Introduction to C++

INTRO

- C++ files end in .cpp
- C++ is a compiled language

compile: `g++ file.cpp`

compile + name: `g++ file.cpp -o execname`

run: `./execname`

- typical comments: `// single line`
  `/* multi line */`

basic setup:

```
#include <iostream>        // required line
int main() {               // main() req.
    std::cout << "Hello world";   // print statement
    return 0;              // return must match func declaration
}
```

VARIABLES

- Support for: int, double, char, string, bool
- Vars must be declared before used

declare + initialize: `<type> <name> = <value>;`

ints	4 bytes	$2^{-31} \cdot 2^{31}$
doubles	8 bytes	can store 15 decimal places
char	1 byte	
bool	1 byte	

(type) value — casting

bool: `std::cout << bool_var` outputs 0 or 1

chaining couts

```
std::cout << "I am " << ageYear << "years old";
```

user input

```
int myName = "unknown";
std::cin >> myName;
```

`>>` ← assign to this variable

`std::cin` ← get input

`>>` ← flipped from cout

constants

- cannot be changed after assignment

```
const pi = 3.14;
```

type conversion

```
(type) value
int newInt = (int) 3.000;
```

- converts value to type
- converting to int truncates not rounds

CONDITIONALS

```
if (<condition>) {
    // code
} else if (<other condition>) {
    // more code
} else {
    // more more code
}
```

Pre increments: `++x`

↳ decrements AND modifies value of x

Post Increment: `x++` ← copy is made, org. incremented, copy returned

```
int x = 10
int y = x--;
std::cout << x << ' ' << y
// 9 10
```

```
int x = 10
int y = --x;
// 9 9
```

Bitwise Operators:

- `x << y` shift bits in x left by y
- `x >> y` shift bits in x right by y
- `~x` flip all bits in x
- `x & y` flip all bits in x : y
- `x | y` flip even bit in x or y
- `x ^ y` XOR even bit in x to y

switch

```
switch (<variable>) {
    case <value> :
        //code
        break;
    case <val> :
        // code
        break;
    default :
        break;
```

w/o breaks every code block after match will execute

logical ops.

- && = and
- || = or
- ! = not

both words and symbols work

Loops

while loops

```
while (condition) {
        //code
}
```

for loops

```
for (int i = 0; i <10; i++) {
        //code
}
```

semicolons, not commas

ERRORS

- **Compile time Errors** = found by compiler
  - **Syntax errors**
  - **Type errors** = mismatch between types declared
- **Link time Errors** = trouble locating function or library found by linker when combining files into executable (i.e. using a func that was never defined)
- **Run time Errors** = occur during runtime : cause crashes (i.e. divide by 0, open a file that doesn't exist)
- **Logic Errors** = errors in logic of program causing unexpected results

VECTORS

- vector = sequence of elements that can be accessed by index
- vectors are **mutable**

declare vector

```
#include <vector>          // must include library
std::vector<type> nums = { 1, 2, 3, 4 };
std::vector<type> name(2); // set initial size w/o init.
```

- type cannot be changed after declaration
- access indicies like normal: `vectorName[index];`

add element

```
name.push_back(newEl);
```

remove element

- pop has no return value in C++

```
name.pop_back();
```

get size	name.size(); **ARRAYS** • arrays are inherited from C and are immutable (in size)
declare array	type name[size]; type name[ ] = [values];

FUNCTIONS	• to return string or vector: std::string, std::vector
	return_type name (params) {
	//code
	}

headers	• sometimes functions are declared above main : defined in seperate files returntype name(params); main () ... if so when compiling remember to link files together g++ main.cpp funcs.cpp • OR create a header file w/ the declarations and include that in main... # include "headername.hpp"
inline functions	• tells compiler to insert function body where function call is. sometimes speeds up execution. inline ← goes above normal declaration rettype name ...

default arguments

```
retType name(int par1, int par2 = 0) { }
```

overloading

- Like Java, functions can be overloaded by defining multiple functions w/ the same name: retType but different parameters

templates

- more efficient than overloading, similar to generics

```
template <typename T>
void myFunc(T item) {
    std::cout << item;
}
```

- slows compile time but speeds up execution
- defined in header file

CLASSES

```
class ClassName {
    type varName;
public:
    ClassName();
    otherMethod();
private:
    privateMethod();
};
```

- access control operators apply to members under them
- `ClassName();` ← constructor declaration
- actual method definitions in className.cpp
- notice semicolon

destructor

- special method handling object destruction ⋮ preventing memory leaks

```
ClassName::~ClassName(){
    // clean up
}
```

REFERENCES

- reference variables are aliases

```
int &refName = orgName;
```

- anything that happens to the reference also happens to the original
- aliases cannot be changed to alias something else

Pass by Variable

- usually normal variables are passed into functions; the function cannot directly modify the variable, just a copy of its value.

Pass by Reference

- when references are passed in to functions the actual variable value can be directly modified
- This is set up in definition:

```
void swap(int &num1, int &num2){
    int temp = i;
    i = j;
    j = temp;
}
```

- If writing a function that shouldn't and doesn't modify variable use const

```
int triple(int const &i) {
    return i * 3;
}
```

saves computing cost of creating a copy, which is unnecessary since no modification anyway

POINTERS

memory address

- `&` outside of a declaration is an address operator, not a reference op
  i.e. `std::cout << &varName` prints the memory address of the var

pointer vars

- pointer vars store memory addresses

```
int* numberPointer;
```
← pointer to an int

create pointer to var

```
int* ptr = &varName;
```
← memory address of var

reference

- contains value pointed to

```
int unpacking = *ptr;
```
← dereference outside of declaration

null pointer

```
int* ptr;
```
BAD! contains address of "somewhere" and is dangerous & causes bugs

To create an empty pointer use:

```
int* ptr = nullptr;
```

MEMORY ALLOCATION

- local vars go out of scope when function ends, but free store can be used for storing longer living variables in memory.

manual memory management

- dynamically allocate memory w/ new : delete operators:

```
ptr = new int[num];
delete ptr;
```

- whatever is created must also be deleted to prevent memory leak

Rule of 5

- if class defines 1 it should have all:

  Destructor, copy constructor, copy assignment operator, move constructor, move assignment op.

Smart pointers

- `unique_ptr` : manages object and deletes it when unique_ptr goes out of scope
- `shared_ptr`: retains shared ownership of object through pointer

DATA STRUCTURES IN C++

Data Structures	ways to store and organize data
Abstract Data Types (ADT)	define data and operations but have no implementation
Implementations	concrete manifestations : uses of data types

LINKED LISTS

Abstract Static List

- Collection of objects of the same type
- Store given # of elements of given data type
- write / modify / read el at position
- Arrays are an implementation

Abstract Dynamic List

- empty list has size 0
- insert / remove / count els
- read / write to pre-existing creation
- Specify a data l
- Access / modify in constant time $O(1)$
- insertion / remove at index $O(N)$
- Adding el that fills array - $O(N)$

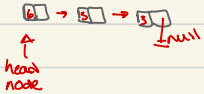
} not very efficient when implemented on top of arrays

Linked Lists

- Two parts to each el - the content and the address of the next index. ← pointer
  - ↳ ⓪ indicates end of list
  - ↳ These are **Nodes**

```
Struct Node {
    int data;
    Node* next;
};
```

Ex

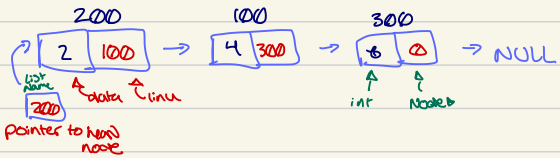


- **Access El : O(N)** **Insertion : O(N)**

Arrays vs. Linked Lists

	Arrays	Linked List
Access	O(1) Know starting #, so just add	O(N) must start at head node and go through every node
Memory	Fixed Requires large block	Dynamic No unused memory, BUT uses extra memory for pointers Can use small blocks
Insertion /Removal	at beginning : O(N)	O(1) - just make new head node
	at end : O(1) if not full	O(N) - traverse to find end, add
	: O(N) if full	
	at "i" : O(N)	O(N)
Ease	easy	hard - memory leakage, seg faults

Implementation




```
Struct Node {
    int data;
    Node* link;
}
```

```
Node* A = NULL;
Node* temp = new Node()
(*temp).data = 2;
(*temp).link = NULL;
A = temp;
```

Insert at beginning

```
void Insert(int x) {
    Node* temp = new Node();
    temp->data = x
    temp->next = head;
    head = temp;
}
```