

take b and store it
where p points to.

```
int k = 2;
```

$p = \&k;$

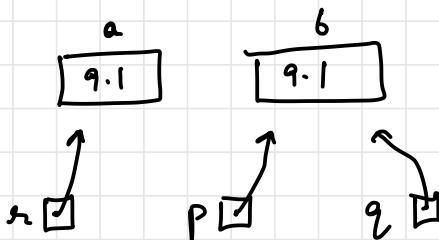
// different type, cannot happen.

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.



$(p == s)$ // false

\downarrow
8.2

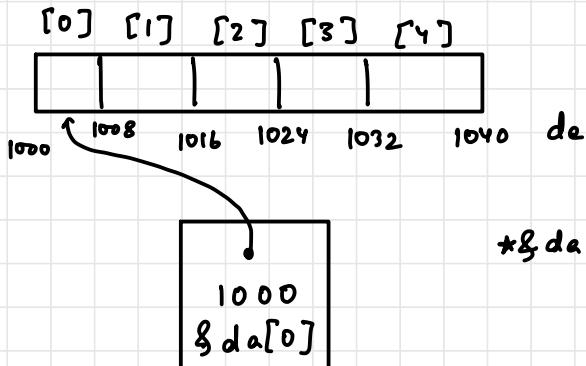
$(p == q)$ // true

$(\star p == \star s)$ // true

Pointers to traverse arrays

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

```
*dp = 3.6;
```



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

Algebra rules

$$\rightarrow \&a[i] \pm j = \&a[i \pm j]$$

Integer + pointer = pointer [sub + integer]

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

$$\rightarrow \&a[i] < \&a[j] \Rightarrow i < j$$

>

<=

>=

= =

!=

These compare positions in an array

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

$$a \leftrightarrow \&a[0]$$

$$\&da[5]$$

$$\&da[0 + 5]$$

$$\&da[0] + 5$$

$$a + 5$$

Template to traverse an array

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

→ $\text{double } b[]$ is practically equivalent to $\text{double* } b$.

→ $p[i] \leftrightarrow *(p + i)$ 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 Employee
{
    string name;
    double salary;
    int age;
};

int main ()
{
    Employee e1;

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

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

0	1	2	n
		RICKY	

Passing to functions : structs

use object of struct type . name of member of struct type
to use attributes of the struct.

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

guarantees 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 ← det 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();

member functions

int pos;
string history;

data members

}

void Target:: init()

] :: implies a member function

{

this → pos = 0;
this → history = ""

] '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;
```

```
}
```

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 then create.

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
```

Undefined Behaviour

$a[k]$ where k is out of bounds.

i/j where $j = 0$.

$p \rightarrow$ [element undefined]

Uninitialized variable assumed to be 0.

Implementation-dependent behaviour

17/-5	17% -5
-3	2
-4	-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 < string.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 = string.size()

CONVERT BOTH POTENTIAL CASES
TO INT AND ACT UPON THAT.

Lecture 1 Part 2

- January 3