



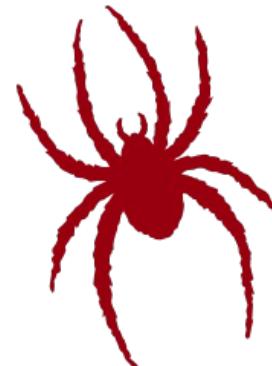
UNIVERSITY OF
RICHMOND

CMSC 240 Lecture 6

CMSC 240 Software Systems Development
Fall 2023

Today

- Code Style
- References
- Heap (Free Store)
- Structs
- In-class exercise



Today

- Code Style
- In-class exercise
- References
- Heap (Free Store)
- Structs





Only the computer
can read this

```
1 #include <iostream>
2 using namespace std; int main(void) {for (int number = 1;
3 number < 100; number++){if (number % 3 == 0 && number % 5 == 0)
4 {cout << "FizzBuzz ";}else if (number % 3 == 0){ cout << "Fizz ";
5 else if (number % 5 == 0){cout << "Buzz ";}else
6 {cout << number << " ";}cout << endl;return 0;}
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     for (int number = 1; number < 100; number++)
7     {
8         if (number % 3 == 0 && number % 5 == 0){
9             cout << "FizzBuzz ";
10        }
11        else if (number % 3 == 0){
12            cout << "Fizz ";
13        }
14        else if (number % 5 == 0){
15            cout << "Buzz ";
16        }
17        else{
18            cout << number << " ";
19        }
20    }
21    cout << endl;
22    return 0;
23 }
```



Code does not have
correct spacing

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     for (int number = 1; number < 100; number++)
7     {
8         if (number % 3 == 0 && number % 5 == 0)
9         {
10             cout << "FizzBuzz ";
11         }
12         else if (number % 3 == 0)
13         {
14             cout << "Fizz ";
15         }
16         else if (number % 5 == 0)
17         {
18             cout << "Buzz ";
19         }
20         else
21         {
22             cout << number << " ";
23         }
24     }
25     cout << endl;
26     return 0;
27 }
```



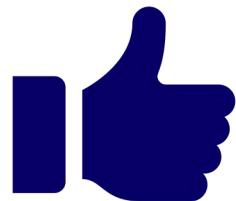
Good style!

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
{
5     float x, y;
6
7     int l = 0;
8     int u = 300;
9     int s = 20;
10
11    x = l;
12
13    while (x <= u)
14    {
15        y = (5.0 / 9.0) * (x - 32.0);
16
17        cout << x << "\t" << y << endl;
18
19        x = x + s;
20
21    }
22
23    return 0;
24 }
```



Variable names are
not descriptive

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
{
5     float fahrenheit, celsius;
6
7     int lower_limit = 0;
8     int upper_limit = 300;
9     int step_size = 20;
10
11    fahrenheit = lower_limit;
12
13    while (fahrenheit <= upper_limit)
14    {
15        celsius = (5.0 / 9.0) * (fahrenheit - 32.0);
16
17        cout << fahrenheit << "\t" << celsius << endl;
18
19        fahrenheit = fahrenheit + step_size;
20    }
21
22
23    return 0;
24 }
```



Good
variable
names!

```
1 #include <iostream>
2 using namespace std;
3
4 #define LOWER_LIMIT 0
5 #define UPPER_LIMIT 300
6 #define STEP_SIZE 20
7
8 int main()
9 {
10     float fahrenheit, celsius;
11
12     fahrenheit = LOWER_LIMIT;
13
14     while (fahrenheit <= UPPER_LIMIT)
15     {
16         celsius = (5.0 / 9.0) * (fahrenheit - 32.0);
17
18         cout << fahrenheit << "\t" << celsius << endl;
19
20         fahrenheit = fahrenheit + STEP_SIZE;
21     }
22
23     return 0;
24 }
```



Good use of
#define

```
1 #include <iostream>
2 using namespace std;
3
4 #define LOWER_LIMIT 0
5 #define UPPER_LIMIT 300
6 #define STEP_SIZE 20
7
8 /**
9  * This program will print a Fahrenheit-Celsius table.
10 * The table will start at the LOWER_LIMIT and stop at the UPPER_LIMIT.
11 * The table will increase by the STEP_SIZE.
12 */
13 int main()
14 {
15     float fahrenheit, celsius;
16
17     fahrenheit = LOWER_LIMIT;
18
19     // While the value for fahrenheit is less than the UPPER_LIMIT.
20     while (fahrenheit <= UPPER_LIMIT)
21     {
22         // Convert the fahrenheit value to celsius.
23         celsius = (5.0 / 9.0) * (fahrenheit - 32.0);
24
25         cout << fahrenheit << "\t" << celsius << endl;
26
27         // Increase the fahrenheit value by STEP_SIZE.
28         fahrenheit = fahrenheit + STEP_SIZE;
29     }
30
31     return 0;
32 }
```



Good use of
comments!

- **Code Quality (30 points)**

- **Readability (10 points)**

- Code has meaningful variable and function names: ____/5
- Code is consistently formatted and indented: ____/5

- **Modularity (10 points)**

- Code is appropriately divided into functions: ____/5
- Each function has a single responsibility: ____/5

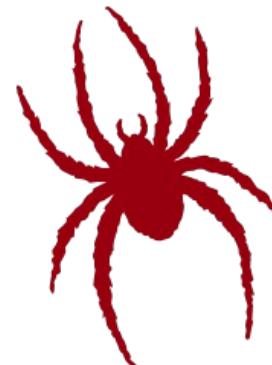
- **Documentation (10 points)**

- Code includes a header comment explaining the program's purpose: ____/3
- Each function has a comment explaining its purpose, inputs, and outputs: ____/5
- Inline comments explain non-obvious sections of the code: ____/2

```
1 #include <iostream>
2 using namespace std;
3
4
5 /**
6  * This program will print a Fahrenheit-Celsius table.
7  * The table will start at the LOWER_LIMIT and stop at the UPPER_LIMIT.
8  * The table will increase by the STEP_SIZE.
9 */
10 int main()
11 {
12     // Initialize fahrenheit to LOWER_LIMIT.
13
14     // While the value for fahrenheit is less than the UPPER_LIMIT.
15
16         // Convert the fahrenheit value to celsius.
17
18         // Print out the fahrenheit and celsius values.
19
20         // Increase the fahrenheit value by STEP_SIZE.
21
22
23     return 0;
24 }
```

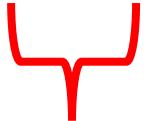
Today

- ~~Code Style~~
- References
- Heap (Free Store)
- Structs
- In-class exercise



A **pointer** is an object that holds an address value.

```
7 // Create a new integer value and assign it the number 50.  
8 int num = 50;  
9  
10 // Create a new integer pointer to hold the address of an integer value.  
11 int* pointer;
```

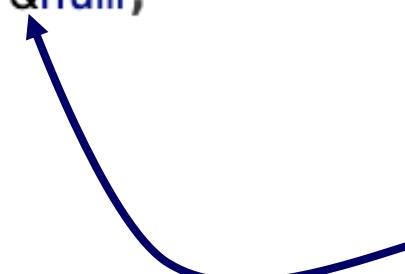


This new type is called
an “**int pointer**” and is
used to hold the address
of an integer variable

&

“address of”

```
7 // Create a new integer value and assign it the number 50.  
8 int num = 50;  
9  
10 // Create a new integer pointer to hold the address of an integer value.  
11 int* pointer;  
12  
13 // Put the address of the variable num into  
14 // the pointer using the "address of" operator &.  
15 pointer = &num;
```



The “**address of**”
the integer num

A diagram illustrating a pointer variable. At the top, there is a horizontal rectangle with a black border. Inside the rectangle, the word "ptr" is written in purple. A dark blue arrow originates from the bottom edge of this rectangle and points downwards and to the right. It then turns vertically downwards again, ending in a small blue arrowhead that points to the top edge of a second horizontal rectangle. This second rectangle also has a black border. Inside, the number "17" is written in red, and the word "var" is written in purple below it.

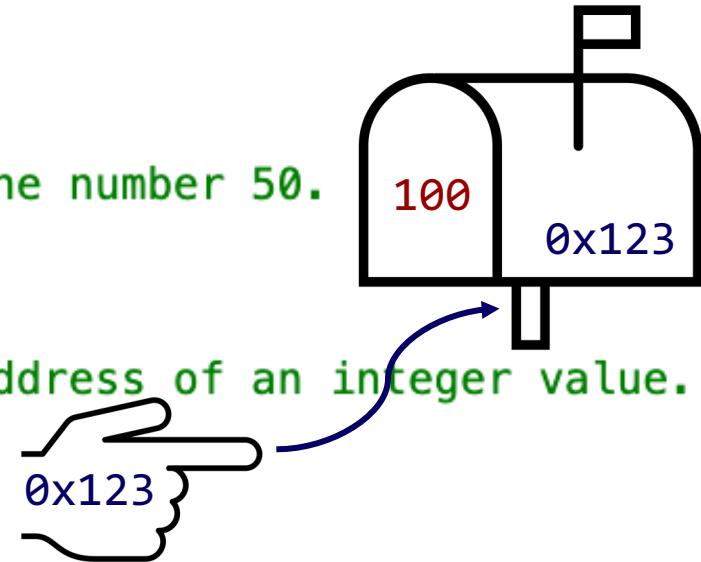
ptr

17
var

*

“contents of”

```
6 // Create a new integer value and assign it the number 50.  
7 int num = 50;  
8  
9 // Create a new integer pointer to hold the address of an integer value.  
10 int* pointer;  
11  
12 // Put the address of the variable num into  
13 // the pointer using the "address of" operator &.  
14 pointer = &num;  
15  
16 // The * symbol means "go to" the address contained in the pointer variable.  
17 // The address is that of the integer variable num.  
18 // So, now num is assigned the value 100.  
19 *pointer = 100;
```



The “**contents of**” the pointer `pointer`
(i.e., the value stored at the address)

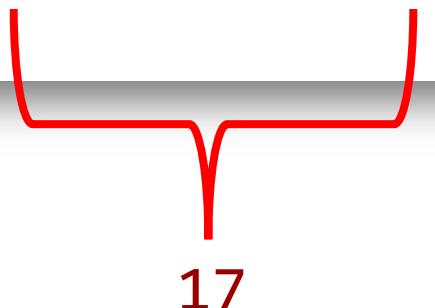
```
6 // Create a new integer value and assign it the number 50.  
7 int num = 50;  
8  
9 // Create a new integer pointer to hold the address of an integer value.  
10 int* pointer;  
11  
12 // Put the address of the variable num into  
13 // the pointer using the "address of" operator &.  
14 pointer = &num;  
15  
16 // The * symbol means "go to" the address contained in the pointer variable.  
17 // The address is that of the integer variable num.  
18 // So, now num is assigned the value 100.  
19 *pointer = 100;
```

The “**contents of**” the pointer `pointer`
(i.e., the value stored at the address)

```
int var = 17;
```

```
int* ptr = &var; // ptr holds the address of var
```

```
int anotherVar = *ptr;
```



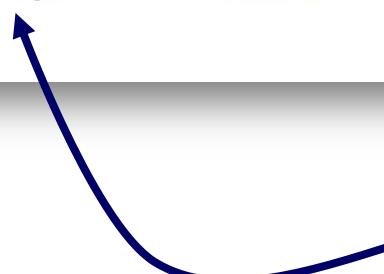
The “**contents of**” the
pointer `ptr`
(i.e., the value stored
at the address)



```
int var = 17;
```

```
int* ptr = &var; // ptr holds the address of var
```

```
*ptr = 24;           // assign a value to var through the pointer
```



The “**contents of**” the pointer `ptr`

A **reference** is an alternative name for an object.

```
int var = 17;
```

```
int& ref = var; // ref is now an alternative name for var
```

This new type is called
an **“int reference”**
and is used as an
alternative name for
an integer variable

No need for the &
“address of” operator here



```
int var = 17;
```

```
int& ref = var; // ref is now an alternative name for var
```

```
ref = 24; // assign a value to var through the reference
```

No need for *

```
int var = 17;
```

```
int& ref = var; // ref is now an alternative name for var
```

```
int anotherVar = ref; // read var through ref (no * needed)
```



17



No need for *

References

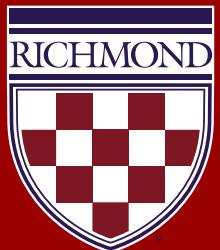
- An alternative name for an object
- Automatically dereferenced pointer
 - They always point to the “**contents of**” the original object
- Immutable
 - You cannot make a reference refer to a different object after initialization

```
int var = 17;

int& ref = var; // ref is now an alternative name for var

ref = 24;        // assign a value to var through the reference
```

Code Demo



```
int var = 17;

int& ref = var;          // ref is now an alternative name for var

cout << "var == " << var << endl;                                // #1
cout << "ref == " << ref << endl;                                // #2

ref = 24;                // assign a value to var through ref (no * needed)

cout << "var == " << var << endl;                                // #3
cout << "ref == " << ref << endl;                                // #4

int anotherVar = ref; // read var through ref (no * needed)

cout << "anotherVar == " << anotherVar << endl;                  // #5
```



```
// Swap two int values.  
void swap(int* a, int* b)  
{  
    int temp = *a;      // store contents of a in temp  
    *a = *b;            // put contents of b into a  
    *b = temp;          // put temp a into b  
}  
  
int main()  
{  
    int x = 12;  
    int y = 33;  
    swap(&x, &y);    // pass by pointer (the addresses of x and y)  
    cout << "x == " << x << "  y == " << y << endl;  
    return 0;  
}
```

```
// Swap two int values.  
void swap(int& a, int& b) // New reference variables refer to the original x and y
```

{

```
    int temp = a; // store contents of a in temp
```

```
    a = b; // put contents of b into a
```

```
    b = temp; // put temp a into b
```

}

```
int main()
```

{

```
    int x = 12;
```

```
    int y = 33;
```

```
    swap(x, y); // pass by reference, just use regular x and y
```

```
    cout << "x == " << x << " y == " << y << endl;
```

```
    return 0;
```

}

```
// Swap two int values.  
void swap(int& a, int& b) // New reference variables refer to the original x and y  
{  
    int temp = a;    // store contents of a in temp  
    a = b;          // put contents of b into a  
    b = temp;        // put temp a into b  
}  
  
int main()  
{  
    int x = 12;  
    int y = 33;  
    swap(x, y); // pass by reference, just use regular x and y  
    cout << "x == " << x << " y == " << y << endl;  
    return 0;  
}
```

It works!

x == 33 y == 12

- When you want to change the value of a variable to a value computed by a function, you have **three choices**:

```
3 // Pass a copy of the value
4 int addOne(int x)
5 {
6     // Compute a new value and return it
7     return x + 1;
8 }
```

```
9
10 // Pass a pointer
11 void addOne(int* x)
12 {
13     // Dereference pointer and increment the result
14     *x = *x + 1;
15 }
```

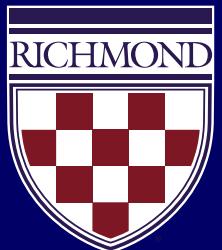
```
16
17 // Pass a reference
18 void addOne(int& x)
19 {
20     // Increment the value directly using the alternate name
21     x = x + 1;
22 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 // Pass a copy of the value
5 float fahrenheitToCelsius(float temperature)
6 {
7     // Compute a new value and return it
8     return (5.0 / 9.0) * (temperature - 32.0);
9 }
10
11 int main()
12 {
13     float temperature = 98;
14
15     // Call the fahrenheitToCelsius to convert the temperature.
16     temperature = fahrenheitToCelsius(temperature);
17
18     return 0;
19 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 // Pass a pointer
5 void fahrenheitToCelsius(float* temperature)
6 {
7     // Dereference the pointer and compute the result
8     *temperature = (5.0 / 9.0) * (*temperature - 32.0);
9 }
10
11 int main()
12 {
13     float temperature = 98;
14
15     // Call the fahrenheitToCelsius to convert the temperature.
16     fahrenheitToCelsius(&temperature);
17
18     return 0;
19 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 // Pass a reference
5 void fahrenheitToCelsius(float& temperature)
6 {
7     // Compute the value directly using the alternate name
8     temperature = (5.0 / 9.0) * (temperature - 32.0);
9 }
10
11 int main()
12 {
13     float temperature = 98;
14
15     // Call the fahrenheitToCelsius to convert the temperature.
16     fahrenheitToCelsius(temperature);
17
18     return 0;
19 }
```

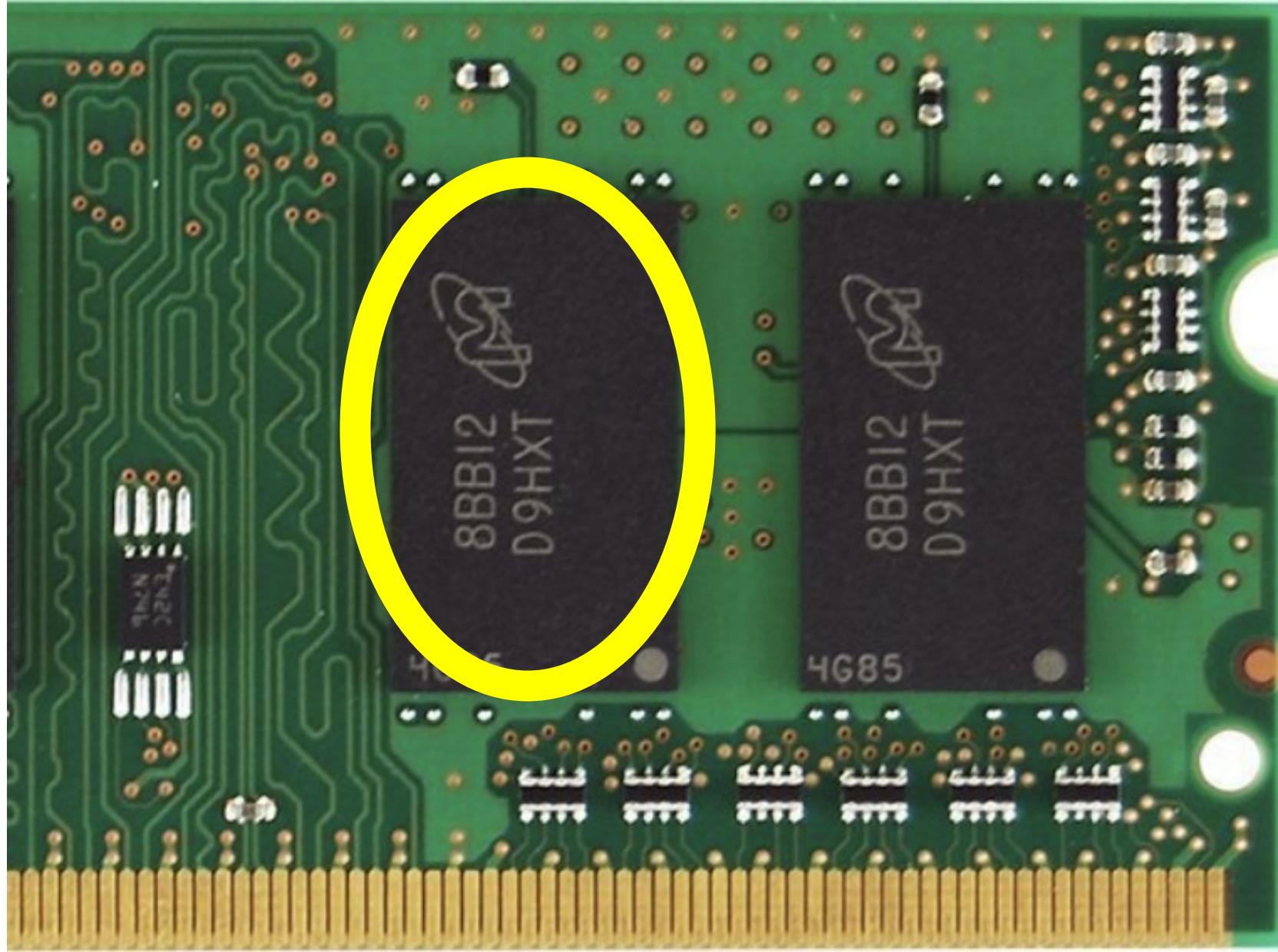
Ask a question



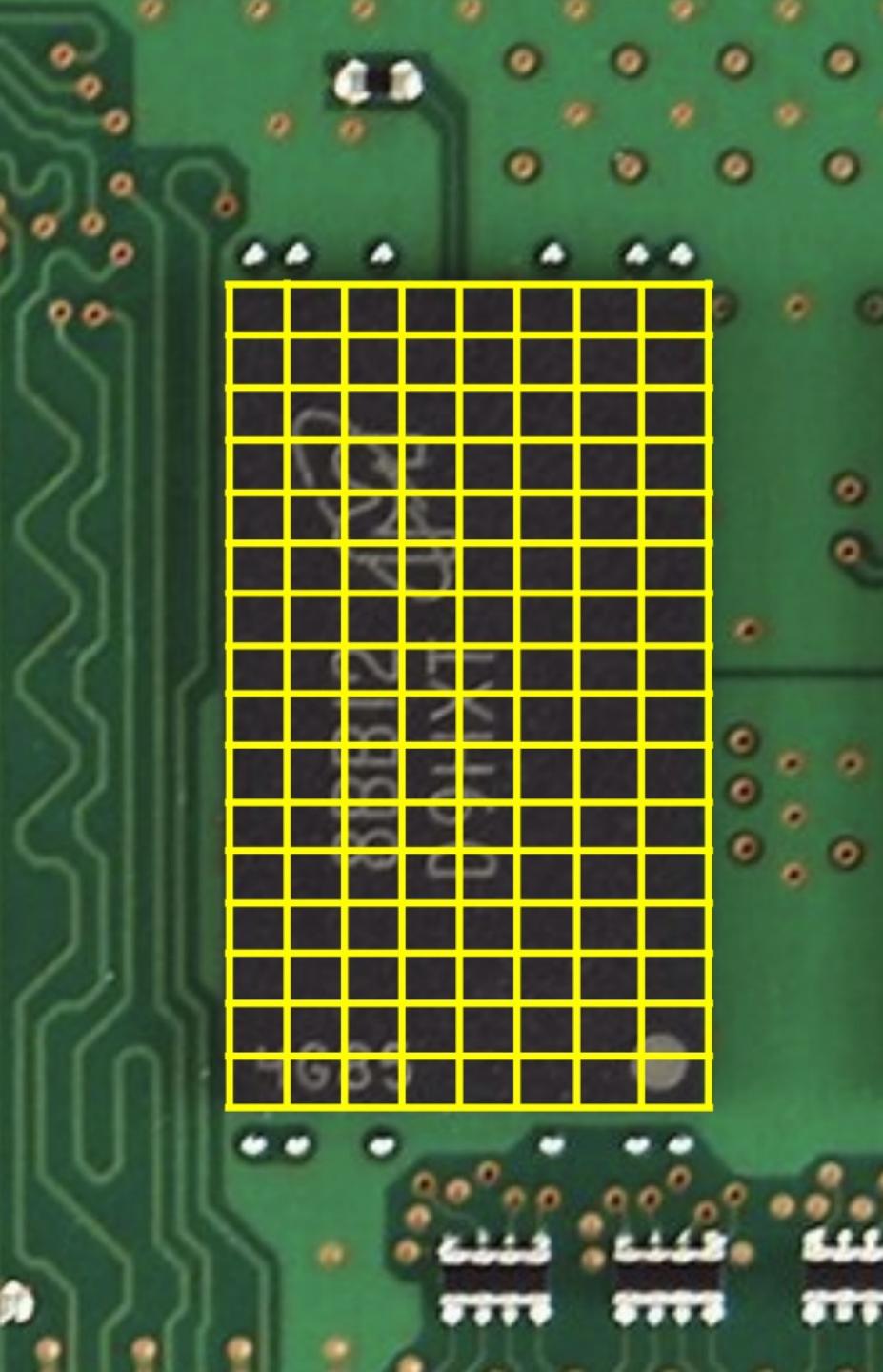
Today

- ~~Code Style~~
- ~~References~~
- Heap (Free Store)
- Structs
- In-class exercise

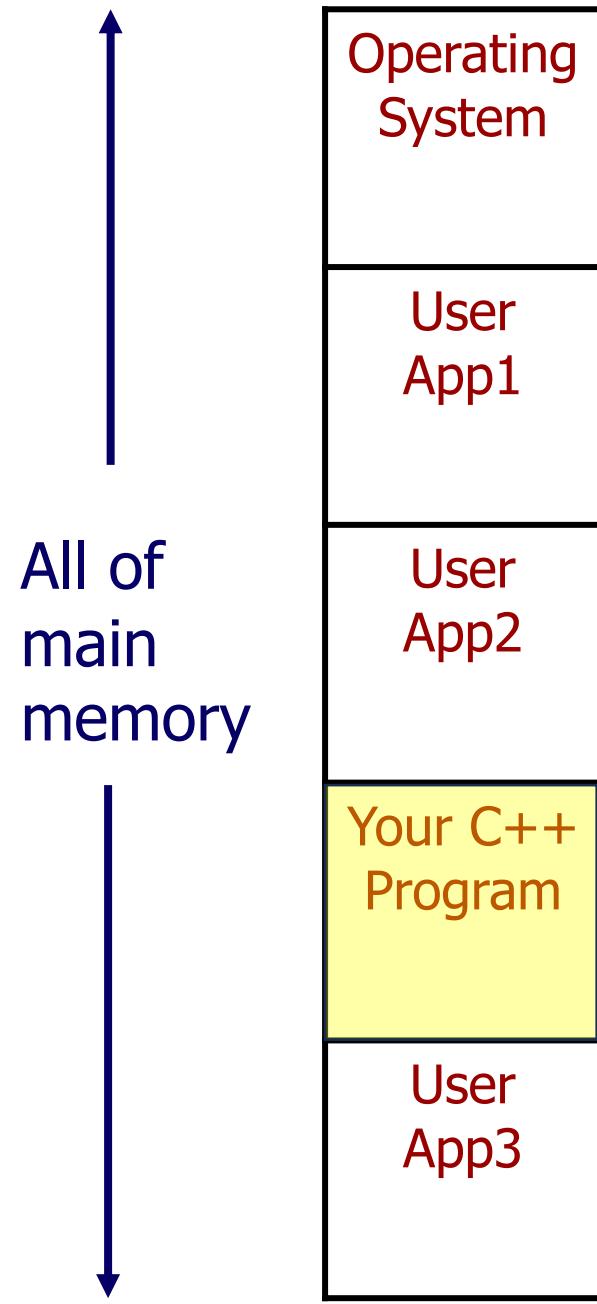






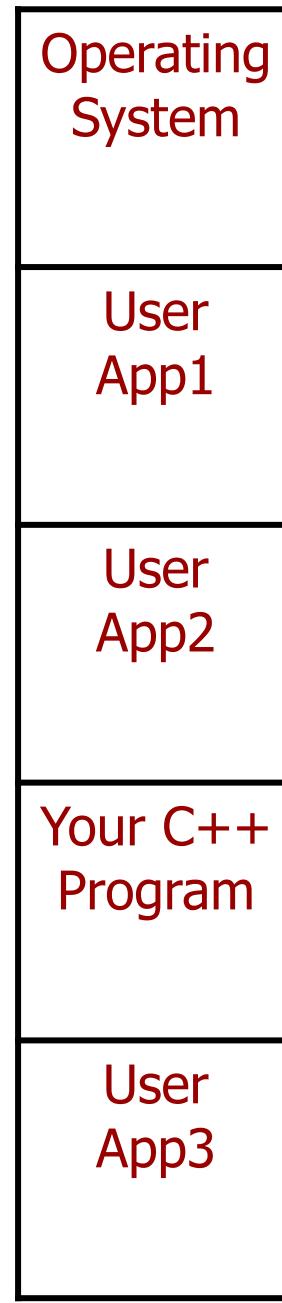


0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47
0x48	0x49	0x4A	0x4B	0x4C	0x4D	0x4E	0x4F

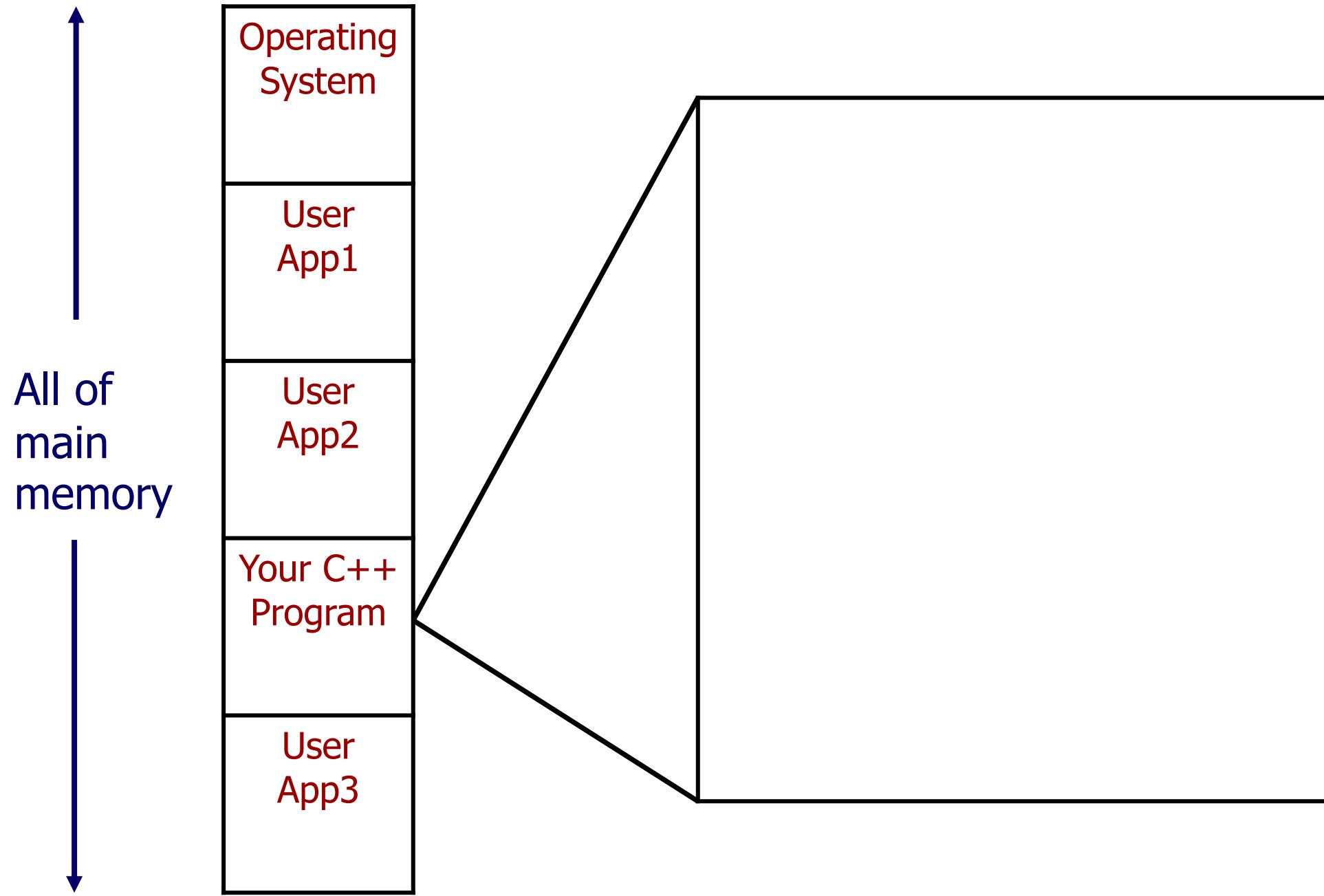


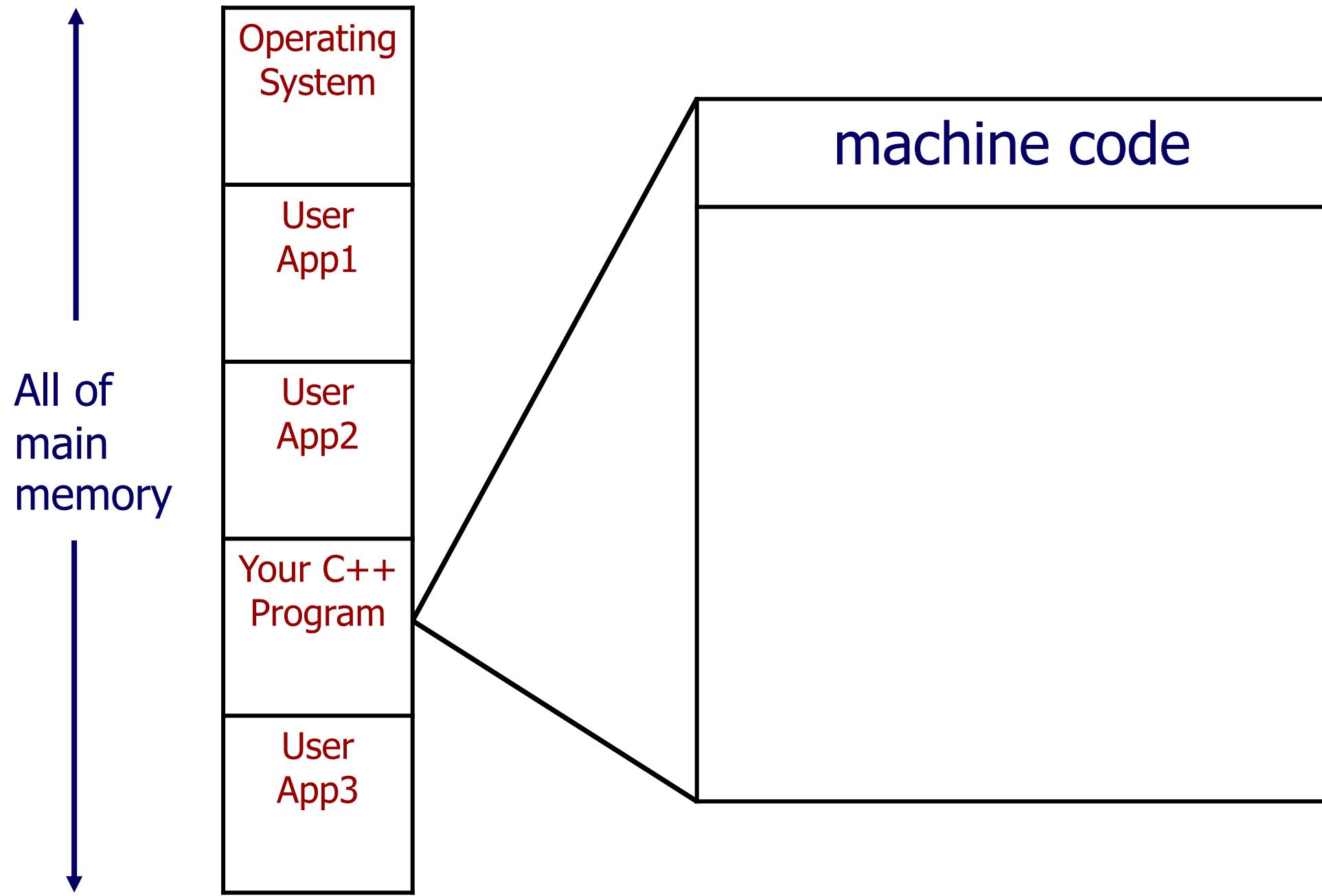
All of main memory

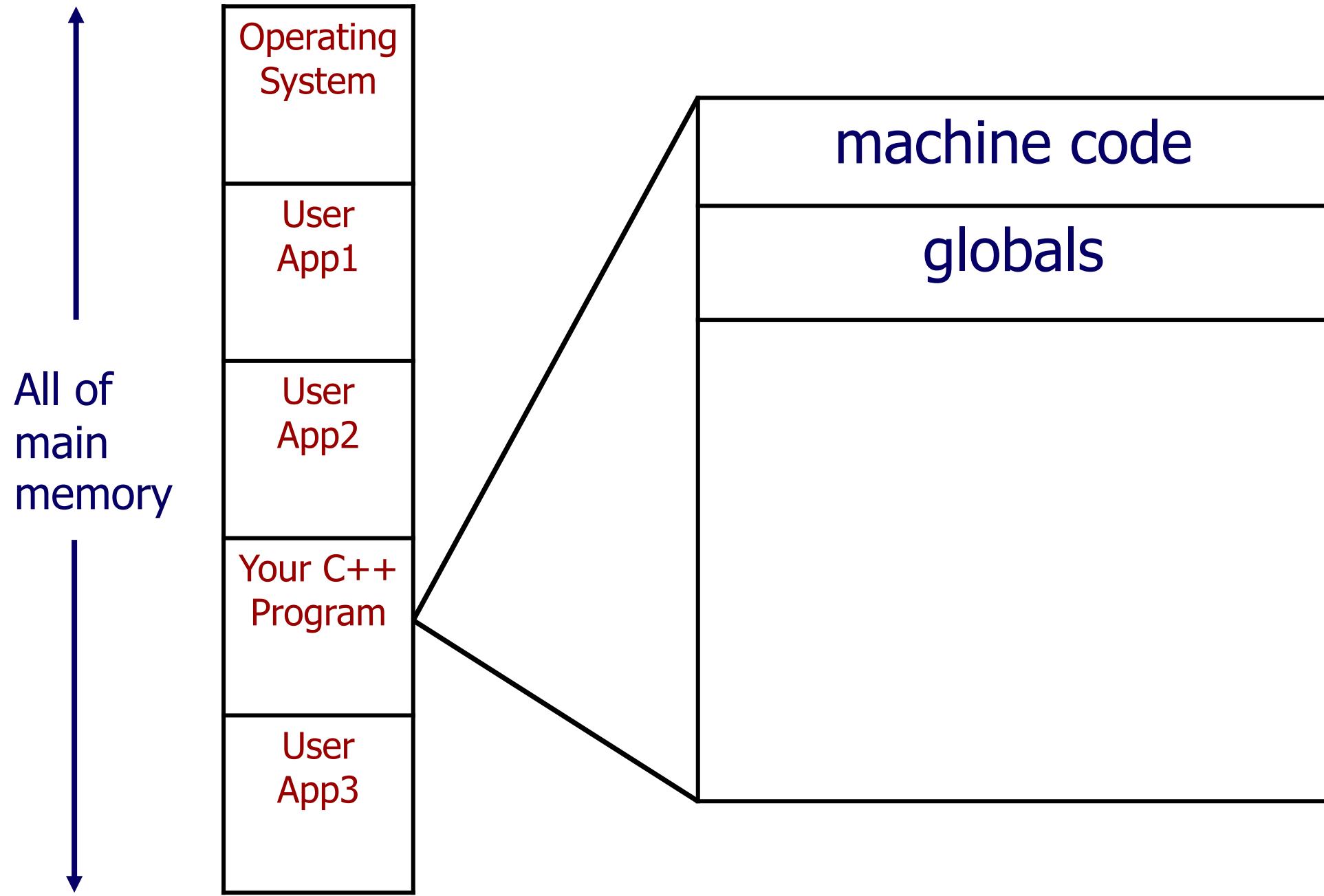
↑
↓

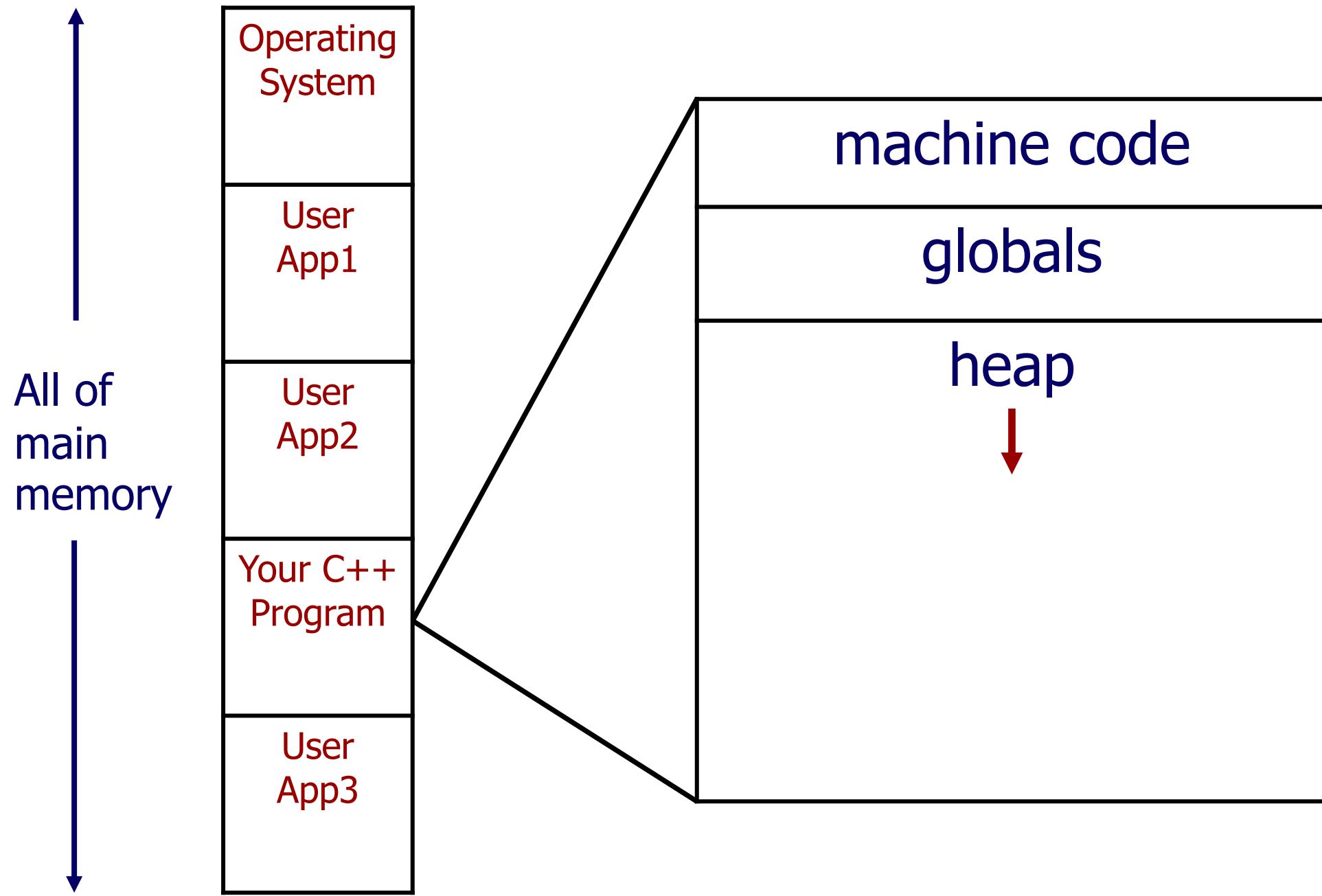


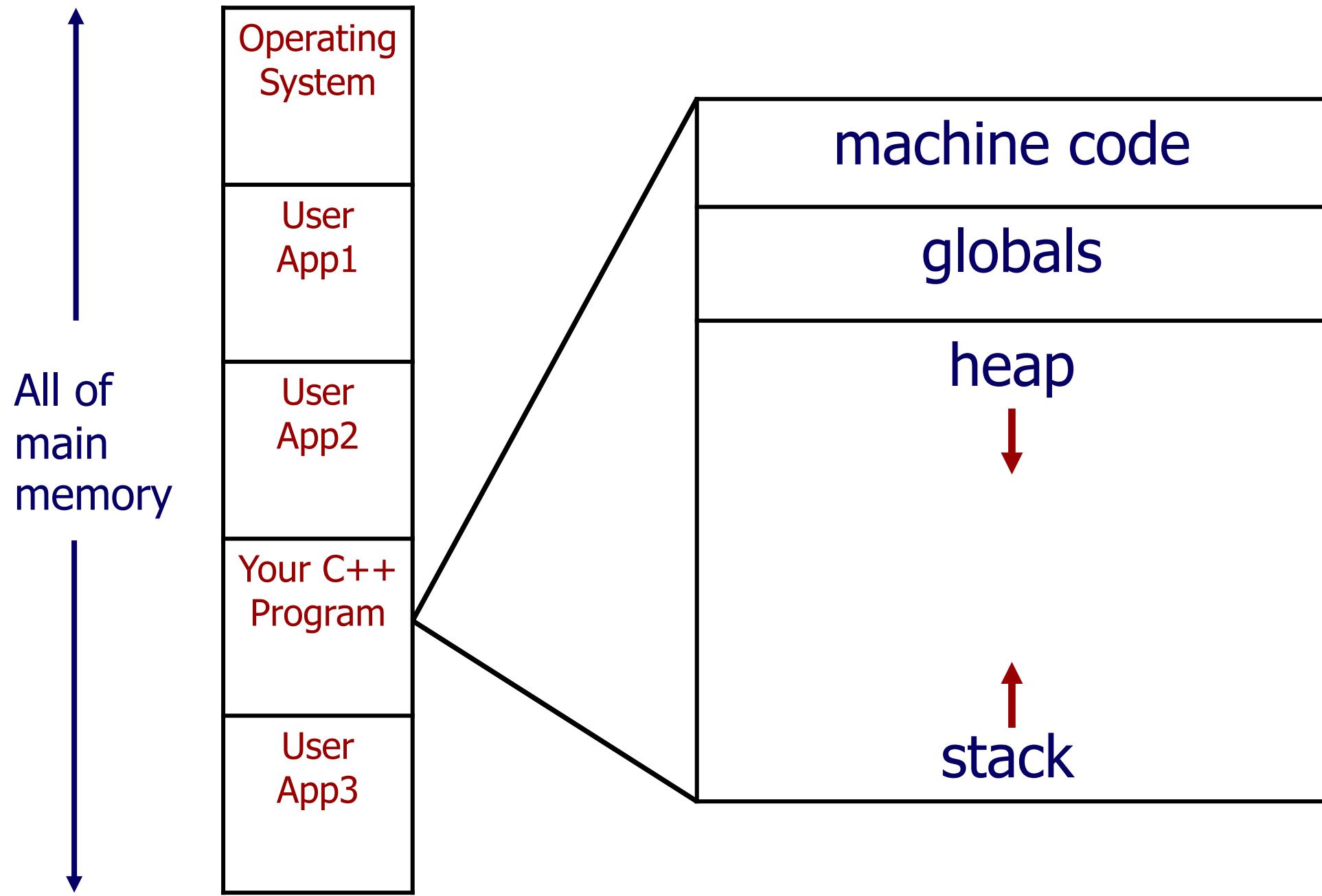
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47
0x48	0x49	0x4A	0x4B	0x4C	0x4D	0x4E	0x4F





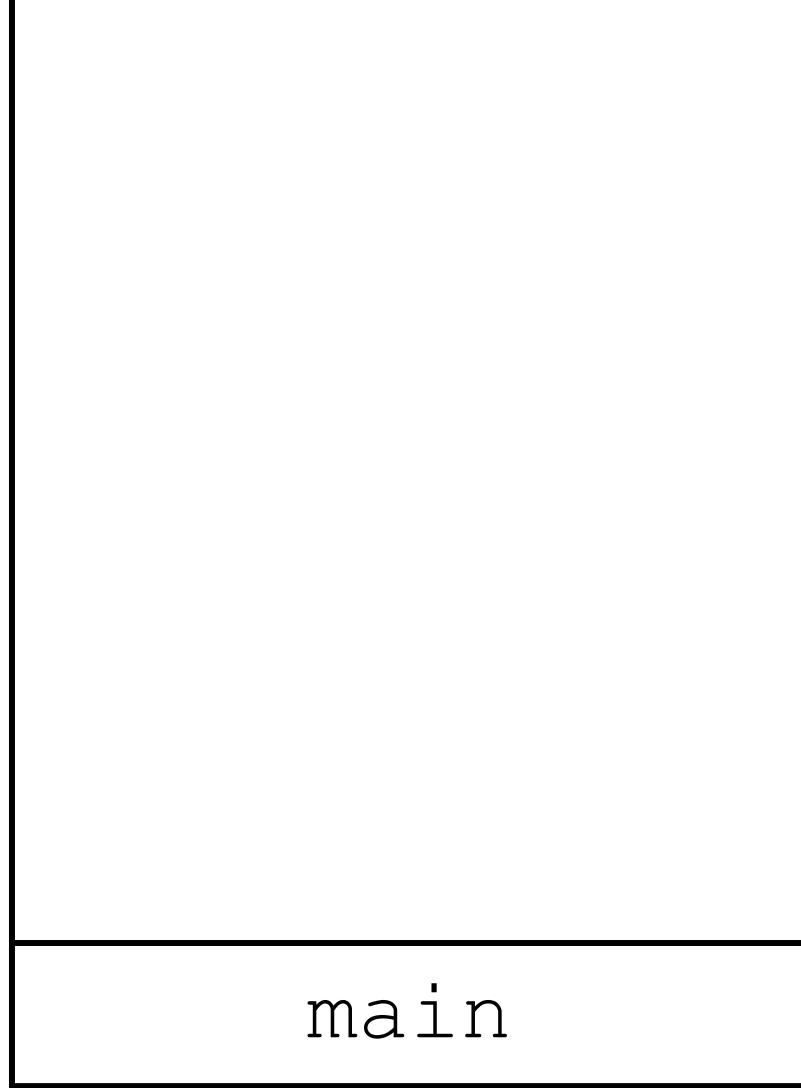




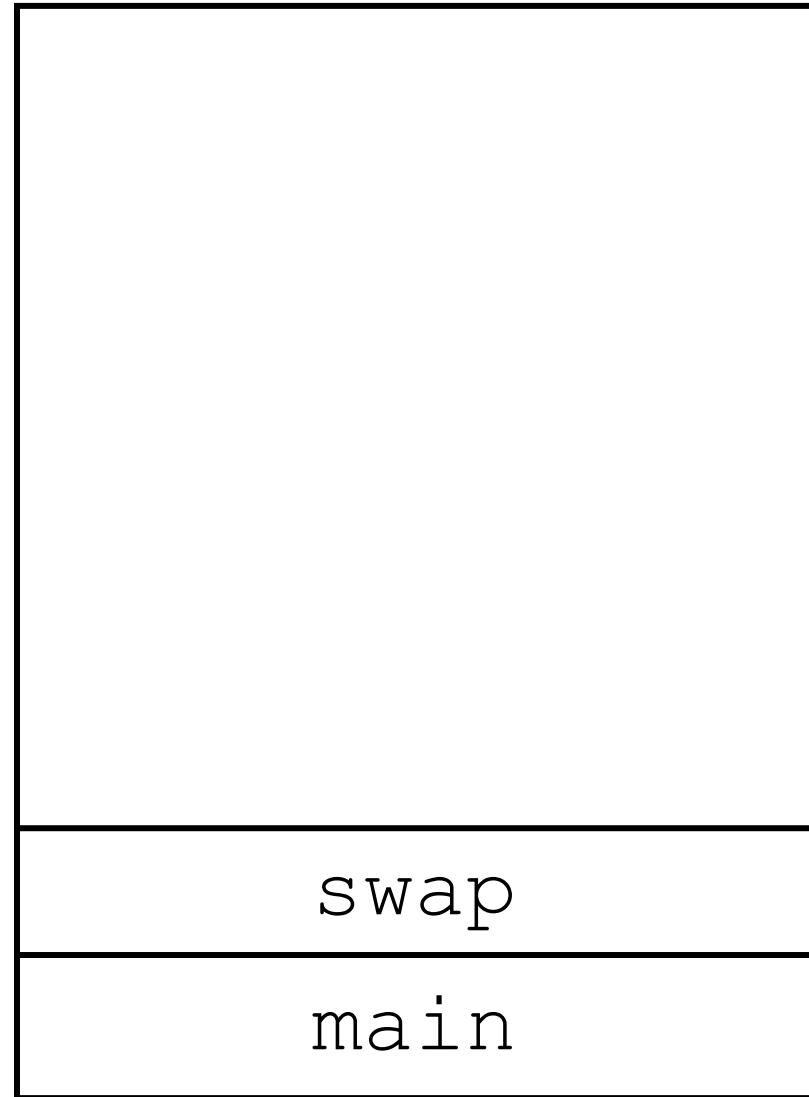


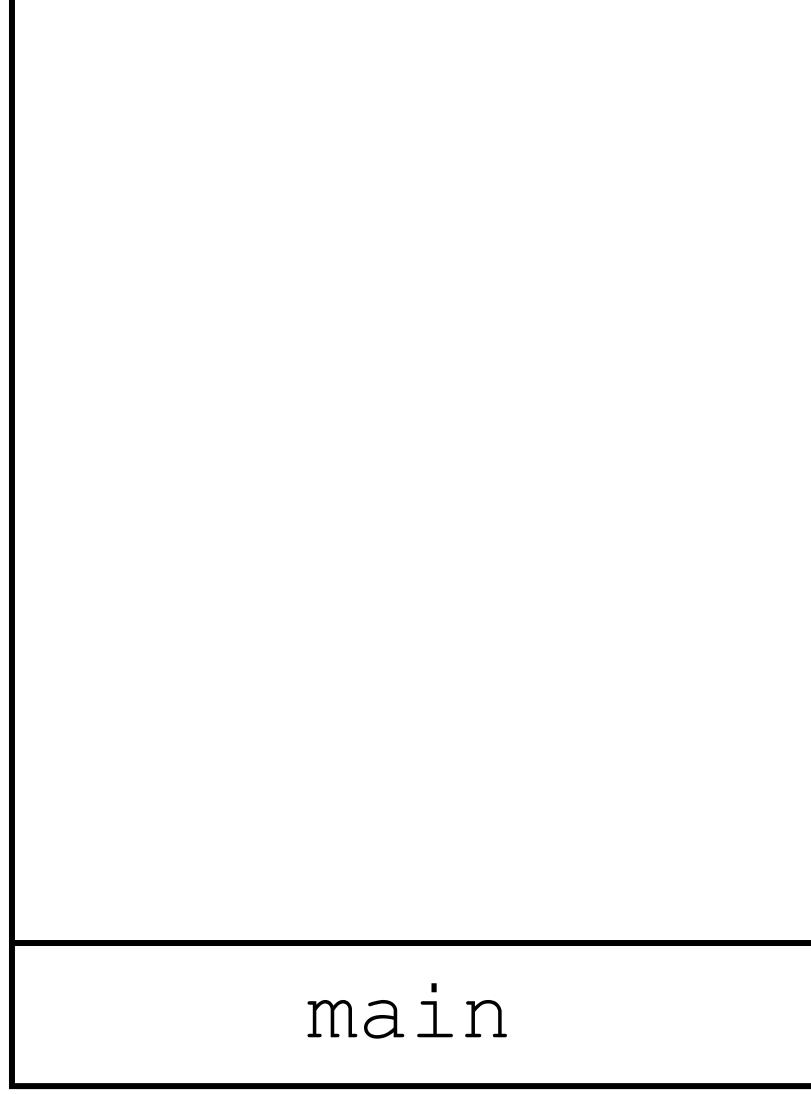
```
// Swap two int values.
void swap(int a, int b)
{
    int temp = a; // store a in temp
    a = b;         // put b into a
    b = temp;       // put temp a into b
}

int main()
{
    int x = 12;
    int y = 33;
    swap(x, y);
    cout << "x == " << x << "  y == " << y << endl; // ?
    return 0;
}
```

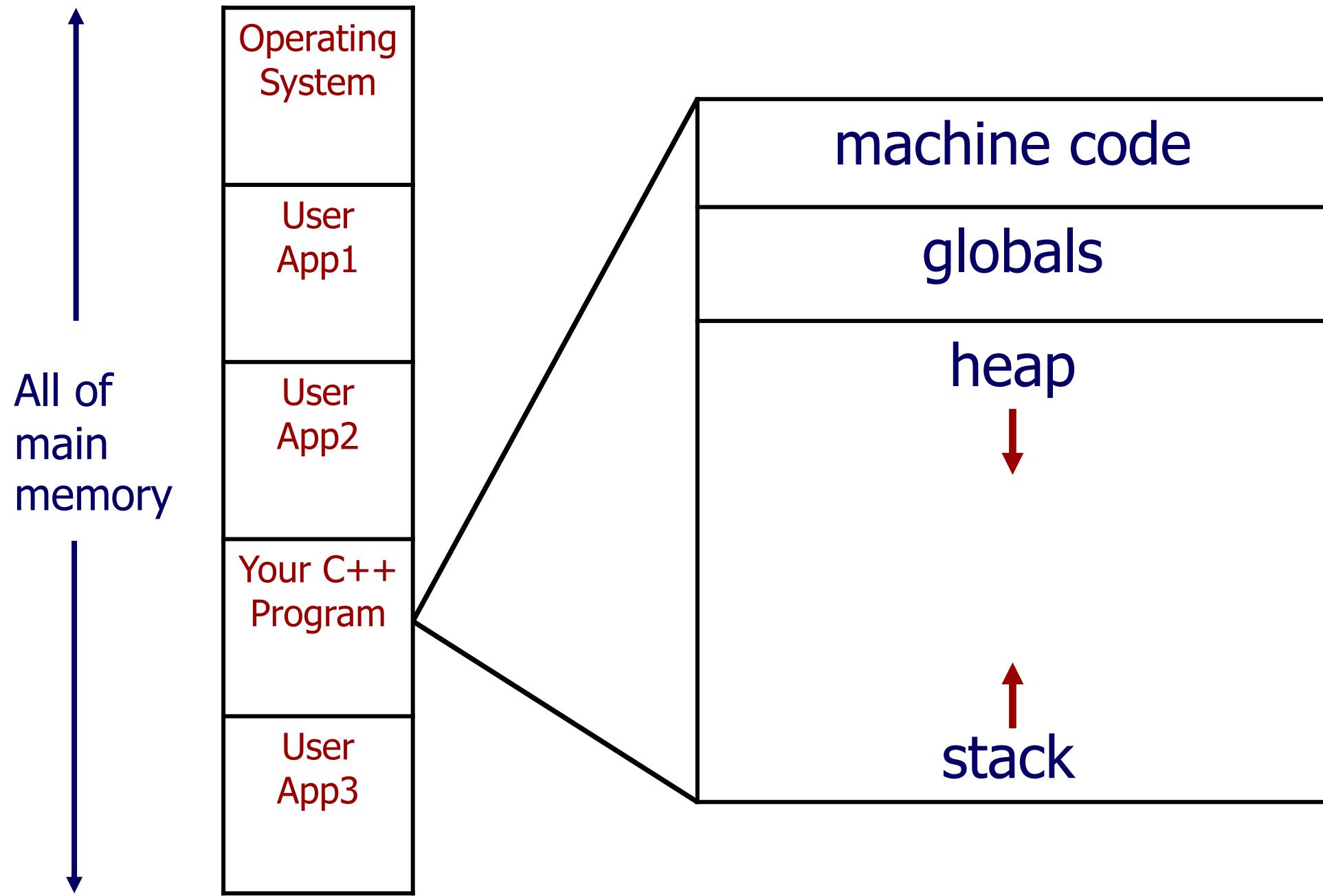


main





main

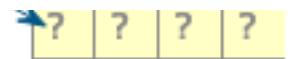
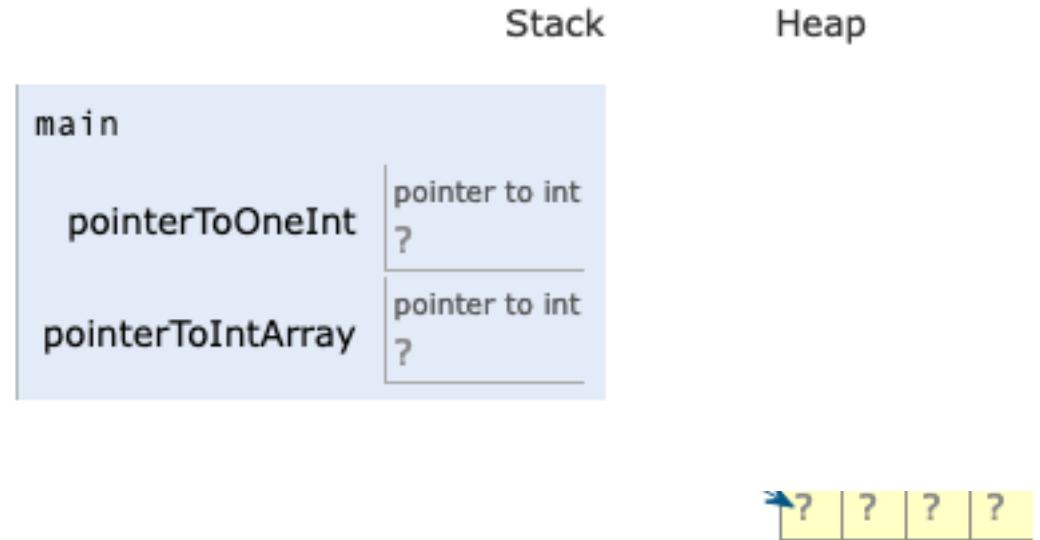


Heap Allocation

- We request memory to be allocated on the heap by using the **new** operator
 - The **new** operator returns a pointer to the allocated memory
 - A pointer value is the address of the **first byte** of the memory
 - A pointer points to an object of a **specified type**
 - A pointer **does not know** how many elements it points to

```
8      // Allocate one int
9      int* pointerToOneInt = new int;
10
11     // Allocate 4 ints (an array of 4 ints)
12     int* pointerToIntArray = new int[4];
13
14     // Allocate one double
15     double* pointerToOneDouble = new double;
16
17     // Allocate 4 doubles (an array of 4 double)
18     double* pointer.ToDoubleArray = new double[4];
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     // Allocate one int
7     int* pointerToOneInt = new int; →
8
9     // Write to the object pointed to by the po:
10    *pointerToOneInt = 50; →
11
12    // Allocate 4 ints (an array of 4 ints)
13    int* pointerToIntArray = new int[4]; →
14
15    // Write to the objects in the array using array notation.
16    pointerToIntArray[0] = 10; →
17    pointerToIntArray[1] = 20; →
18    pointerToIntArray[2] = 30; →
19    pointerToIntArray[3] = 40; →
20
21    return 0;
22 }
```

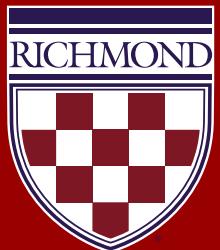


Heap Deallocation

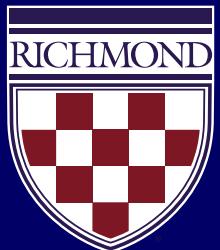
- Since computer memory is limited, we should return memory to the heap once we are finished using it
 - Forgetting to free memory is called a “memory leak”
 - The operator for returning memory to the heap is **delete**
 - We apply **delete** to the pointer returned by **new**

```
6      // Allocate one int
7      int* pointerToOneInt = new int;
8
9      // Free the allocated int
10     delete pointerToOneInt;
11
12     // Allocate 4 ints (an array of 4 ints)
13     int* pointerToIntArray = new int[4];
14
15     // Free the allocated 4 ints
16     delete[] pointerToIntArray;
```

Code Demo

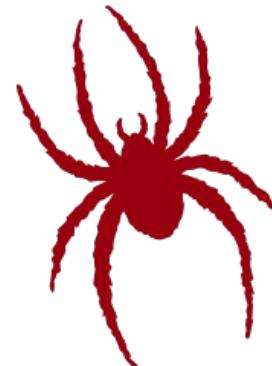


Ask a question



Today

- ~~Code Style~~
- ~~References~~
- ~~Heap (Free Store)~~
- Structs



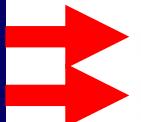
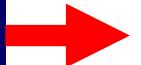
Structs

- A **struct** (short for "structure") is a user-defined data type
 - Groups together variables of different data types under a single name
 - These variables inside a struct are called "members"
 - The members are separated by a semi-colon

```
struct Point3D
{
    double x;
    double y;
    double z;
};
```

```
struct Car
{
    int year;
    string brand;
    string model;
};
```

```
1 #include <iostream>
2 using namespace std;
3
4 // Declare a structure named "Car"
5 struct Car
6 {
7     int year;
8     string brand;
9     string model;
10};
11
12 int main()
13 {
14     // A Car variable (named object).
15     Car dreamCar;
16     dreamCar.year = 1969;           // use . to access fields
17     dreamCar.brand = "Ford";
18     dreamCar.model = "Mustang";
19
20     // Another Car object.
21     Car myCar = {2006, "Honda", "CRV"};
22
23     cout << "Dream car: " << dreamCar.year
24     |           |           |           |
25     |           |           |           |   << " " << dreamCar.brand
26     |           |           |           |   << " " << dreamCar.model << endl;
27
28 }
```



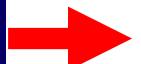
```
1 #include <iostream>
2 using namespace std;
3
4 // Declare a structure named "Car"
5 struct Car
6 {
7     int year;
8     string brand;
9     string model;
10 };
11
12 void printCar(Car carToPrint)
13 {
14     cout << "Car: " << carToPrint.year
15     << " " << carToPrint.brand
16     << " " << carToPrint.model << endl;
17 }
18
19 int main()
20 {
21     // Object of type Car
22     Car myCar = {2006, "Honda", "CRV"};
23
24     printCar(myCar);
25
26     return 0;
27 }
```

Use structs to pass around grouped information

```
1 #include <iostream>
2 using namespace std;
3
4 // Declare a structure named "Car"
5 struct Car
6 {
7     int year;
8     string brand;
9     string model;
10 }
11
12 Car getCorollaByYear(int modelYear)
13 {
14     Car corolla;
15     corolla.year = modelYear;
16     corolla.brand = "Toyota";
17     corolla.model = "Corolla";
18
19     return corolla;
20 }
21
22 int main()
23 {
24     // Get a Toyota
25     Car toyota = getCorollaByYear(2023);
26
27     return 0;
28 }
```

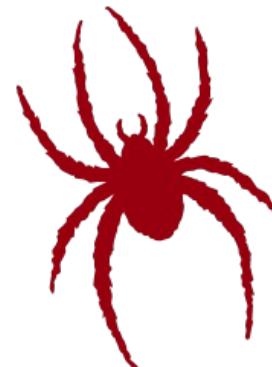
Use structs to
return grouped
information

```
1 #include <iostream>
2 using namespace std;
3
4 // Declare a structure named "Car"
5 struct Car
6 {
7     int year;
8     string brand;
9     string model;
10};
11
12 int main()
13 {
14     // Create a new struct on the heap.
15     // Using a pointer to a Car struct.
16     Car* dreamCar = new Car;
17
18     // Use . to access fields after dereferencing pointer.
19     // Notice the use of the * "contents of" operator.
20     (*dreamCar).year = 1969;
21     (*dreamCar).brand = "Ford";
22     (*dreamCar).model = "Mustang";
23
24     cout << "Dream car: " << (*dreamCar).year
25     << " " << (*dreamCar).brand
26     << " " << (*dreamCar).model << endl;
27
28     return 0;
29 }
```



Today

- ~~Code Style~~
- ~~References~~
- ~~Heap (Free Store)~~
- ~~Structs~~
- In-class exercise



Credits

- Malan CS50 
 - Computer memory image and yellow grid
 - Lecture materials
- Open-AI  DALL·E
 - 3-D rendered garbage can image
- Unsplash.com
 - Image of post office boxes
- PythonTutor.com
 - Images of Stack and Heap