Introduction to Programming (CS 101)

Spring 2024

Lecture 19:

Namespaces, Variable scope, Global/Static Variables

Instructor: Preethi Jyothi

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

Recap (I): Works?

```
#ifndef MATHOPS_H
#define MATHOPS_H

#include <cmath>

float roundup(float a) {
    return ceil(a);
}

#endif  // MATHOPS_H
```

mathops.h

```
#include <iostream>
#include "mathops.h"
#include "mathops.h"

int main() {
   float input; std::cin >> input;
   std::cout << "Round up " << input << " to "
             << roundup(input) << std::endl;
}
```

INPUT: **22.3**

OUTPUT: **Round up 22.3 to 23**

main.cpp

Adding `#include "mathops.h"` twice will not cause any issues because of the header guard (within the `ifndef` block)

Recap (II): What does this function return?

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

```
node* find(node* head) {
  int l = 0;
  node* tmp = head;
  while(tmp) {
    l++; tmp = tmp->next;
  }
  tmp = head;
  for(int i = 0; i < l/2; i++) {
    tmp = tmp->next;
  }
  return tmp;
}
```

Returns a pointer to the middle node in the linked list. 1->2->3->4, points to 3; 1->2->3, points to 2. Requires two passes.

```
node* find(node* head) {
  node* n1 = head;
  node* n2 = head;
  while(n2 && n2->next) {
    n1 = n1->next;
    n2 = n2->next->next;
  }
  return n1;
}
```

Can you find the middle node by only traversing the list once?

Namespaces

CS 101, 2025

Namespaces

- Standard library contains useful functions (swap, max, min, distance, begin, end, sort, move, …), data types (string, vector, list, …) and operators (cout, cin, …), many with common names
- But this can be problematic, especially due to function overloading!

Namespaces

- Standard library contains useful functions (swap, max, min, distance, begin, end, sort, move, …), data types (string, vector, list, …) and operators (cout, cin, …), many with common names
- But this can be problematic, especially due to function overloading!
- Suppose you write a function `to_string` as follows:

```
#include <simplecpp>
string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
    short a = 1; int b = 1;
    cout << to_string(a) << " vs. " << to_string(b) << endl;
}
```

invokes our to_string

invokes to_string from the standard library!

```
non-zero vs. 1
```

Namespaces

```
#include <simplecpp>
string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
     short a = 1; int b = 1;
     cout << to_string(a) << " vs. " << to_string(b) << endl;
}
```

```
non-zero vs. 1
```

- Why did this happen?
  - Standard library already has a function (included via `<simplecpp>`)
    `string to_string (int)`
    (but no function that takes a short — so it was not an error to define ours)
  - For the call `to_string(b)`, the compiler used this library function (which is a better fit than using our function which takes a `short`)

Namespaces

- <simplecpp> has a statement `using namespace std;` which made all the entities in std namespace available without the qualifier `std::`
  - We shall instead use the standard header <iostream>
- To keep entities (functions, types, variables) in a library separate from ours
  - `to_string` vs. `std::to_string`

Risky!

```
#include <iostream>
std::string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
    short a = 1; int b = 1;
    std::cout << to_string(a) << " vs. " << to_string(b) << std::endl;
}
```

Invokes our `to_string`, with b cast into a short.
Only `std::to_string` invokes the one from the library.

Namespace definition

```
namespace <name-of-namespace> {
  // declarations or definitions of names
}
```

- Example:

```
namespace num {
  int GCD(int, int);
  int LCM(int m, int n) { return (m*n)/GCD(m,n); }
}
```

- Inside the namespace, you can refer to names in it directly. Example: Use of `GCD` above in the `LCM` function
- Outside the namespace block, use the full `num::GCD` name
- The namespace directive `using namespace num;` allows all names within the namespace `num` to be used directly (without the `num::`)

Example

numbers.h

```
namespace num {
int GCD(int, int);
int LCM(int, int);
}
```

numbers.cpp

```
#include "numbers.h"
#include <cmath>
int num::LCM(int a, int b) {
      return std::abs(a*b)/GCD(a,b); // GCD is num::GCD
}
...
```

main.cpp

```
#include <iostream>
#include "numbers.h"
using std::cout; using std::cin; using std::endl;
int main() {
  cout << "Enter 2 positive numbers: ";
  int a, b; cin >> a >> b;
  if (a<=0 || b<=0) return -1;
  cout << "GCD(a,b) = " << num::GCD(a,b) << endl;
}
```

```
$ g++ -c main.cpp       # this produces main.o
$ g++ -c numbers.cpp    # this produces numbers.o
$ g++ main.o numbers.o  # this produces a.out
$ g++ main.cpp numbers.cpp # produces a.out directly
```

The global namespace

- Functions defined without using a namespace implicitly become part of a *global namespace*
- Using a name without a namespace qualifier means it is either in the global namespace or a named one. Functions in the global namespace can be accessed as `::<function-name>`

numbers.h

```
namespace num {
  int GCD(int, int);
  int LCM(int m, int n) { return (m*n)/GCD(m,n); }
}
```

main.cpp

```
using namespace num;

int LCM(int m, int n) { return m+n; }

int main() {
  cout << LCM(24,36);
}
```

Compiler error! Ambiguous use of `LCM`
Use either `num::LCM` or `::LCM` that refers to the global namespace

Namespaces

- Conventions to avoid unexpected conflicts
  - Every library should (and typically does) keep the entities they define within a separate (hopefully unique) namespace
    - E.g., std, boost, …
  - Programmers access entities in a library by explicitly specifying the namespace (e.g. `std::to_string(…)`, `std::string`, etc.)
  - But if desired, a programmer can shorten `nspace::entity` to just `entity` (say, because it is used in a lot of places in the program), by adding the statement `using nspace::entity;`
  - Alternately, one can write `using namespace nspace;` and the prefix `nspace::` can be dropped for *all* the entities in `nspace` (Might be risky, as we saw earlier with `to_string`!)

Example (I)

```
namespace One {
  int aggregate(int x, int y) {
    return x + y;
  }
}

namespace Two {
  int aggregate(int x, int y) {
    return x * y;
  }
}

int aggregate(int x, int y) {
  return x - y;
}

int main() {
  // What is the output?
  int x = 10, y = 5;
  cout << aggregate(x,y);
}
```

OUTPUT: **5**

Example (II)

```
namespace One {
  int aggregate(int x, int y) {
    return x + y;
  }
}

namespace Two {
  int aggregate(int x, int y) {
    return x * y;
  }
}

int aggregate(int x, int y) {
  return x - y;
}

int main() {
  // What is the output?
  int x = 10, y = 5;
  cout << Two::aggregate(x,y);
}
```

OUTPUT: **50**

Global and static variables

CS 101, 2025

Global variables

- Global variables are variables defined outside all functions
- These variables are accessible by any function. Example:

```
int i = 3; //global variable definition

void f() { i *= 3; } //accessing global variable

int main() {
  cout << i; //accessing global variable
  f();
  cout << " " << i << endl; //accessing global variable
}
```

- Use of global variables are generally not encouraged since any function could potentially alter it
- If a local variable has the same name as a global variable, then the local variable will *shadow* the global variable (example coming up)

Recall: Scope of Variables

- In C++, a variable can be used only where its declaration is "visible"
  - Visible only <u>within the "block"</u> it is declared in
  - And only <u>after</u> it is declared
  - Scope of a variable: region in the code where it is visible

```
{
  {
    // not visible here (before declaration)
    int x;
    // visible here
    {
      // visible here
    }

    // visible here
  }
  // not visible here (outside the block)
}
```

- A variable cannot be declared twice within the same block
  - However can declare a new variable with the same name (but possibly a different type) in a "sub-block"
  - In its scope, the new variable "shadows" the old one

Recall: Scope of Variables

```
void f(int x) {
  ...
}
```

```
{
  ...
}
```

```
for(int x=0;;) {
  ...
}
```

```
while(condition) {
  ...
}
```

```
{
  {
    // not visible here (before declaration)
    int x;
    // visible here
    {
       // visible here
    }

    // visible here
  }
  // not visible here (outside the block)
}
```

scope of x

- Examples of different kinds of blocks:
  - A function's body (including parameter declarations)
  - A block of statements enclosed in braces
  - A for loop (including declarations in the initialisation)
  - A while or do-while statement (condition can have declarations)
  - ...

Scope of Variables

```
int g; // a global variable. remains visible till the end of the file
...
void f(int x) { // x is visible inside the body of the function
  int y; // visible from here till the end of the function
  for(int g=x; g<3; g--) { // a new local g! visible till
    ...                     // the end of the for statement.
  } // now this g goes out of scope. global g visible again.
  {  // start of a new scope
    g = x + 1;  // this refers to the global g
    float g; // this is a different g! global g not visible.
  }  // now this g goes out of scope. global g visible again.
  g++;   // global g
}        // here x, y go out of scope.
```

Lifetime of Variables

- A variable is *created* (a "box" allocated for it) when control reaches its declaration
- It gets destroyed when the variable "goes out of scope"
  - i.e., control goes outside the block in which it was defined

```
{
  int c=0; // c "created" here
  while(c<12){
     int x = 2;  // x "created" in each iteration
     x++; c += x;
  } // at the end of each iteration x "destroyed"
} // here c is "destroyed"
```

Lifetime of Variables

- A variable is *created* (a "box" allocated for it) when control reaches its declaration
- It gets destroyed when the variable "goes out of scope"
  - i.e., control goes outside the block in which it was defined

```
for(int c=0 /* c "created" here */; c<12; ) {
  int x = 2;  // x "created" in each iteration
  x++; c += x;
}  // at the end of each iteration x "destroyed", but c is alive
// on exiting the loop, c is "destroyed"
```

Lifetime of Variables

- A variable is *created* (a "box" allocated for it) when control reaches its declaration
- It gets destroyed when the variable "goes out of scope"
- But a variable stays alive when it is shadowed

```
void f(int x) {// in each call of f, x is created and initialised

  for(int c=0 /* c "created" here */; c<12; ) {
    int x = 2;   // x "created" in each iteration// parameter x visible
    x++; c += x;                                . parameter x shadowed.
  }  // at the end of each iteration x "destroyed", but c is alive

  // on exiting the loop, c is "destroyed"
  return x; // parameter x's value to be returned. x is destroyed.
}
```

Static Variables in Functions

- Global variables (possibly declared in a namespace) are useful as they stay alive throughout the program.
  - But they can be modified from many points in the program, making it hard to debug
- A local variable in a function can be declared to be static, so that it behaves like a global variable in terms of lifetime, but a local variable in terms of scope
  - Like a global variable, the lifetime of a static variable starts when it is first accessed, and lasts till the end of the program
  - However, the scope is limited to the function: can only be accessed from within the function

Static Variables in Functions

- Example:
- Here, p will be initialised on the first call to the function
- Even after the function returns, p remains alive
- In subsequent calls, the value of p at the end of the previous invocation is retained (initialisation skipped)

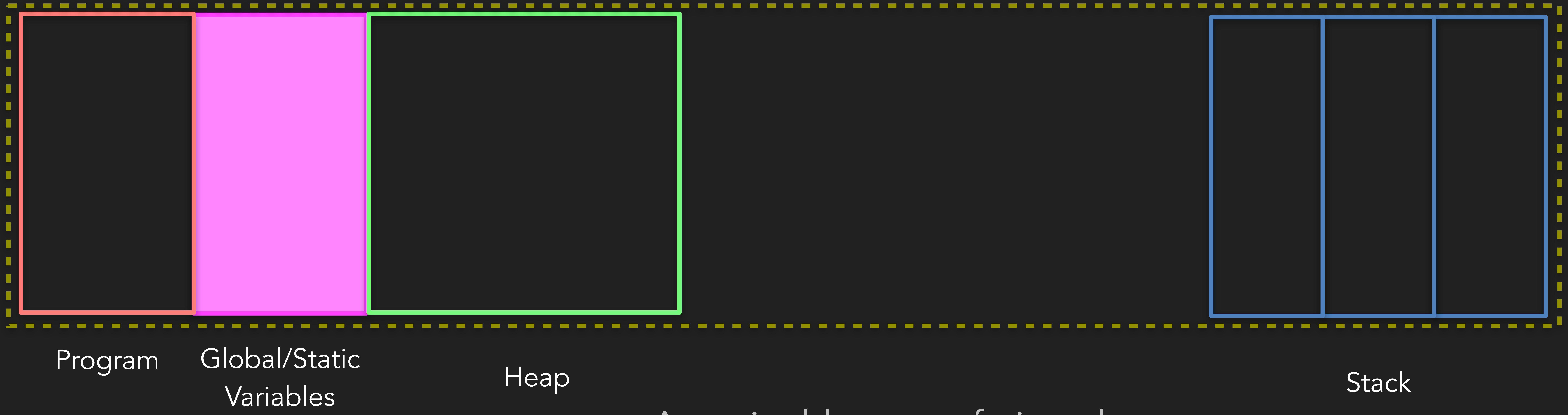
```
struct posn { double x, y, deg; };

posn change(double step, double turn) {
    static posn p = {0, 0, 0};

    p.deg += turn;
    p.x += step*cosine(p.deg);
    p.y += step*sine(p.deg);
    return p;
}
```

Global/Static Variables in Memory

- Global and static variables occupy a region of memory, separate from the stack
- These variables stay alive across function calls, for the duration of the program



A typical layout of virtual memory

Example of a counter using a static variable

```
#include <iostream>
using namespace std;
void tick() {
    static int count = 0;  // Static variable
    count++;
    cout << " count << " ";
}
int main() {
    for (int i = 0; i < 3; i++) {
        tick();
    }
    return 0;
}
```

OUTPUT: 1 2 3

- **Main takeaway**: A static variable inside a function retains its value across function calls. It is only initialized once during the lifetime of the program.

Debugging (Next class)

CS 101, 2025

"Debugging is like being the detective in a crime movie where you are also the murderer."

- Filipe Fortes