## Variable Identifiers/Types

Identifiers can only begin with an alpha/underscore and contain both and numbers

double x = 5;    double x(5)    doublex{5}

static_case<type> (variable) → does not change variable

Built-in types are garbage value by default

Boolean: true(1)    false(0)

(Boolean expression) ? n1 : n2;

If true, return n1    If false, return n2

If (a[i] = a[i+1]) -> returns a[i] after execution

## Precision

cout.setf(ios::fixed);

cout.precision(2);

## Arithmetic Operators

'!=' not equal to, '==' equal to

&& - and  || - or  '%'- modulus

If both are double, result is double

If both are int, result is int

If one is double, result is double

1/3 → 0

14% 5 = 4    14/5 = 2

x++ → return x then increment

++x → increment than return x

## While Loop

```
while (Boolean expression) {
        body;
        increment;
}
```

## If Statement

```
if (Boolean expression) {
        body;
else
        body;
}
```

## Do While //executes body at least once

```
do
{
        body;
} while (Boolean);
```

## For Loop

```
for(initialization; expression; increment;) {
        body;
}
```

Break statement: breaks out of nearest enclosing loop

Continue abandons current iteration and goes to next one

## Switch Statement

```
switch (variable) {
        case type:
                break;
        case type1:
                break;
        default:
                break;
}
```

Variables in switch statements can only be int, double or char

If you do not use a break between cases, it will run until it finds a break

## Scope

```
Int sum;
for (int k = 0; k < 2; k++)
        sum += 1;
```

You cannot access k after you leave the for loop

Variables declared outside a function do not exist inside function, and vice versa

Look at scope block-by-block

## Functions

return type identifier(parameters); → prototype

```
return type identifier(parameter) { → definition
        body;
}
```

Can have zero parameters → void () returns no value

- return; terminates without returning a value

A parameter with const cannot be modified within a function

When calling a function, you only pass in variables, not types

## Strings

Input: getline(cin, answer);

Use cin.ignore(1000, '\n'); when inputting a number then a string

Strings are empty by default

Int s = "COMP";

s + "ILE" = "COMPILE"    s.length() = 2;    s[2] = 'M'

s.substr(0, 3) = "COM"    s.substr(1) = "OMP"

**Integer-Character conversion**

'0' + integer, will give you desired numerical character

Reversing a string

```
for (int i = 0; i < s.length()/2; i++){
    char temp = s[i];
    s[i] = s[s.length() - 1 - i];
    s[s.length() - 1 - i] = temp;
}
```

Comparison: AB < AC    AB < yz    AC < ABE

## <CCTYPE> Functions

isdigit, isupper, toupper, tolower, isalpha → only take char

## Arrays

Valid declarations: <type><name>[size];

int arr[10];    int arr[] = {1,2};

const int MAXSIZE = 2;   string s[MAX_SIZE];

- Passed by reference sort-of (arrays are pointers)
- Size of the array should be passed to the function, call to function only passes in array name
- Size MUST be specified if not initialized
- Int a[3] = {1,2,3,4,5} //**ERROR** → int a[3] = {1,2} // **OK**

Multidimensional arrays specific size of second dimension

You can make arrays const so you don't change the elements

String arrays are initialized to empty string

## C Strings

A sequence of zero or more characters

char a[] = "Hello"    char a[] = {'H', 'e', 'l', 'l', 'o', '\0'};

cout << a + 1; → "ello"    cout << a + 3; → "lo"

Always remember null byte, especially in size and functions
char[5] = “Hi” == char[5] = {‘H’, ‘i’, ‘\0’, ‘\0’, ‘\0’}
Only prints c string up to the null byte

**Functions**

strlen(s)     strcpy(destination, source) **SIZE CHECK adds** ‘\0
strcat(append to, append) **SIZE CHECK**
strcmp(first, second)

- If first > second return (>0)
- If first == second return (0)
- If first < second return (<0)

char s[10]; - can initialize as empty string
s = “abcdefg”; //**ERROR use strcpy**
s[1] = ‘t’; // **OK**
char s[3][3] = {“Hi”,“Yo”,“Be”}// s[rows][columns]
s[2]= “Be”     s[2][0] = ‘B’     s[1][1] = ‘o’

Converting C string into C++ string

```
char cs[10] = "Hello";
string cpps;
cpps = cs;
```

Converting C++ string into C string

```
string cpps = "Hello";
strcpy(cs, cpps.c_str());
```

## Pointers

Pointers store the addresses of variables in memory
double* p = &x; “pointer to a double”
*p = 5.0 → dereferencing the pointer

- Arrays are pointers that point to first element
- Array + 1 will point to second element

cout << arr will print the same as cout << &a[0]
&da[1] = dp++ = dp + 1
*(array + 1) == a[1]     &a[i] = &a[j] = i-j
int Find(const string a[]) == int Find(const string* a)
Another way of passing by reference
If int = k, double* p = &k //**ERROR not same type**
double* q;   *q = 5; //**ERROR we don’t know what q points to**
&da[0 + 1] == &da[1] == da + 1 == da++     &a[i] – j == &a[i-j]
double arr[] = {1,2,3,4};     double* p = arr;     p =&a[0]

- If a function parameter is (int* p) you can pass in (&b);
- Deleting a NULL ptr is undefined behavior, deleting a nullptr is well-defined behavior
- After deleting an object a pointer points to, you do not have to set it to nullptr if the pointer is a local variable

const ptr = const ptr  const ptr = ptr   ptr = const ptr //**ERROR**

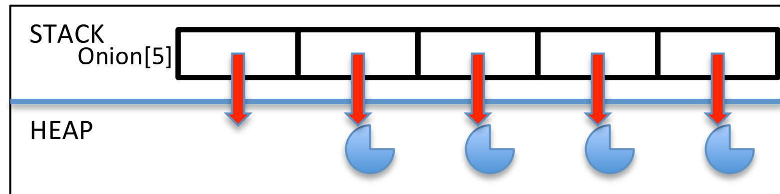
- You can do constptr++, but you cannot modify what pointer points to

**Pointers in a for loop**

```
char chaArray[] = "12345";
for (int k = 0; *(chaArray + k) != '\0'; k++)
   cout << chaArray[k] << endl; // 1 2 3 4 5
for (char* cP = chaArray; *cP != '\0'; cP++)
     cout << "*cP = " << *cP << endl; // 1 2 3 4 5
for (char* cP = chaArray; cP < chaArray + 5; cP++)
     cout << "cP = " << cP << endl;
  //12345 //2345 //345 //45 //5
```

## Dynamic Allocation

Remember when you could not create an array w/o knowing size?
Int len = 100;
double *arr = **new** double[len]; → array of double, arr points to first element
The place dedicated for dynamic memory is called heap
Memory leak: when we lose pointers to objects and then have no way of accessing the object, so delete unused vars




```
int *p = new int;
delete p;
```

For each **new** statement, there should be a **delete** statement
Variables can only be accessed by pointers, pointer is on the left hand side

```
*m_fish[m_nFish] = new Goldfish(capacity); or Goldfish();
<type>* name = new <type>(parameters)
```

Memory is freed only after delete is used
Do not delete something that has been deleted or is not dynamically allocated

```
for (int i = 0; I < 5; i++)
        onion[i] = new dullExample();
```

**Five pointers that point to five dynamically allocated objects**

```
for (int i =0; i < 5; i++)
        delete onion[i];          or delete *(onion + i)
```

**Delete all five dynamically allocated objects**

```
dullExample::dullExample () {
      evenDuller = new superDull(); // new object
  }
int main () {
  dullExample* faucet = new dullExample();
  cout << faucet->evenDuller->lame << endl;
  delete faucet; //ERROR
}
```

Delete faucet & dullExample,→dullExample dynamically allocates an object, **ONLY** if you do not have a destructor

## Structures

```
struct Student {
        //Member variables, data members
        int age;
        string name;
        //Member functions
        void setAge(int n); //Mutator
        string getName(); //Accessor
};
int main() {
        Student k; //k is an object of type Student
        k.name = “Fred”;
        k.age = 5;
```

k.age++ //age now equal 6
}

Syntax: an object of some type **.** the name of the member

a pointer to an object of some struct/class type **->** the name of a member of that struct/class type

- By definition -> means the same as *

Inside the member function's parameters there is something called "this" a pointer to whatever you are working on.

Specify object when using functions

- void Target::move(char dir) - :: scope operator
- The constructor will be passed to a pointer to the object that is being constructed, responsible for initialization

Strings are initialized to empty, built-in types are garbage

Data members can be:

Private

- only accessed by member functions
- insures that user does not mess up program
- Called encapsulation

Public

- can be called and used by user
- You cannot have two members of a class with the same name (variable and function)
- You can overload a function by using the same name, but different parameters (pointer and arr or char arr are same type)

Big objects are usually passed by reference

***The only difference between struct and class is that class members are inherently private if not declared, and struct are public

**Constructor initializes : Student();

- Can be multiple constructors, with different parameters

**Destructor deletes objects in memory: ~Student();

If you want to pass a const parameter to a function, make sure any functions that are called inside are constant

void Target::move(char dir) const { }

Student A[5]: array of five Student objects

A[2].name = "Fred"; the third student object is named Fred

Student *p p->name; p points to first member in struct

- Accessors are implemented as const functions

Local variables ("automatic) live on the stack, dynamic live in the heap, the newest local variable is on the top of the stack

Constructor with no arguments is called zero-argument constructor

- Compiler will write default constructor if you do not

**DO NOT** try and follow a nullptr or delete an already deleted element

## Classes

A construct used to group related fields (var) and methods

- Every instance of a class has its own members

When defining functions outside of class definition

return type Class/-name::function_name(argument_list)

Cat p1; // creates a Cat instance using default constructor

Constructors can be called within a constructor

Initialization list

- Organizes initialization statement

```
Cat:: Cat()
: m_age(0), m_weight(0), m_gender(1)
{
    Body of code;
} obj1, obj2; //declaring objects right away class definition
Cat *pKitty = new Cat();
pKitty->meow();
```

**Destructor** ~Cat();

- No parameters, no return type
- Cannot be manually called

**Called when local variable falls out of scope or delete is used**

There can only be one destructor

```
~SuperInterestingExample () {
  cout << "[!] Destructor called!" << endl;
  delete ob;
}
int main () {
  // NOTICE: sup is now a dynamically allocated var
SuperInterestingExample* sup = new SuperInterestingExample()
cout << sup->ob->actuallyNotThatInteresting << endl;
  delete sup; //Calls destructor to delete all member variables
}
```

The compiler calls the constructor when you initialize a new object, you do not have to call it

pKitty->Cat(); //**ERROR**

## This Pointer

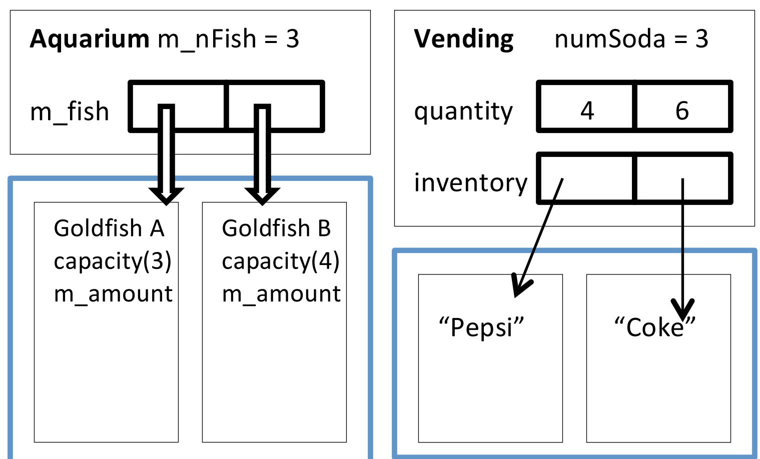
- This is a pointer to an object
- This holds the address of the object

When the parameter name is the same as private variable name, you have to use this pointer

```
private:
string name; //Member variable
void Soda::setName(string name) {
  this->name = name;
}
```

## Precedence

**(::)** (., ->, ++a) (++a, !, **, &, new, delete)** **(%, /,*(x) )** (+, -)



```
class Goldfish {
public:
   Goldfish(int capacity);
   ~Goldfish();
   void remember(char c);
   void forget();
   void printMemory() const; // Prints memory
private:
   char *m_memory; // Pointer to memory.
   int m_amount; // # of chars remembered.
   int m_capacity; // # of chars this fish can remember.
};
int MAX_FISH = 20;
class Aquarium {
public:
   Aquarium();
   bool addFish(int capacity);
   Goldfish *getFish(int n);
   void oracle();
   ~Aquarium();
private:
   Goldfish *m_fish[MAX_FISH];  // Pointers to fish.
   int m_nFish;                 // Number of Fish.
};
Aquarium::Aquarium() {
   m_nFish = 0;
}
bool Aquarium::addFish(int capacity) {
      m_fish[m_nFish] = new Goldfish(capacity);
      m_nFish++;
      return true;
}
Aquarium::~Aquarium() {
   for (int i = 0; i < m_nFish; i++)
      delete m_fish[i];
}
void Aquarium::oracle() {
   for (int i = 0; i < m_nFish; i++)
      m_fish[i]->printMemory();
      m_fish[i]->forget();
}
```

If Aquarium declares a Goldfish object, but does not give it a capacity, it will not compile, since there is no default constructor for Goldfish. You have to construct Goldfish, before Aquarium constructs.

```
Aquarium() : Bob(10) { }
private:
   Goldfish Bob;
```

Constructing: Inner → Outer (Goldfish, then Aquarium)

Destructor: Outer → Inner (Aquarium, then Goldfish)

**Reference**

int& n – pass-by reference: changes made to n inside function will remain outside the function, access variable outside

```
   int a = 2;
   int &b = a; // a = 2, b = 2
   a = 4; // a = 4, b = 4
```

Passing a pointer by reference (int* &p)

```
class Soda {
   public:
      Soda();
      void setName(string name); string getName() const;
   private:
      string name;
};
void Soda::setName(string name) {
      this->name = name; // We have to use this->name here.
}
class VM {
public:
      VM(int n);
      ~VM();
      void restock(string name,int quantity);
       Soda* getSoda(string name);
       bool buySoda(string name);
private:
   Soda* inventory[MAXSODA]; int quantity[MAXSODA];
   int numSoda;
};
VM::VM(int n) {
   numSoda = n;
   for(int i=0;i<numSoda;i++)
      inventory[ i ] = new Soda();
}
void VM::restock(string name,int quantity) {
   for(int i=0;i<numSoda;i++)
      if( inventory[i]->getName() == "NA") {
         inventory[i]->setName(name);
         this->quantity[i] = quantity;
         break;
      }
Soda* VM::getSoda(string name) {
   for(int i=0;i<numSoda;i++)
      if( inventory[i]->getName() == name)
         return inventory[i];
}
bool VM::buySoda(string name) {
   for(int i=0;i<numSoda;i++)
      if( inventory[i]->getName() == name && quantity[i] > 0)
         quantity[i]--;
         return true;
int main() {
   VM vm(5);
   vm.restock("Coke",4);
   vm.restock("Diet Coke",5);
   if(vm.buySoda("Coke"))
      cout << "I bought " << vm.getSoda("Coke")->getName()
<< endl;
   else
         cout << "Coke is sold out!!" << endl;
   if(vm.buySoda("Pepsi"))
```