# Introduction to Programming (CS 101)

## Spring 2024


## Lecture 21:

Classes continued (constructors, destructors), Exceptions

**Instructor:** Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade and Prof. Manoj Prabhakaran

# Recap (I): `cin`

```
#include <iostream>
using namespace std;

int main() {
  int a, b;
  string str;
  cin >> a >> b; // the user enters "5 12" and then presses Enter
  getline(cin, str); // the user enters "hello"
  cout << a << " " << b << " " << str;
}
```

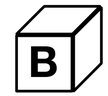
**OUTPUT: 5 12**

`cin` reads the newline left in the buffer when the user hits Enter. Next, when `getline(cin, str);` is called, it reads from the input stream until the delimiter (newline) is found. So, `str` is an empty string. To fix this, use `cin.ignore()` prior to `getline`.

# Recap (II): Pointer assignment

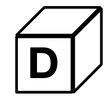
```
#include <iostream>
using namespace std;

int main() {
  int a = 2, b = 3;
  int* p = &a;
  int* q = &b;
  p = q;
  *p = (*q) + 1;
  cout << a << " " << b << endl;
}
```


3 3




4 3


2 4




4 4

With `p = q,` both pointers contain the address of `b`. `*p` and `*q` would both deference `b`

# Recap (III): new, pointer scope, list

What is the output of the following program?

```
#include <iostream>
using namespace std;

struct Student {
  int val;
  Student* next;
};


void printlist(Student* head) {
  while(head) {
    cout << head->val << endl;
    head = head->next;
  }
}
```

```
Student* pushQ(Student* head, int value) {
  Student* s = new Student;
  s->val = value; s->next = head;
  return s;
}

void clearlist(Student* head) {
  while(head) {
    Student* tmp = head;
    head = head->next;
    delete tmp;
  }
}

int main() {
  Student* s1 = new Student; s1->val = 25;
  s1->next = nullptr;
  s1 = pushQ(s1, 36);
  printlist(s1);
  clearlist(s1);
}
```

```
36
25
```

Can return a local variable `s` defined inside `pushQ` because `s` is a pointer to a memory region within the heap -- which is accessible across functions

# Recap (IV): Access to members

```
#include <iostream>
using namespace std;
class Teacher; //forward declaration
class Student {
  float value;
  public:
    string name;
    void setName(string n) { name = n; }
    void access(Teacher&);
};

class Teacher{
  float value;
  public:
    string name;
    void setName(string n) { name = n; }
    void access(Student&);
};
```

```
void Student::access(Teacher& obj) {
  cout << name << " accesses value of "
               << obj.name << endl;
};

void Teacher::access(Student& obj) {
  cout << name << " accesses value of "
               << obj.name << endl;
};

int main()
{
    Teacher alice; alice.setName("alice");
    Student bob; bob.setName("bob");
    alice.access(bob); bob.access(alice);
    return 0;
}
```

If it was `obj.value` instead of `obj.name` in the `access` functions, it would throw a compiler error since we're trying to access a private member of another class.

**OUTPUT:**

alice accesses value of bob
bob accesses value of alice

# Recap (V): Namespaces and classes

```
#include <iostream>
using namespace std;

namespace Data {
  int value = 10;
  class Data {
      int value;
    public:
      void setValue(int v) { value = v; }

      void show() {
        cout << value << " " << Data::value << endl;
      }
  };
}
```

```
int main() {
  Data::Data obj1, obj2;
  obj1.setValue(10);
  Data::value += 20;
  obj2.setValue(20);
  obj1.show(); obj2.show();

  return 0;
}
```

**OUTPUT:**
10 10
20 20

`Data::value` in a member function accesses `value` corresponding to the object. In order to access `value` in the namespace, use `::Data::value`


# Classes: Constructors, Destructors

CS 101, 2025

# Recap: Classes

```
class BankAccount {
  private:
    unsigned long int acctnum;
    string name;
    float balance;

  public:
    string getName() { return name; }
    void updateBalance();
    void setName(string n);
};

void BankAccount::updateBalance() {
   ...
}
```

Members with different access levels `private`, `public`

Private members are accessible only within class member functions

Public members are accessible outside the class as well

Access to private members is via "getter" functions (that are public)

Modification of private members is via "setter" functions (that are public)

Member functions can be defined outside the class definition with the `Bankaccount::` modifier

# Constructors and Destructors

- Classes are complex data types with internal structures including data and functions
  - Set-up and clean-up for classes is more involved than for simple data structures
- *Constructors* and *destructors* are special member functions of classes that are used to *construct* and *destroy* objects, respectively.
- Typically:
  - Constructors deal with memory allocation and variable initialisations
  - Destructors deal with memory deallocation
- Compiler automatically calls constructors when defining objects and destructors when class objects go out of scope

# Constructors

Can define one or more **constructor** functions in the class, which will be automatically invoked when the object's life begins (when, e.g., comes into scope or new is called).

Special syntax: *classname*(*arguments*)
No return type. Typically public.
Can be overloaded.

Before the constructor is executed, all the members are initialised (using their constructors), in the order in which they appear in the class

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  queue() {}
  queue(int v) {enqueue(v);}
};
```

# Constructors

- A constructor can specify how to initialise the members before its own code is invoked
  - By providing the inputs to their (default or user-defined) constructors
  - Otherwise, their default initialisation (if any) will be used

```
class box {
  int h = 100, w = 100;
public:
  box(int a) : h(a) {
    // when this code starts
    // h, w already initialized
    // h=a and w=100
  }
};
```

Before the constructor is executed, all the members are initialised (using their constructors), in the order in which they appear in the class

# Constructors

```
// these call the constructor without args
queue Q1;
queue* q1 = new queue;

// these call the one with an int arg.
queue Q2(3);
queue* q2 = new queue(4);
```

OK to have no constructor. If so, compiler implicitly adds a "default" constructor (with empty arguments and empty body).

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  queue() {}
  queue(int v) {enqueue(v);}
};
```

# Constructors demo

Demo of boxes.cpp and various constructors (shared on Moodle)

# Default constructors

What is the output of the following program?

```
#include <iostream>
using namespace std;

class A {
  public:
    int i = 0, j = 0;
    A() { i = 5; j = 10; }
};

class B {
  public:
    int i = 0, j = 0;
    A a;
};

int main() {
  B b;
  cout << b.i << " " << b.j << " " << b.a.i << " " << b.a.j << endl;
}
```

**OUTPUT:**
0 0 5 10

# Copy Constructor

- Often a new object needs to be constructed by copying an existing object
  - E.g., passing arguments by value
- We can explicitly specify how it should work
  - E.g., a "deep copy" for queue
- If none specified, the compiler adds a default copy constructor which copies all members (using their copy constructors)

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
};
```

# Copy Constructor

Deep copy

The default copy constructor would have just copied head, tail

```
queue::queue(const queue& q) {
  for(node* n = q.head; n; n = n->next)
    enqueue(n->val);
}
```

Being a function in the class, all members (including the private members) are visible here.

Can also access the private members of other objects of the same class.

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
};
```

# Copy Constructor

```
queue::queue(const queue& q) {
  for(node* n = q.head; n; n = n->next)
    enqueue(n->val);
}
```

```
queue Q1;
...
// invoking the copy constructor
queue Q2(Q1); // not an assignment
```

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
};
```

# Destructor

```
void queue_demo() {
  queue Q;
  for(int i=0; i < 10000000; i++)
    Q.enqueue(1);
  Q.clear();
}
```

Recall: Even though `Q` is a local variable that is automatically destroyed, if `clear()` not called here, memory leak!

Can define a **destructor** function in the class, which will be automatically invoked when the object's life ends (when, e.g., goes out of scope or delete called).

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
};
```

# Destructor

```
void queue_demo() {
  queue Q;
  for(int i=0; i < 10000000; i++)
    Q.enqueue(1);
  Q.clear();
}
```

`Q.~queue()` is called here.

Can define a **destructor** function in the class, which will be <u>automatically invoked</u> the object's life ends (when, e.g., goes out of scope or delete called).

Special syntax: *~classname()*. No return type.

Typically public. No overloading in destructors.

*After* the destructor code finishes, destructor of each member (if present) is invoked.

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
};
```

# Constructors and Destructors: Example

What is the output of the following program?

```
#include <iostream>
class Testing {
  private:
    int t_id;

  public:
    Testing(int i): t_id(i) {
      std::cout << "Constructor " << t_id << '\n';
    }
    ~Testing() {
      std::cout << "Destructor " << t_id << '\n';
    }
};

int main() {
   Testing t1(1);
   { Testing t2(2); }
   return 0;
}
```

**OUTPUT:**
Constructor 1
Constructor 2
Destructor 2
Destructor 1

# Other features of C++ (that we will not cover)

- Inheritance
- Polymorphism
- Templates
- Smart pointers
- Complex casting
- Operator overloading
- Associative arrays (maps, sets)

⋮