Introduction to Programming (CS 101)

Spring 2024

Lecture 20:

Debugging/exceptions, Introduction to classes

Instructor: Preethi Jyothi

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

Recap (IA): Access to static variable

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

int* counter() {
  static int l = 0;
  l++; cout << l << " ";
  return &l;
}

int main() {
  (*counter())++;
  counter();
}
```

- A: 1 3
- B: 1 1
- C: 1 2
- D: 0 2
- E: 0 1

One can use pointers to access a static variable outside the function where it is defined.

Recap (IB): Static and global variables

```
#include <iostream>

int g = 1;

int display(int j) {
  static int g = 2;
  return (g+=j);
}

int main() {
  g = display(g);
  std::cout << display(g);
}
```

- A 3
- B 6
- C 5
- D 2

Lifetime of both static and global variables is the entire duration of the program

Recap (II): Namespace with function overloading

The literal `3.14` is of type double. Since there is no exact match and double could be cast to either int or float, this will throw a compiler error.

Changing `int show(float a)` to `int show(double a)` will print 13.

```
#include <iostream>
namespace out {
  int i = 5;
  int show(int a) {
    i += a;
    return i;
  }
}

namespace out {
  int j = 5;
  int show(float a) {
    j += (a + i);
    return j;
  }
}

int main() {
  std::cout << out::show(3.14);
}
```

A 13.14

B 13

C 8

D Compiler error

Recap (III): `new` and `delete`

```cpp
#include <iostream>

struct Book {
  std::string title;
  float price;
};

struct Shelf {
  int capacity;
  Book* b;
};
```

```cpp
int main() {
  Shelf* sh = new Shelf;
  sh->capacity = 16; sh->b = new Book;
  {
    Shelf* sh, sh1;
    sh1.capacity = 25; sh1.b = nullptr;
    sh = &sh1;
  }
  std::cout << sh->capacity;
  delete sh->b; delete sh;
}
```

OUTPUT: 16

Local Shelf variable (`sh`) shadows a previously defined Shelf variable (`sh`)

`delete sh` alone will not deallocate the memory allocated to `sh->b`

Debugging

CS 101, 2025

When Things Go Wrong

- Programs should be written to be robust against all possible inputs in all possible environments
  - "Foolproof"

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

- Douglas Adams

- So far, our programs have focused on the "normal cases" rather than the exceptions

Wrong Kind of Inputs

- Recall that std::cin helps with reading in formatted inputs
- If the input is not of the right kind, it will "silently fail"

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main() {
  int x, sum = 0;
  cout << "Enter non-negative numbers to sum (end with -1): ";
  for(cin >> x; x >= 0; cin >> x)
    sum += x;
  cout << "Sum is " << sum << endl;
}
```

- Goes into an infinite loop if a non-number input is included!

Input Errors

- iostream objects set internal flags when errors occur
  - They can be checked via public functions

```
cin.fail(); // or just !cin (i.e., !bool(cin))
            // true if >> failed: wrong/no input, or bad()
cin.bad();  // true if system-level IO failure
```

  - Can try to recover from wrong input by discarding inputs

```
cin.clear();      // clear the fail flag (so that we can retry)
```

Checking for Errors After Input

- A possible fix:
  - Ideally, should notify the user about the error

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main() {
  int x, sum = 0;
  cout << "Enter non-negative numbers to sum (end with -1): ";
  for(cin >> x; cin && x >= 0; cin >> x)
    sum += x;
  cout << "Sum is " << sum << endl;
}
```

Checking for Errors After Input

- A possible fix:
  - Ideally, should notify the user about the error

```
#include <iostream>
using std::cin; using std::cout; using std::endl; using std::cerr;
int main() {
  int x, sum = 0;
  cout << "Enter non-negative numbers to sum (end with -1): ";
  for(cin >> x; cin && x >= 0; cin >> x)
    sum += x;
  if(!cin)
    cerr << "There was an error while reading inputs." << endl;
  cout << "Sum is " << sum << endl;
}
```

`cerr` for debugging

- When printing out debugging statements, good practice to use `std:cerr` instead of `std:cout`

- By default, both `std::cout` and `std::cerr` print text to the console

- Output streams (specific to `cout` and `cerr`) can be redirected to files, so that these files can be reviewed later

    - `g++ main.cpp 2>err.txt`

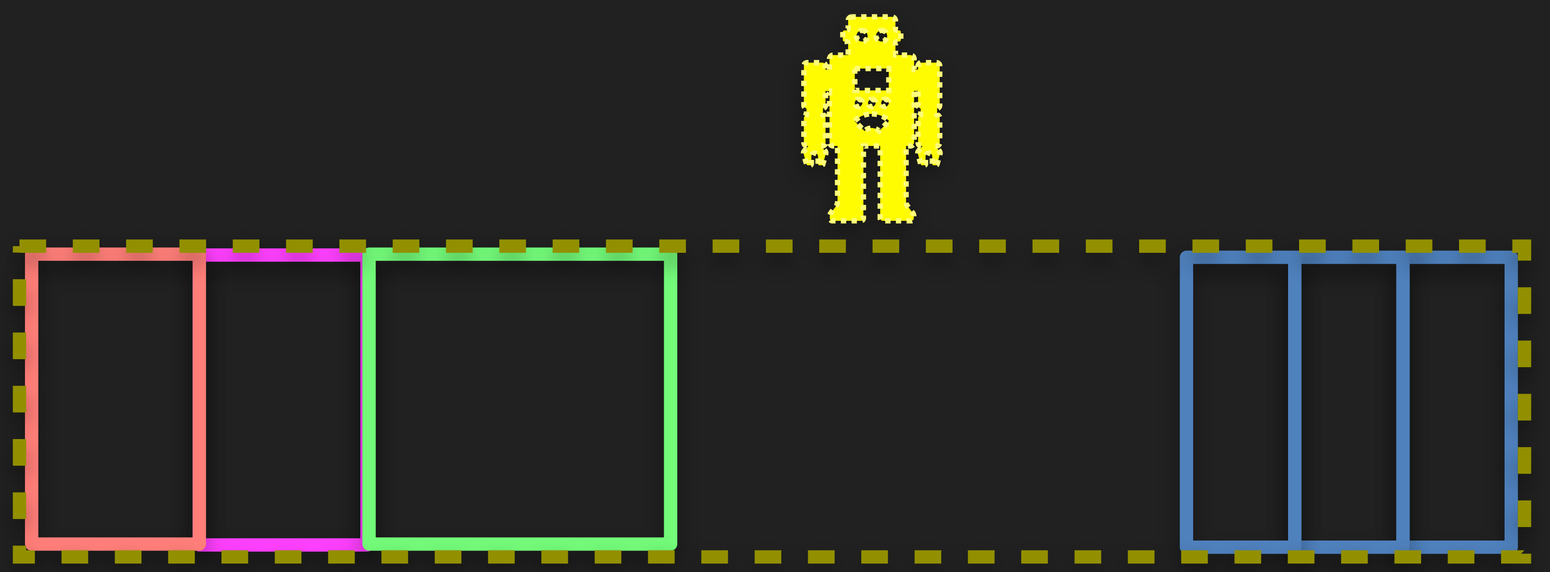
    - This command redirects everything printed via `cerr` to an output file named `err.txt`

Classes

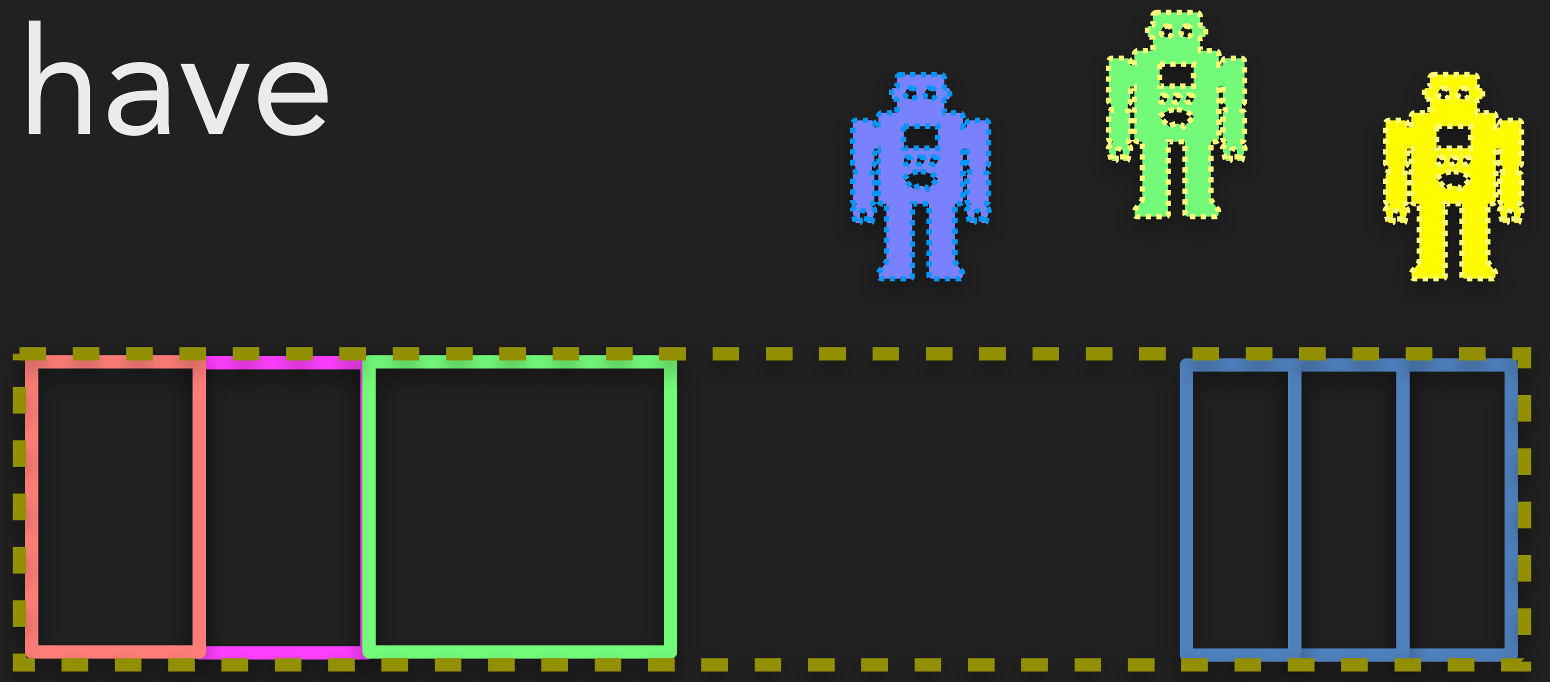
CS 101, 2025

Object-Oriented Programming

- Recall our model of program execution
  - A single agent (processor) executing instructions and reading/updating memory



- But we may think of different functions as different agents, one calling up another and waiting for a response
  - Static variables allow each of these agents to have memory (state) across calls
  - But only one such state per function



- In OOP, we think in terms of dynamically created agents (possibly many of the same kind) with their own states, interacting with each other

Structs a.k.a. Classes

- In C, *objects* that carry complex data have a `struct` type
- C++ provides more support for "Object-Oriented Programming"
  - Objects not only carry data, but also "know" how to compute on that data
    - Note: The "box" associated with an object stores its data
  - The type of such an object is traditionally called its *class*
- In C++, keywords `struct` and `class` both refer to the same idea, with a small difference w.r.t. member accessibility

Structs a.k.a. Classes

```
struct node {
  int val;
  node* next = nullptr;
};
```

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

These members are "private."

Visible in member functions of the same class, but nowhere else in the program.

Encapsulation:

Keep implementation details private. Expose only the interface publicly.

```
struct node {
  int val;
  node* next = nullptr;
};
```

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

Class

Optional keyword; members treated as private by default

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

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

Member function defined inside the class definition: *Inline* member function

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

Member function defined outside the class definition. Note function name is then specified as <classname>::<functionname>

Structs a.k.a. Classes

```
struct node {
  int val;
  node* next = nullptr;
};
```

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

These members are "private."

Visible in member functions of the same class, but nowhere else in the program.

Can do the same in a struct too. But the default is public.

```
struct node {
  int val;
  node* next = nullptr;
};
```

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

More Encapsulation

- Keeping implementation details inaccessible outside of the class ensures that the entire class can be reimplemented without affecting code that uses objects of the class

Private members are not visible outside of the class

Classes/structs can be nested

The struct node is an implementation detail of the class queue, so its definition can be moved within the class queue

(Note that the class queue doesn't have any member of the type node.)

```
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();
};
```

Access level

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

class Student
{
private:
    string name;

public:
    void teaches(const Student& s)
    {
        cout << name << " teaches " << s.name << '\n';
    }
    void setName(string n) { name = n; }
};
```

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

This program prints "`alice teaches bob`". A member function can access the private members of ***any other object*** of the same class type (that's in scope).

More about classes... in the next class

CS 101, 2025