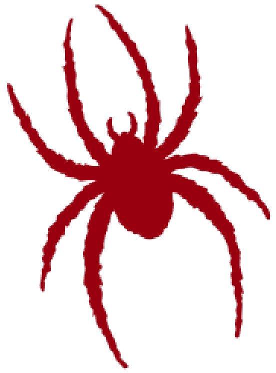
References & Heap

CMSC 240 Software Systems Development

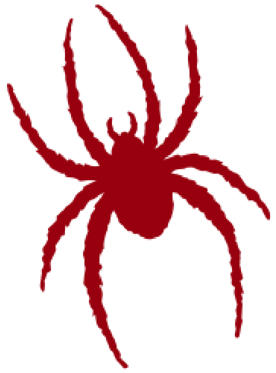
Today

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



Today

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



Only the computer
can read this

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

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

int main(void)
{
for (int number = 1; number < 100; number++)
{
if (number % 3 == 0 && number % 5 == 0){
cout << "FizzBuzz ";
}
else if (number % 3 == 0){
cout << "Fizz ";
}
else if (number % 5 == 0){
cout << "Buzz ";
}
else{
cout << number << " ";
}
}
cout << endl;
return 0;
}
```

Code does not have correct spacing

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

int main(void)
{
    for (int number = 1; number < 100; number++)
    {
        if (number % 3 == 0 && number % 5 == 0)
        {
            cout << "FizzBuzz ";
        }
        else if (number % 3 == 0)
        {
            cout << "Fizz ";
        }
        else if (number % 5 == 0)
        {
            cout << "Buzz ";
        }
        else
        {
            cout << number << " ";
        }
    }
    cout << endl;
    return 0;
}
```

Good style!

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

int main()
{
    float x, y;

    int l = 0;
    int u = 300;
    int s = 20;

    x = l;

    while (x <= u)
    {
        y = (5.0 / 9.0) * (x - 32.0);

        cout << x << "\t" << y << endl;

        x = x + s;
    }

    return 0;
}
```

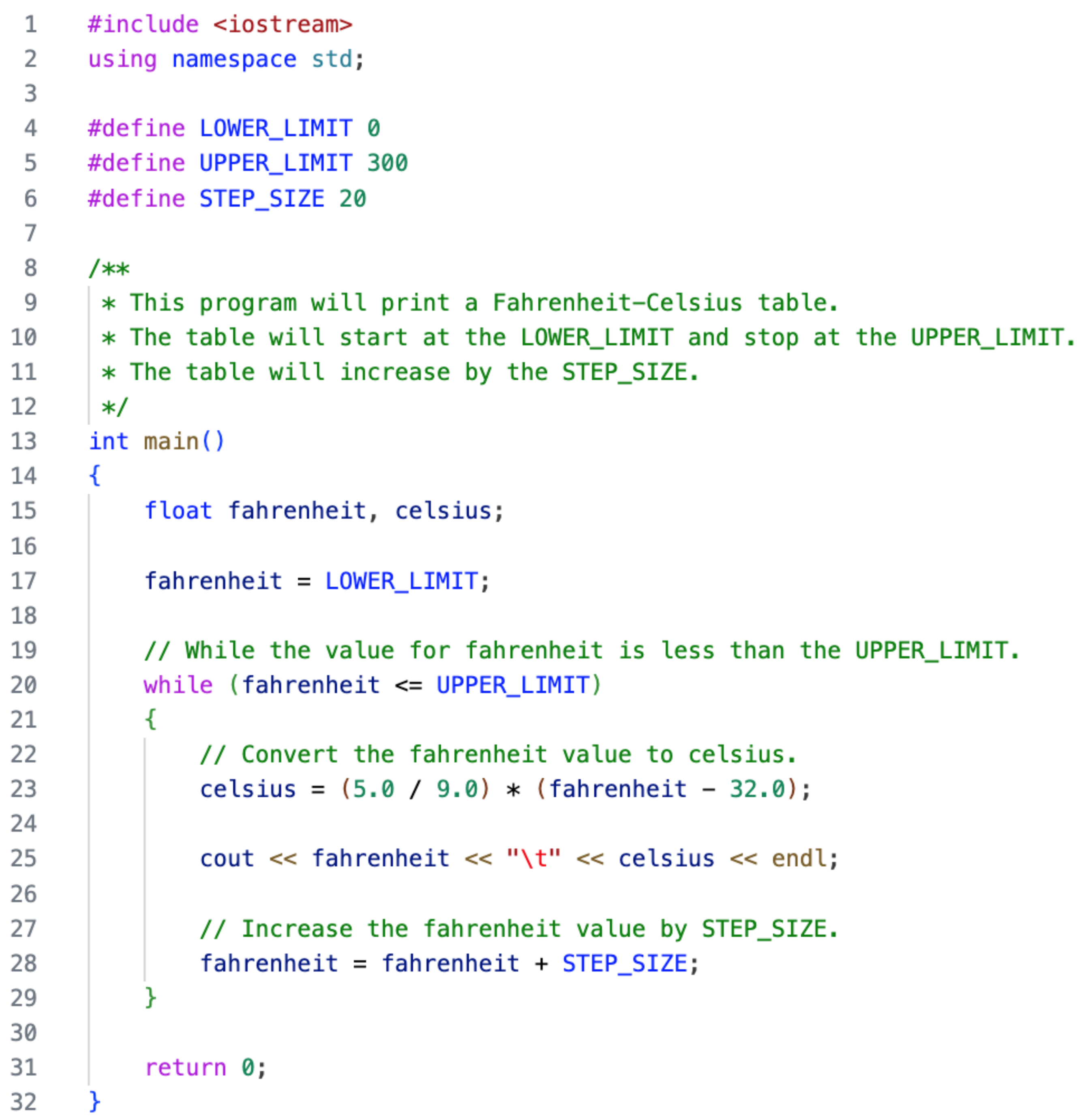
Variable names are not descriptive

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

int main()
{
    float fahrenheit, celsius;

    int lower_limit = 0;
    int upper_limit = 300;
    int step_size = 20;

    fahrenheit = lower_limit;

    while (fahrenheit <= upper_limit)
    {
        celsius = (5.0 / 9.0) * (fahrenheit - 32.0);

        cout << fahrenheit << "\t" << celsius << endl;

        fahrenheit = fahrenheit + step_size;
    }

    return 0;
}
```

Good variable names!

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

#define LOWER_LIMIT 0
#define UPPER_LIMIT 300
#define STEP_SIZE 20

int main()
{
    float fahrenheit, celsius;

    fahrenheit = LOWER_LIMIT;

    while (fahrenheit <= UPPER_LIMIT)
    {
        celsius = (5.0 / 9.0) * (fahrenheit - 32.0);

        cout << fahrenheit << "\t" << celsius << endl;

        fahrenheit = fahrenheit + STEP_SIZE;
    }

    return 0;
}
```

Good use of #define



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

#define LOWER_LIMIT 0
#define UPPER_LIMIT 300
#define STEP_SIZE 20

/**
 * This program will print a Fahrenheit-Celsius table.
 * The table will start at the LOWER_LIMIT and stop at the UPPER_LIMIT.
 * The table will increase by the STEP_SIZE.
 */
int main()
{
    float fahrenheit, celsius;

    fahrenheit = LOWER_LIMIT;

    // While the value for fahrenheit is less than the UPPER_LIMIT.
    while (fahrenheit <= UPPER_LIMIT)
    {
        // Convert the fahrenheit value to celsius.
        celsius = (5.0 / 9.0) * (fahrenheit - 32.0);

        cout << fahrenheit << "\t" << celsius << endl;

        // Increase the fahrenheit value by STEP_SIZE.
        fahrenheit = fahrenheit + STEP_SIZE;
    }

    return 0;
}
```

Good use of comments!

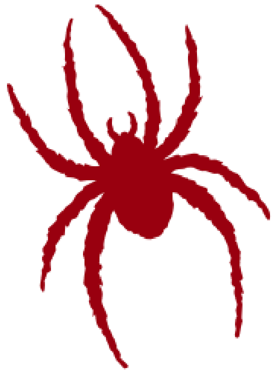
- **Code Quality (30 points)**
  - **Readability (10 points)**
    - Code has meaningful variable and function names: ___/5
    - Code is consistently formatted and indented: ___/5
  - **Modularity (10 points)**
    - Code is appropriately divided into functions: ___/5
    - Each function has a single 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

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


/**
 * This program will print a Fahrenheit-Celsius table.
 * The table will start at the LOWER_LIMIT and stop at the UPPER_LIMIT.
 * The table will increase by the STEP_SIZE.
 */
int main()
{
    // Initialize fahrenheit to LOWER_LIMIT.

    // While the value for fahrenheit is less than the UPPER_LIMIT.

        // Convert the fahrenheit value to celsius.

        // Print out the fahrenheit and celsius values.

        // Increase the fahrenheit value by STEP_SIZE.


    return 0;
}
```

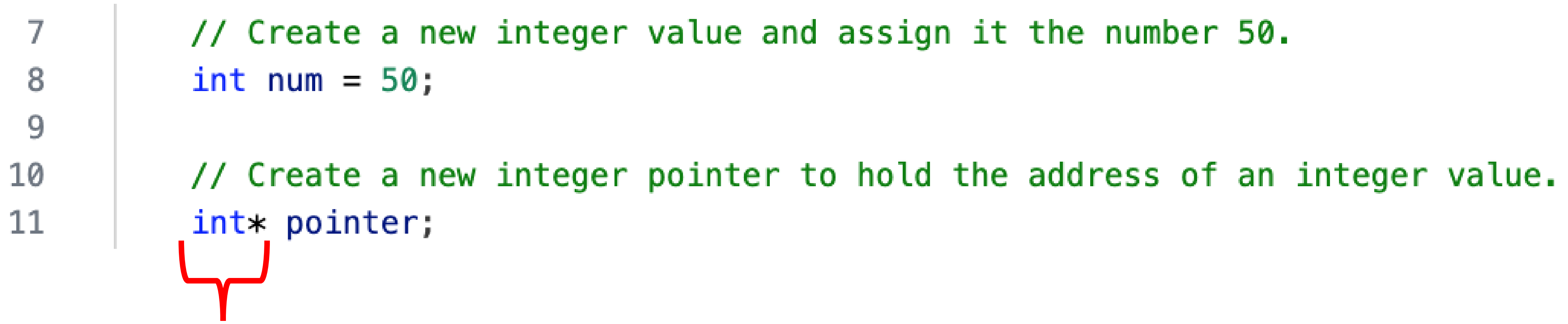
Today

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

- In-class exercise



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



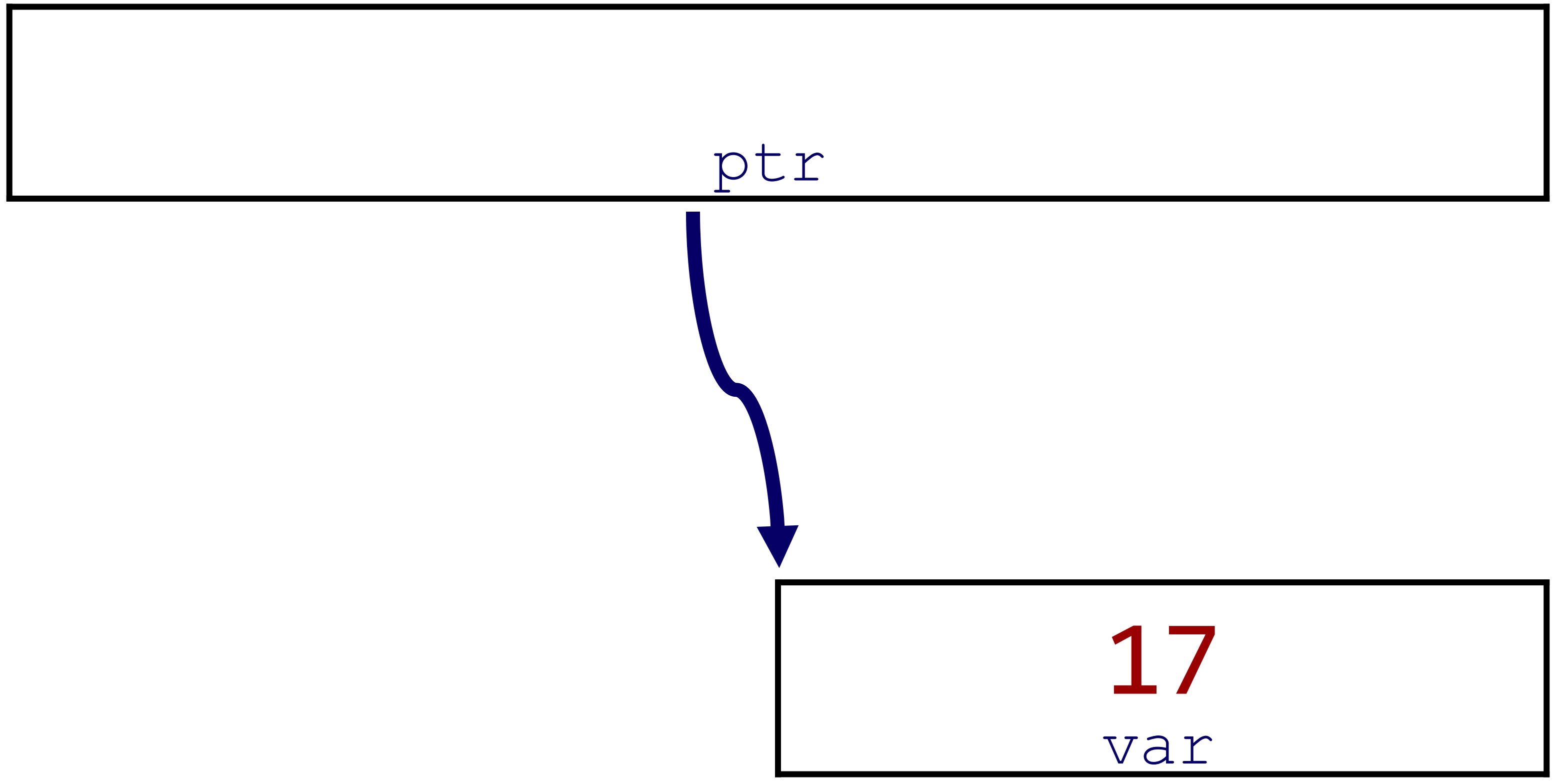
```
// Create a new integer value and assign it the number 50.
int num = 50;

// Create a new integer pointer to hold the address of an integer value.
int* pointer;
```

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

&   “address of”

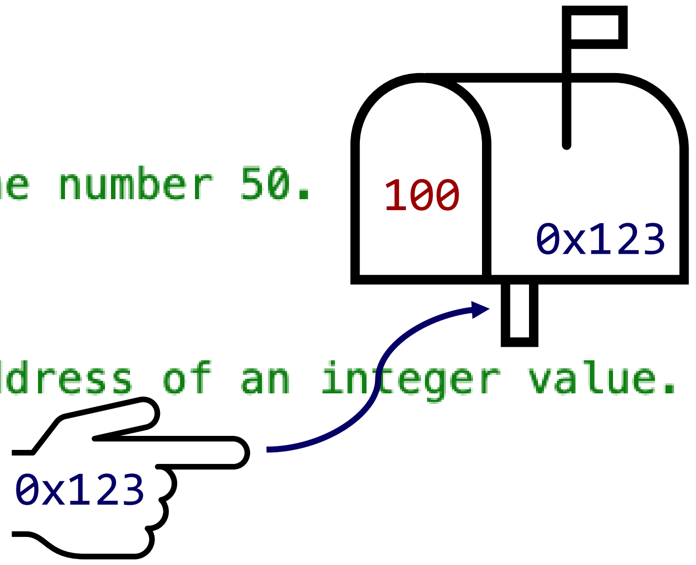
```c
// Create a new integer value and assign it the number 50.
int num = 50;

// Create a new integer pointer to hold the address of an integer value.
int* pointer;

// Put the address of the variable num into
// the pointer using the "address of" operator &.
pointer = &num;
```

The “**address of**” the integer `num`

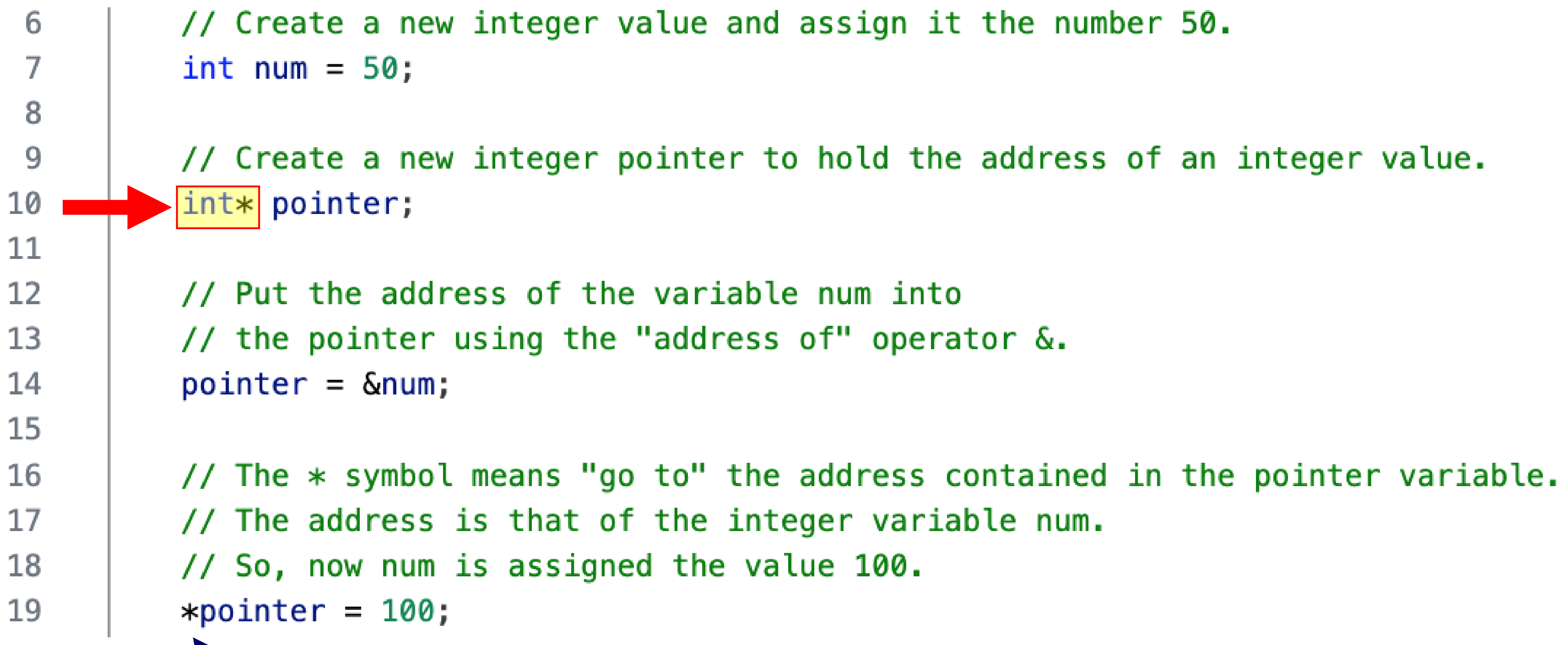

ptr
17
var

* “contents of”

```c
// Create a new integer value and assign it the number 50.
int num = 50;

// Create a new integer pointer to hold the address of an integer value.
int* pointer;

// Put the address of the variable num into
// the pointer using the "address of" operator &.
pointer = &num;

// The * symbol means "go to" the address contained in the pointer variable.
// The address is that of the integer variable num.
// So, now num is assigned the value 100.
*pointer = 100;
```

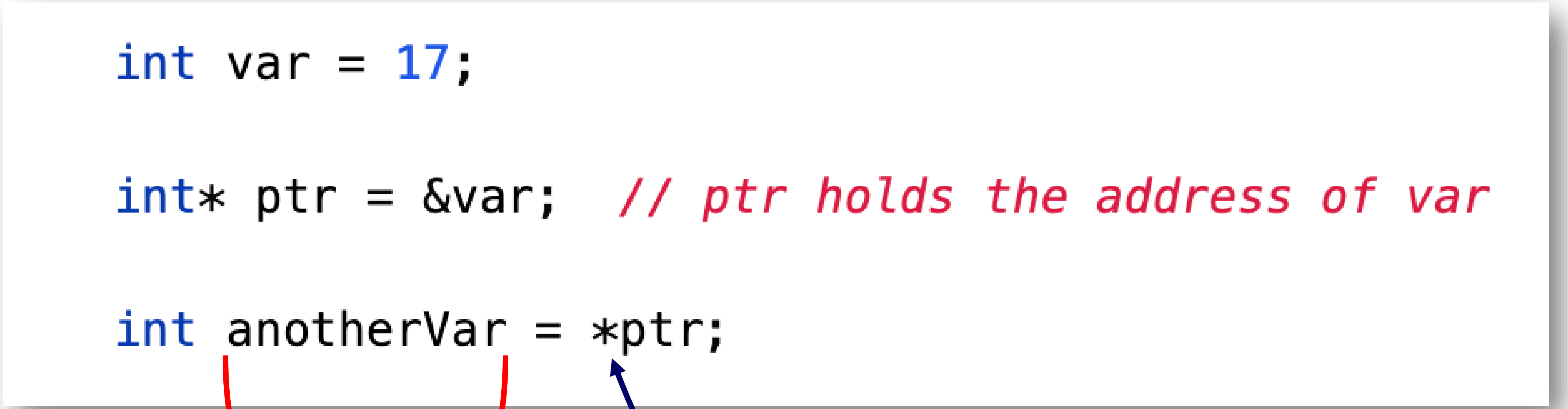


The “**contents of**” the pointer `pointer`
(i.e., the value stored at the address)



```c
// Create a new integer value and assign it the number 50.
int num = 50;

// Create a new integer pointer to hold the address of an integer value.
int* pointer;

// Put the address of the variable num into
// the pointer using the "address of" operator &.
pointer = &num;

// The * symbol means "go to" the address contained in the pointer variable.
// The address is that of the integer variable num.
// So, now num is assigned the value 100.
*pointer = 100;
```

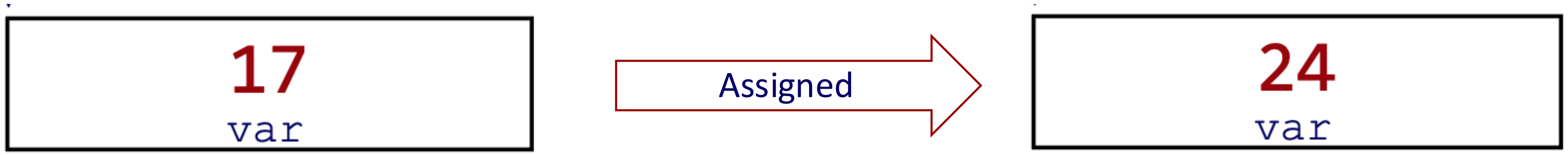
The "**contents of**" the pointer `pointer`
(i.e., the value stored at the address)



```
int var = 17;

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

int anotherVar = *ptr;
```

17

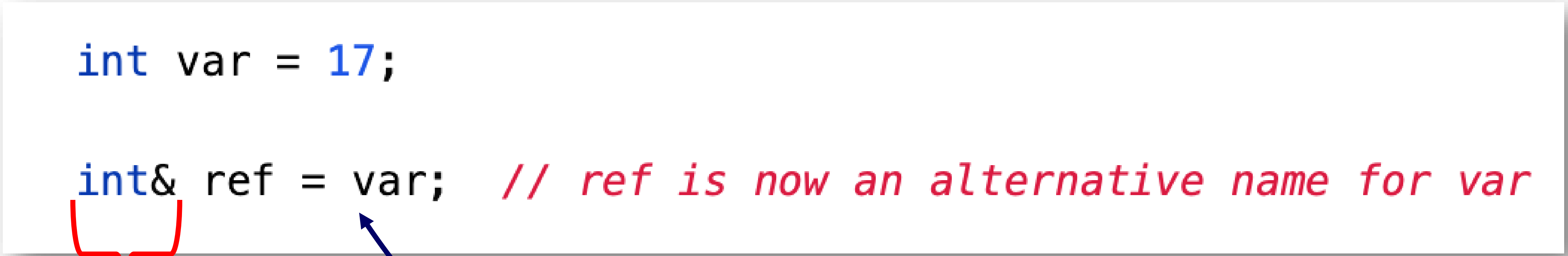
The "**contents of**" the pointer `ptr`
(i.e., the value stored at the address)



```
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`

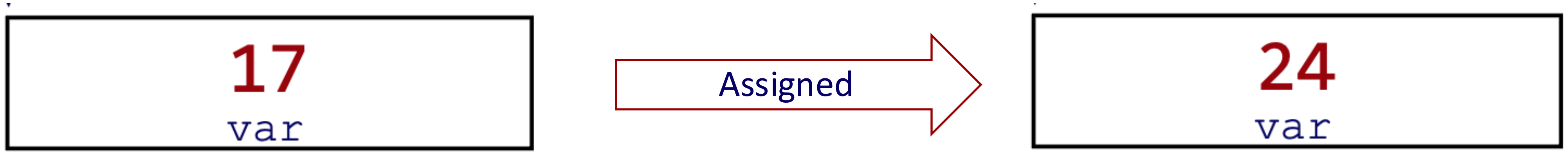
A **reference** is an alternative name for an object.



```
int var = 17;

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

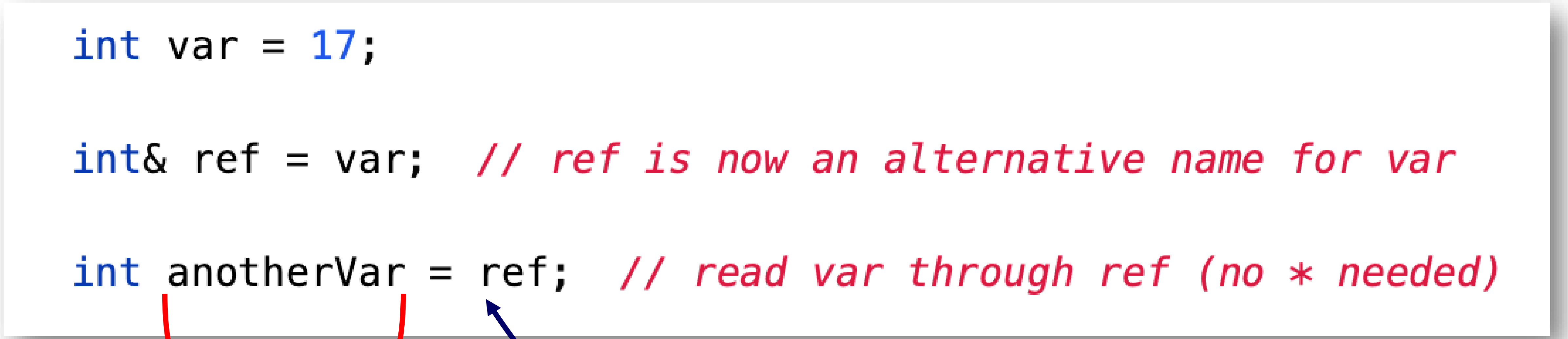
This new type is called an "**int reference**" and is used as an alternative name for an integer variable

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 *



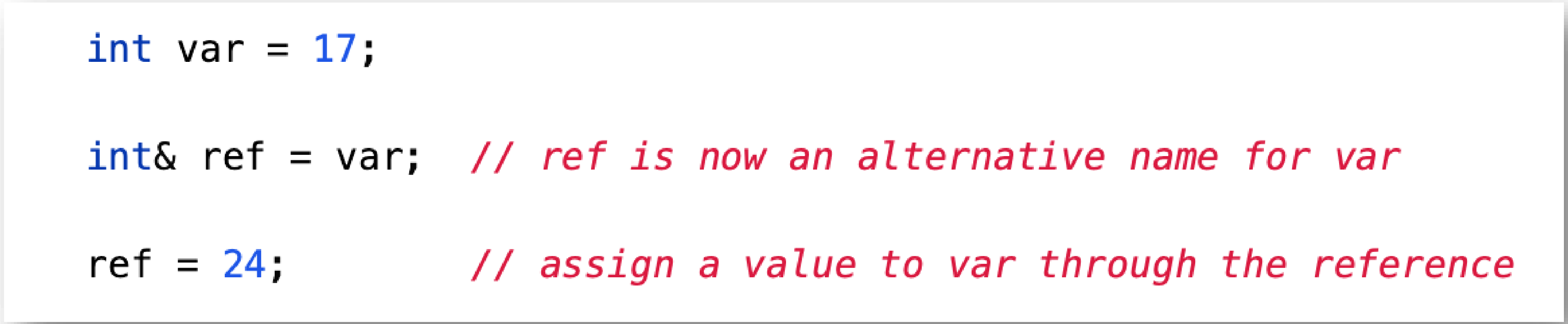
```
int var = 17;

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

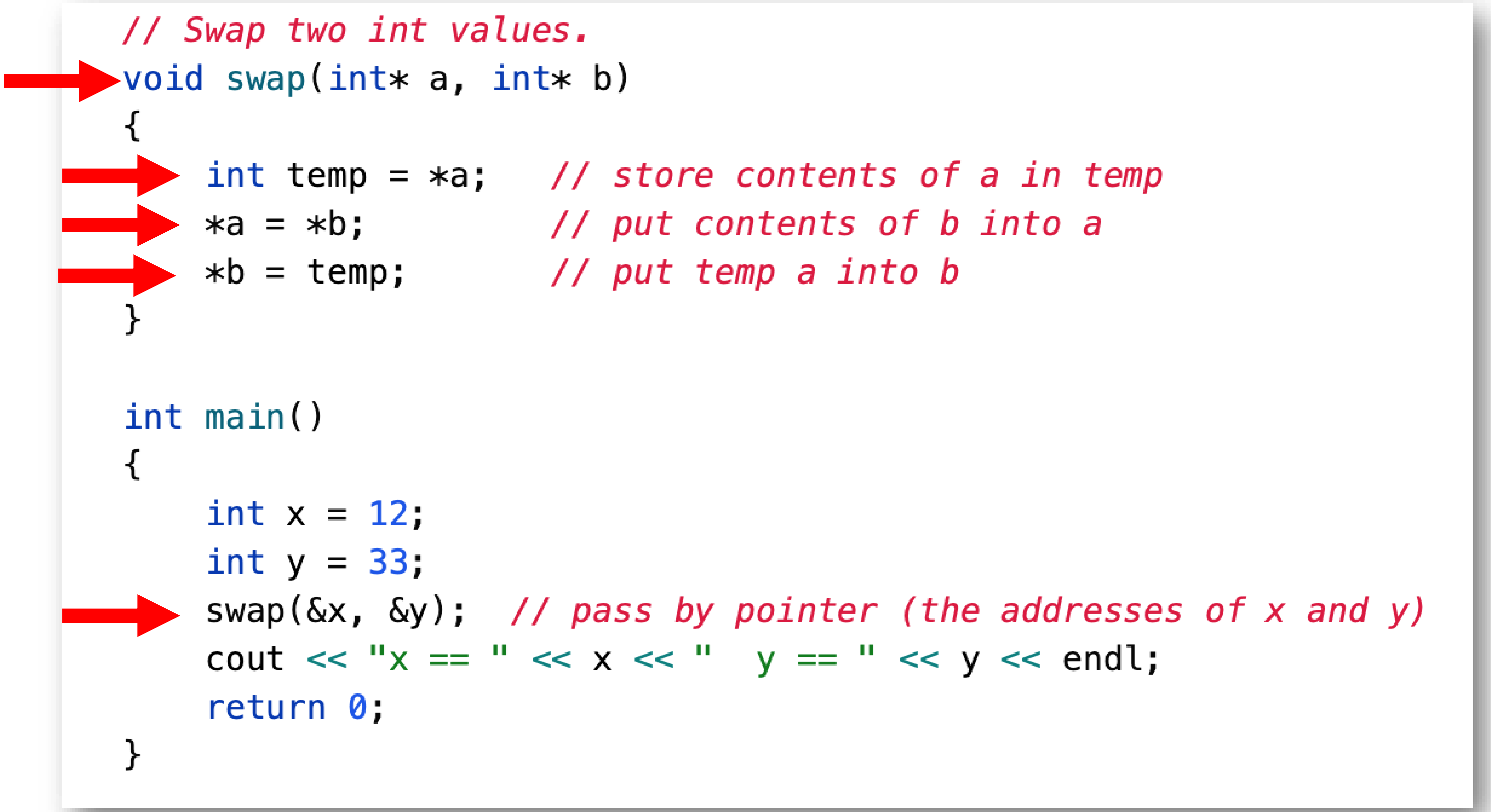
- An alternative name for an object
- Automatically dereferenced pointer
  - They always point to the “**contents of**” the original object
- Immutable
  - You cannot make a reference refer to a different object after initialization



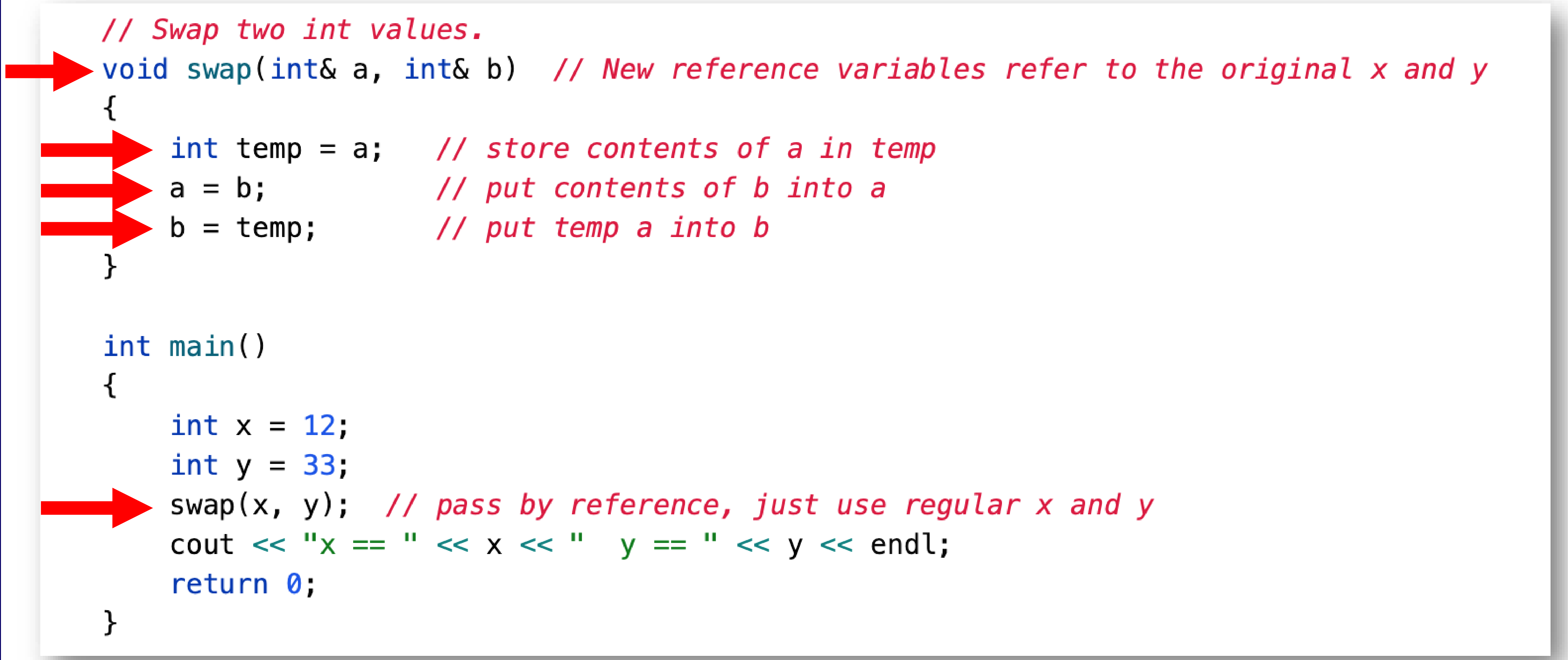
```cpp
int var = 17;

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
```



```cpp
// 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;
}
```



```cpp
// 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;
}
```

```cpp
// 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**:

```cpp
// Pass a copy of the value
int addOne(int x)
{
    // Compute a new value and return it
    return x + 1;
}

// Pass a pointer
void addOne(int* x)
{
    // Dereference pointer and increment the result
    *x = *x + 1;
}

// Pass a reference
void addOne(int& x)
{
    // Increment the value directly using the alternate name
    x = x + 1;
}
```

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

// Pass a copy of the value
float fahrenheitToCelsius(float temperature)
{
    // Compute a new value and return it
    return (5.0 / 9.0) * (temperature - 32.0);
}

int main()
{
    float temperature = 98;

    // Call the fahrenheitToCelsius to convert the temperature.
    temperature = fahrenheitToCelsius(temperature);

    return 0;
}
```

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

// Pass a pointer
void fahrenheitToCelsius(float* temperature)
{
    // Dereference the pointer and compute the result
    *temperature = (5.0 / 9.0) * (*temperature - 32.0);
}

int main()
{
    float temperature = 98;

    // Call the fahrenheitToCelsius to convert the temperature.
    fahrenheitToCelsius(&temperature);

    return 0;
}
```

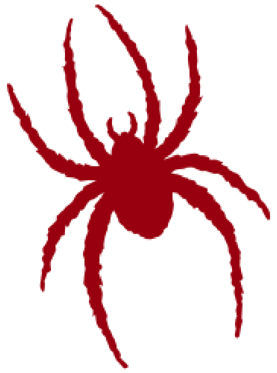
```cpp
#include <iostream>
using namespace std;

// Pass a reference
void fahrenheitToCelsius(float& temperature)
{
    // Compute the value directly using the alternate name
    temperature = (5.0 / 9.0) * (temperature - 32.0);
}

int main()
{
    float temperature = 98;

    // Call the fahrenheitToCelsius to convert the temperature.
    fahrenheitToCelsius(temperature);

    return 0;
}
```

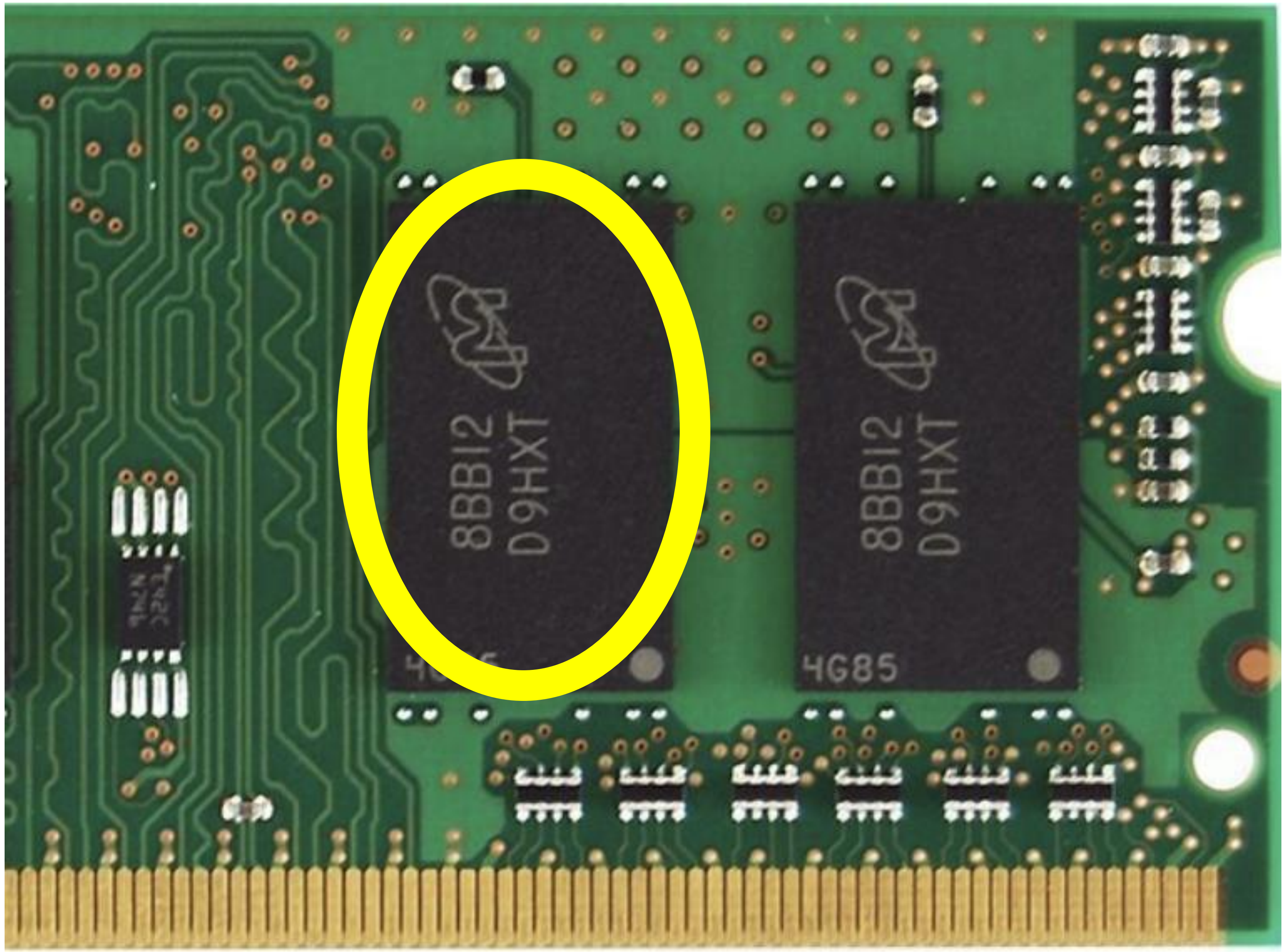
Ask a question

Today

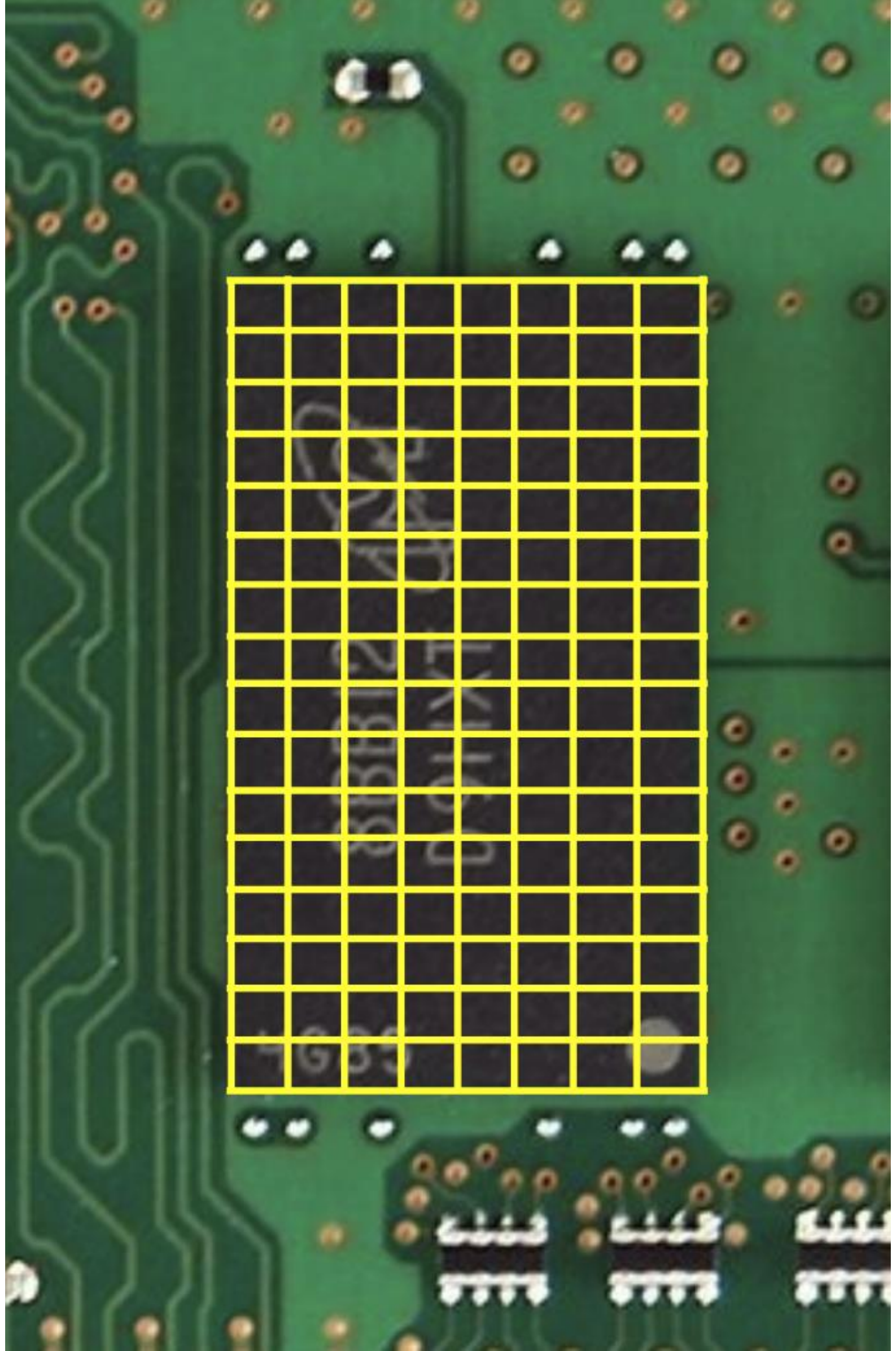
- ~~Code Style~~
- ~~References~~
- Heap (Free Store)
- Structs

- In-class exercise

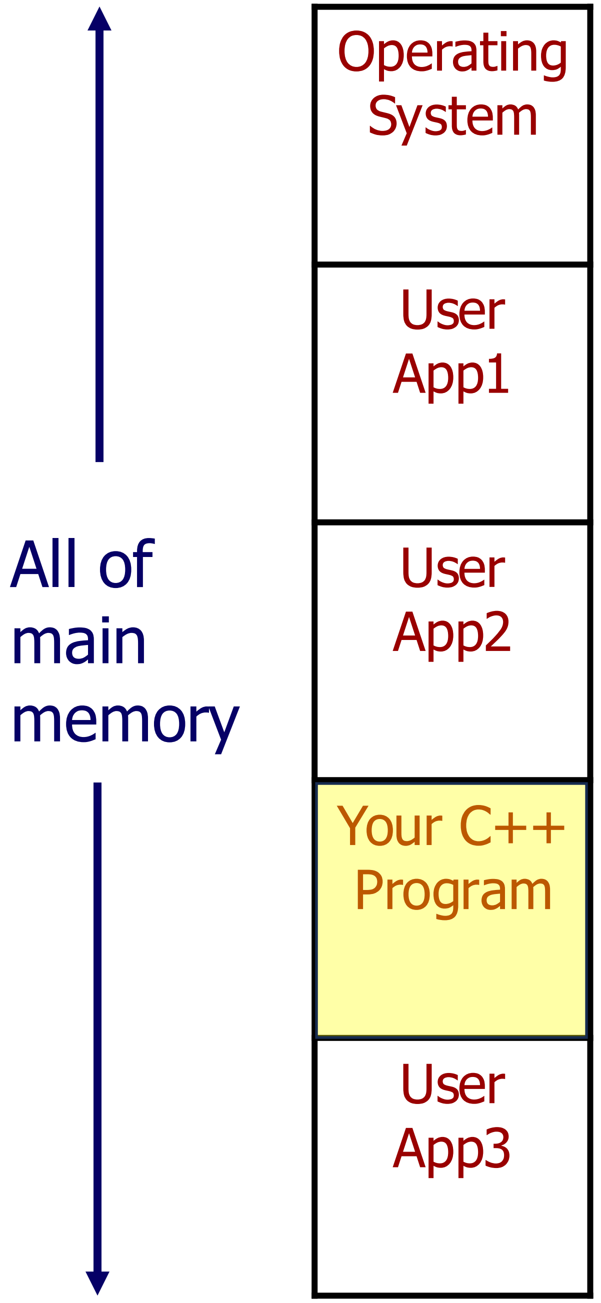


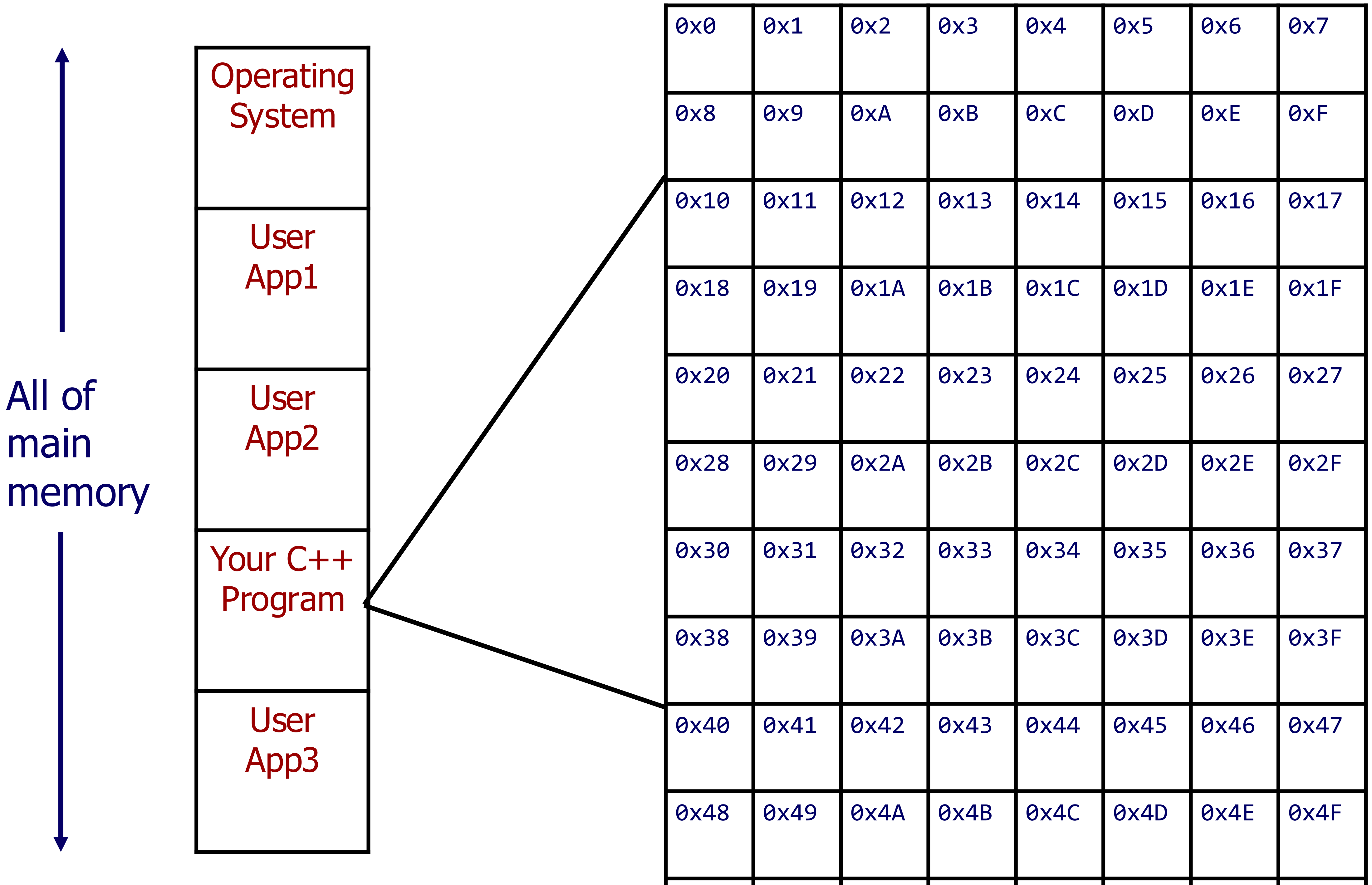

8BBI2
D9HXT
8BBI2
D9HXT
4G85

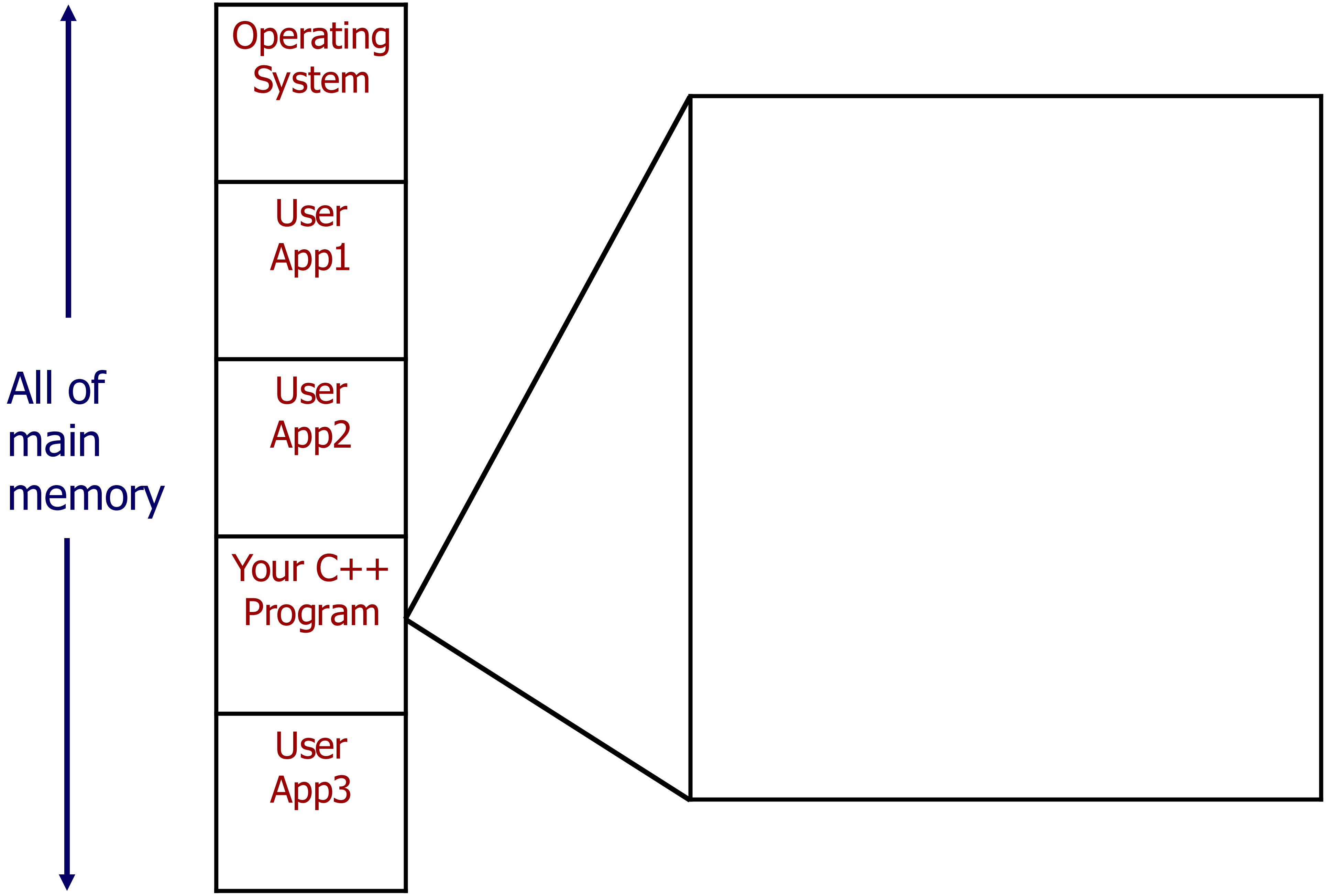
8BB12
D9HXT
4G85

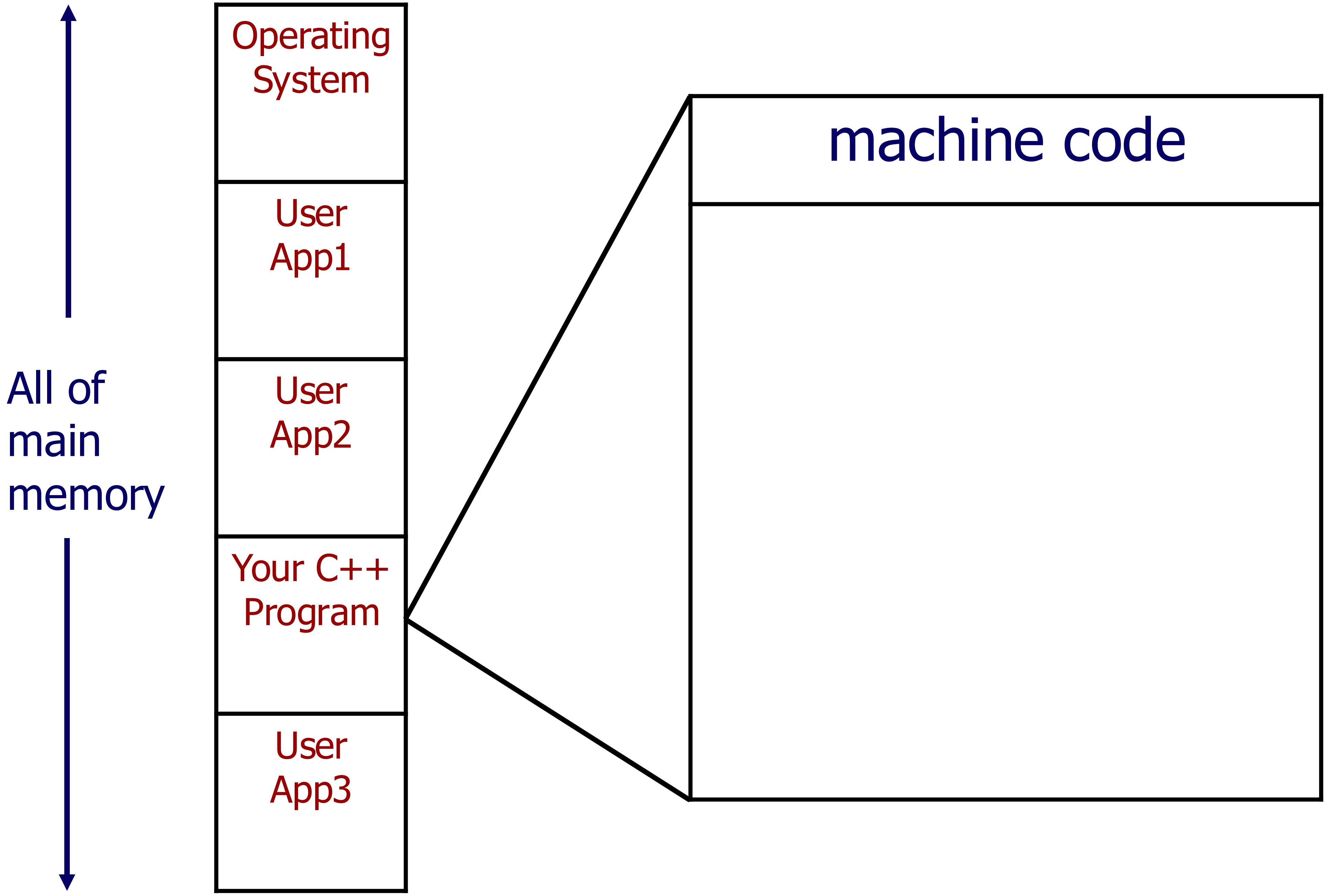


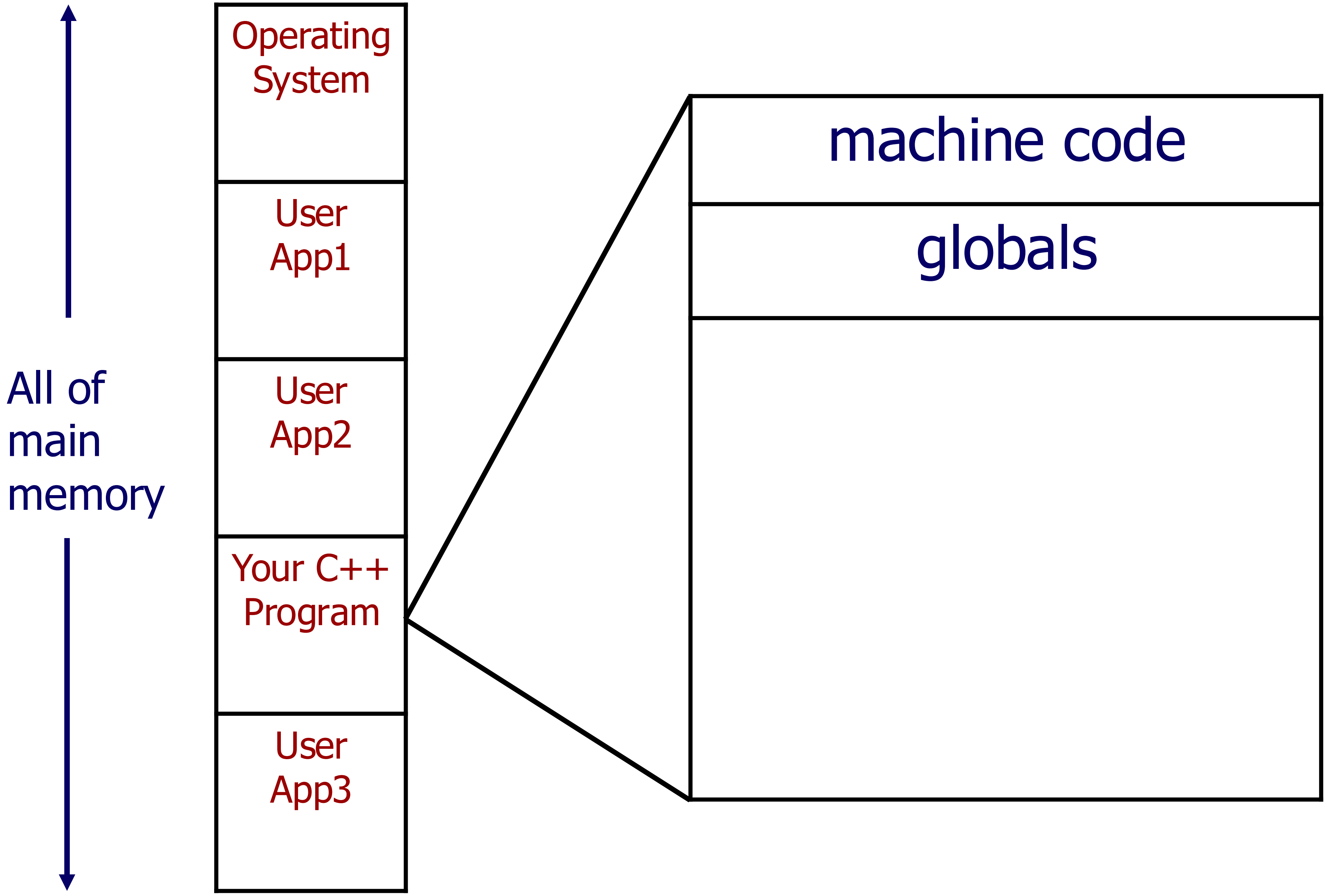
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F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

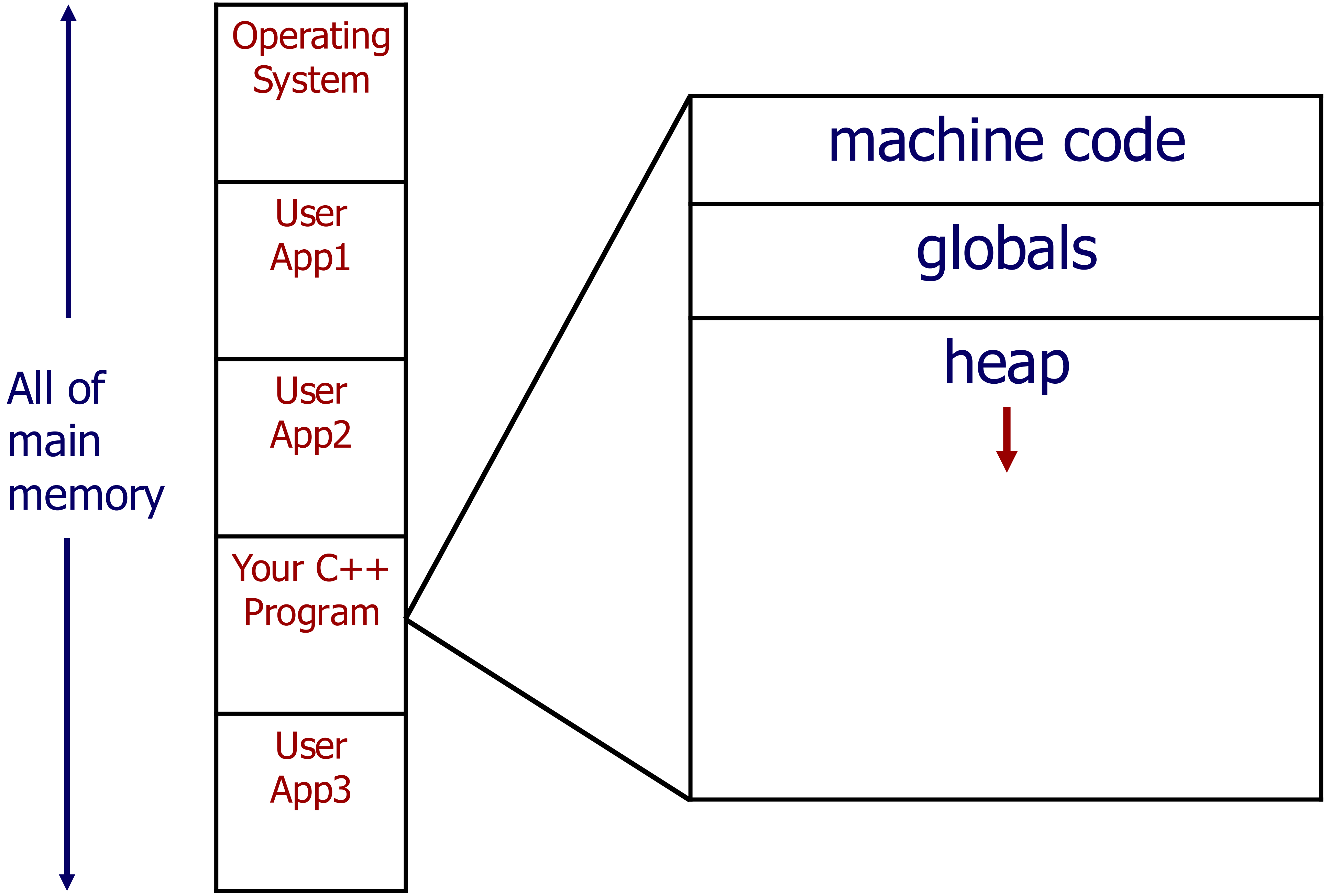

All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3

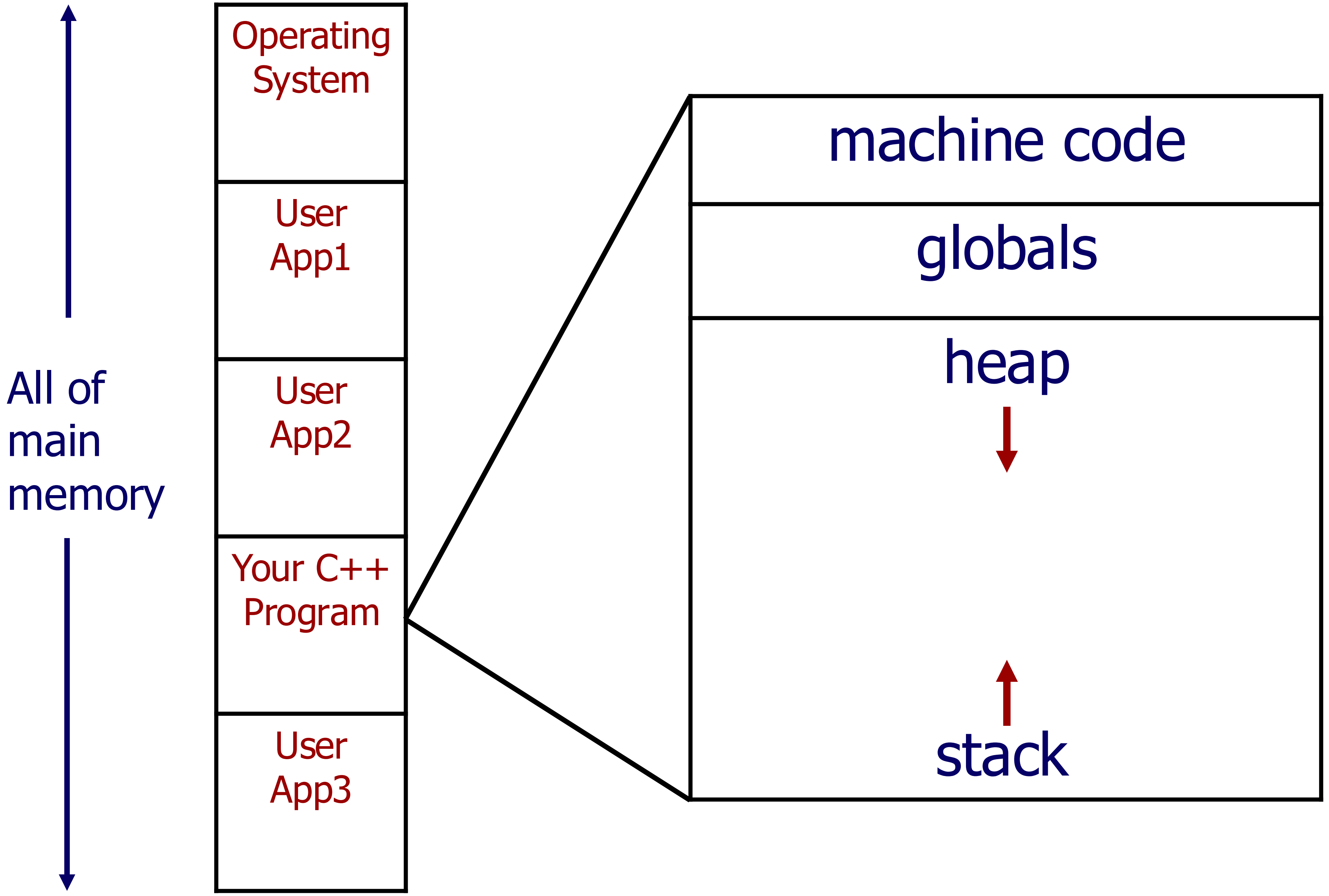

All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7
0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17
0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27
0x28 0x29 0x2A 0x2B 0x2C 0x2D 0x2E 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
Operating System
User App1
User App2
Your C++ Program
User App3

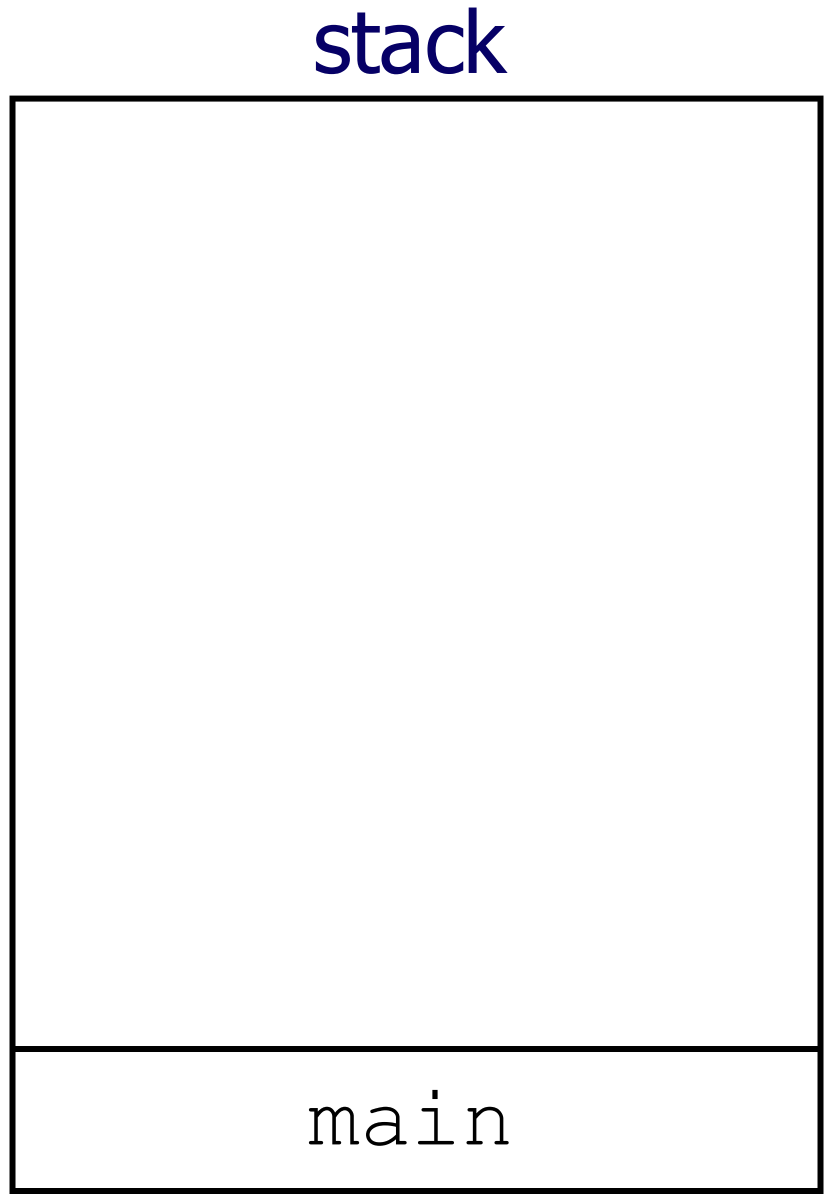

All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
machine code


All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
machine code
globals

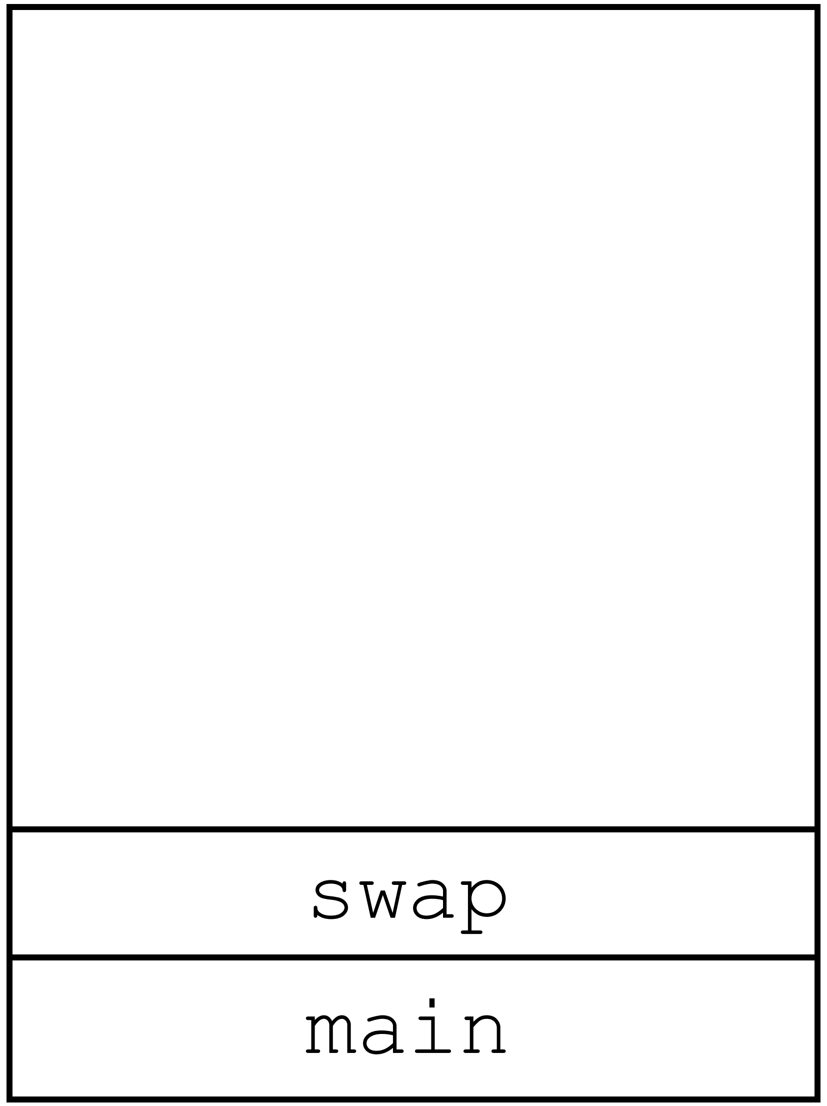

All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
machine code
globals
heap


All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
machine code
globals
heap
stack

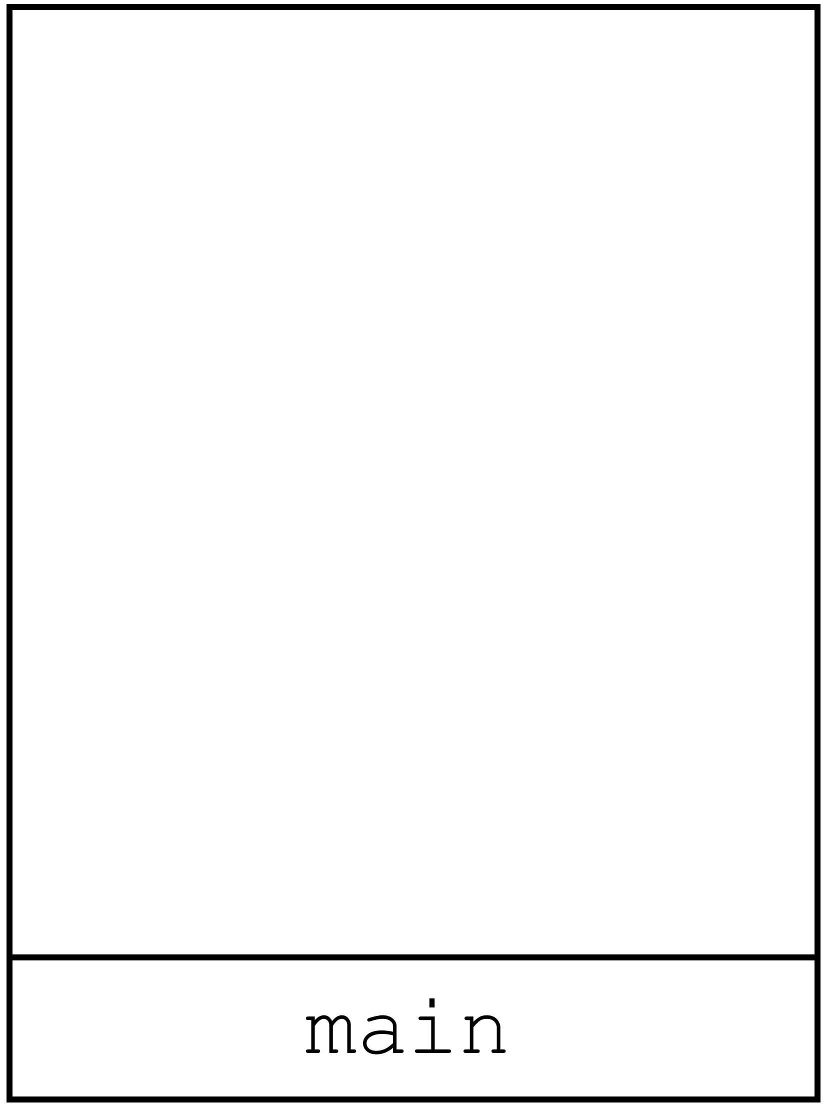
```cpp
// Swap two int values.
void swap(int a, int b)
{
    int temp = a;  // store a in temp
    a = b;         // put b into a
    b = temp;      // put temp a into b
}

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

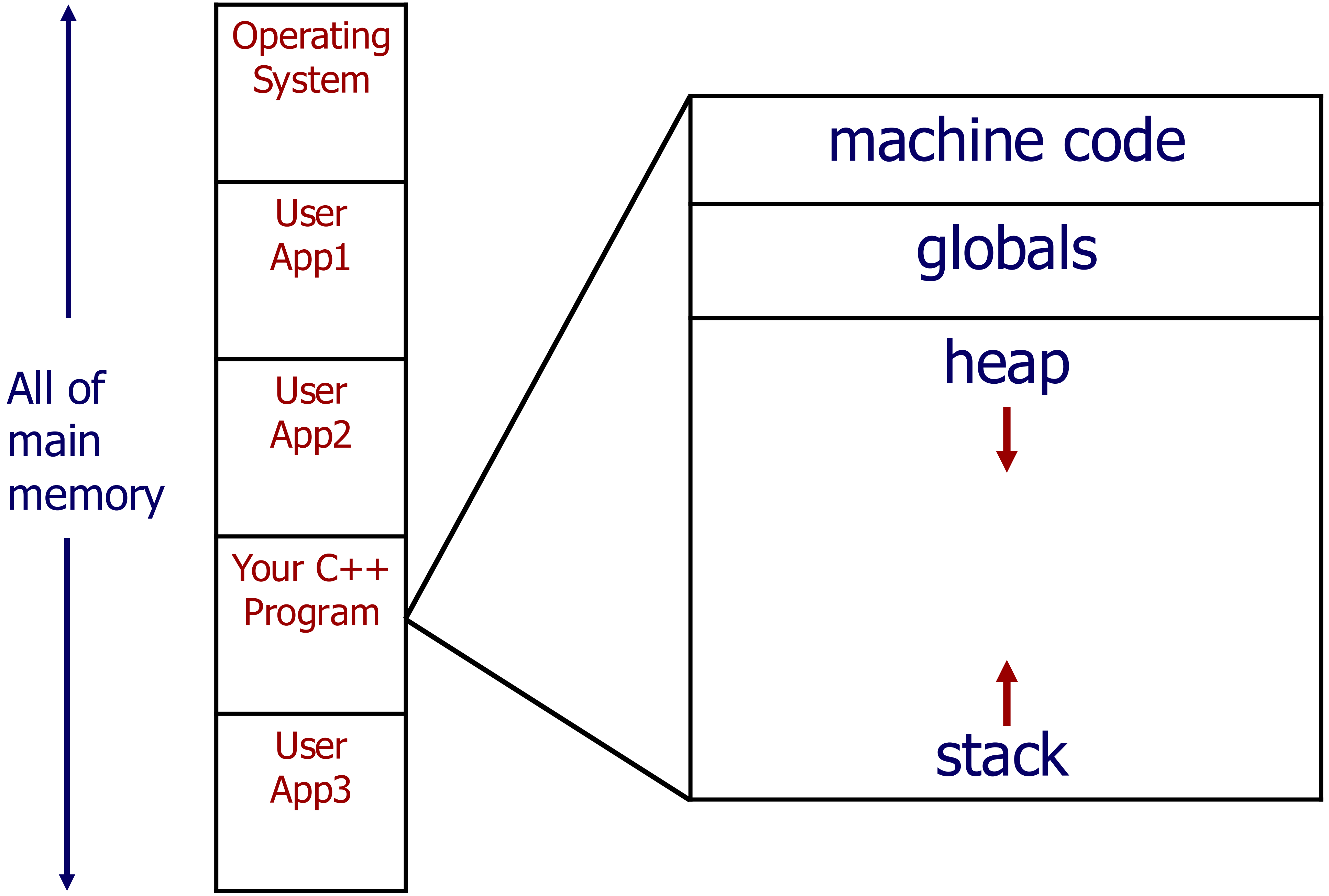

stack
main

stack



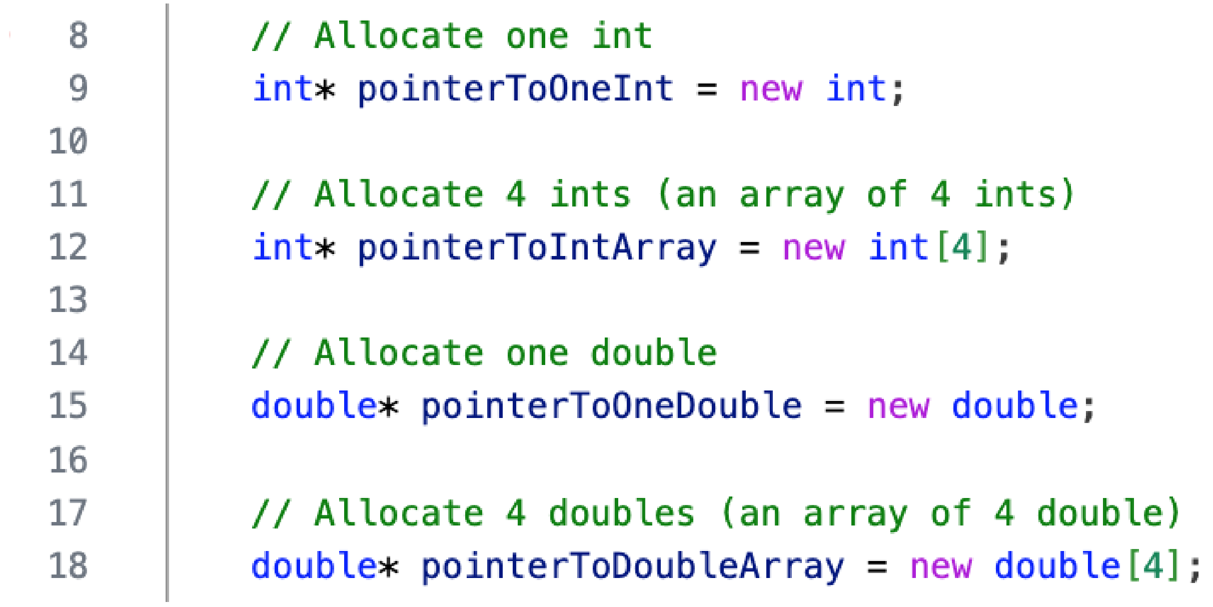
stack




All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
machine code
globals
heap
stack

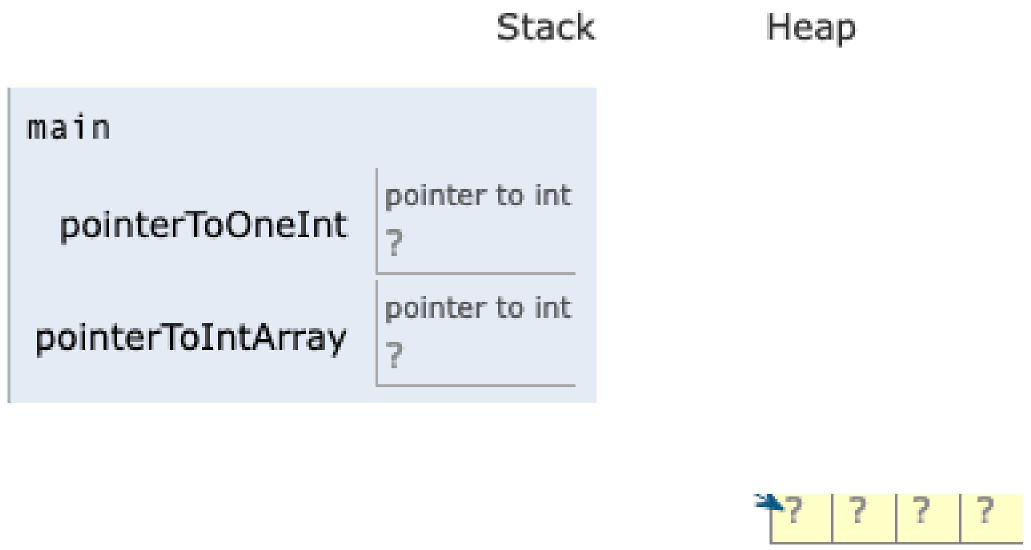
Heap Allocation

- 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



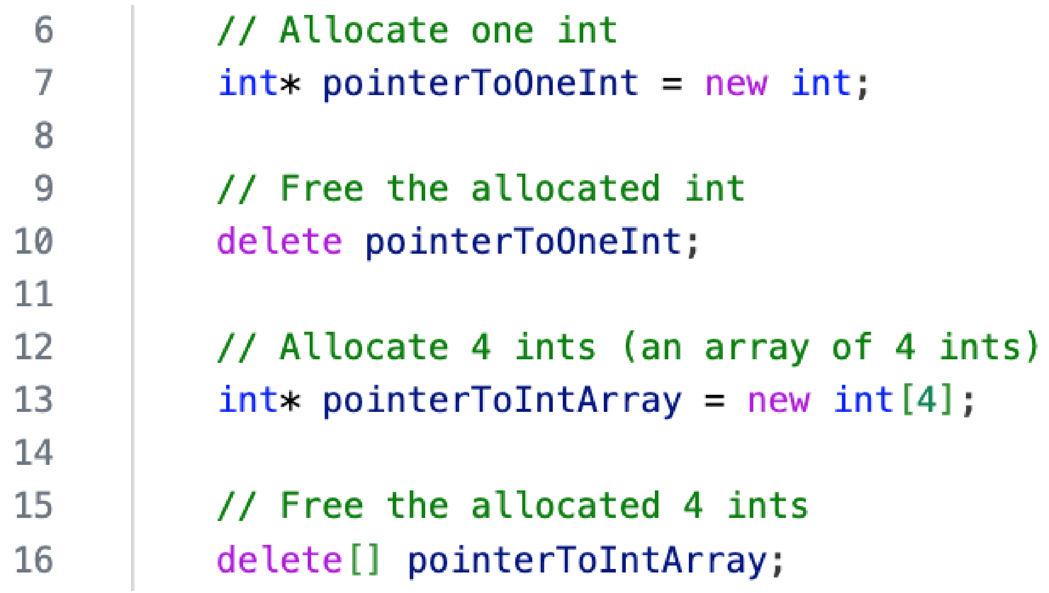
```cpp
    // Allocate one int
    int* pointerToOneInt = new int;

    // Allocate 4 ints (an array of 4 ints)
    int* pointerToIntArray = new int[4];

    // Allocate one double
    double* pointerToOneDouble = new double;

    // Allocate 4 doubles (an array of 4 double)
    double* pointerToDoubleArray = new double[4];
```

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

int main()
{
    // Allocate one int
    int* pointerToOneInt = new int;

    // Write to the object pointed to by the poi
    *pointerToOneInt = 50;

    // Allocate 4 ints (an array of 4 ints)
    int* pointerToIntArray = new int[4];

    // Write to the objects in the array using array notation.
    pointerToIntArray[0] = 10;
    pointerToIntArray[1] = 20;
    pointerToIntArray[2] = 30;
    pointerToIntArray[3] = 40;

    return 0;
}
```



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`**



```cpp
// Allocate one int
int* pointerToOneInt = new int;

// Free the allocated int
delete pointerToOneInt;

// Allocate 4 ints (an array of 4 ints)
int* pointerToIntArray = new int[4];

// Free the allocated 4 ints
delete[] pointerToIntArray;
```

Why Use the Heap?

- **Dynamic Memory Allocation (Size and Lifetime Flexibility)**

  - **Heap variables have a flexible size**
    - The heap allows you to allocate memory during runtime based on dynamic requirements.

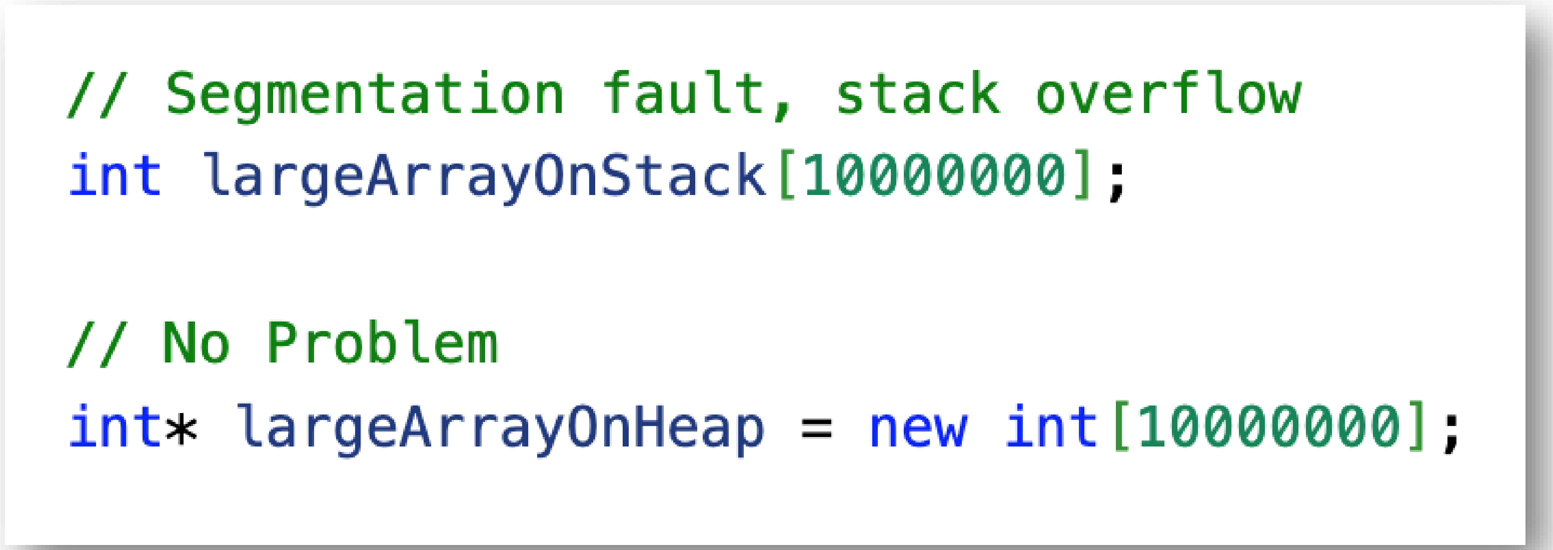
  - **Heap variables persist until explicitly freed**
    - Unlike stack variables, whose lifetime is tied to the scope in which they're declared, heap variables remain in memory until they are explicitly deallocated using delete.
      - Useful when a variable needs to outlive the function where it was created.

Why Use the Heap?

- **Avoiding Stack Limitations**
  - **Stack size is limited:**
    - The stack has a relatively small, fixed size (often a few megabytes). If you allocate large objects or arrays on the stack, you risk a stack overflow.

❌

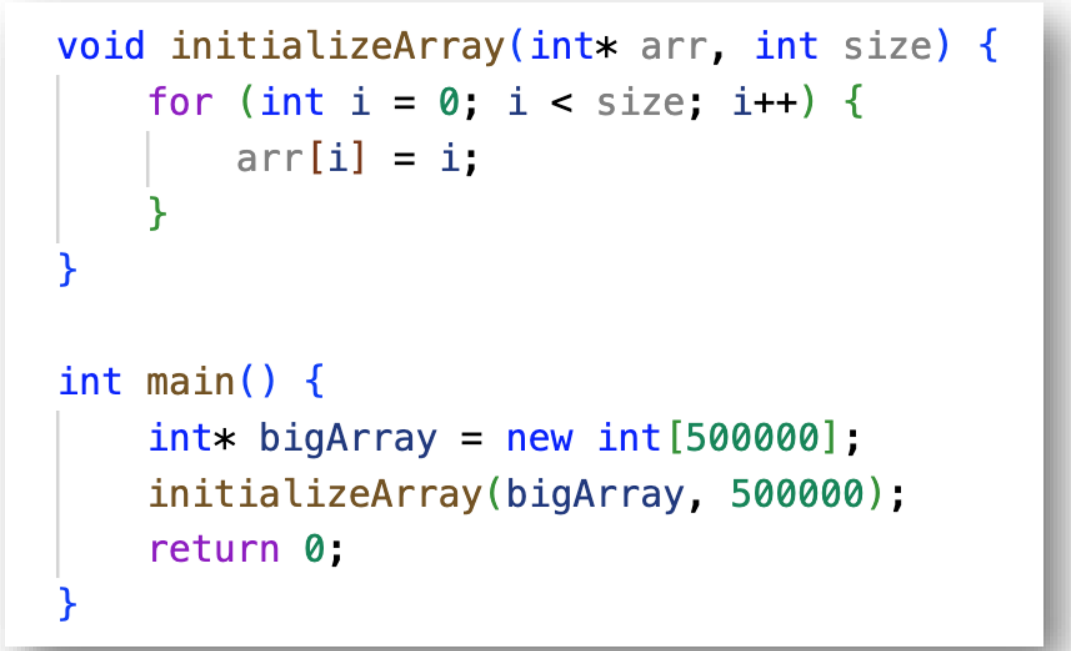
✓



```cpp
// Segmentation fault, stack overflow
int largeArrayOnStack[10000000];

// No Problem
int* largeArrayOnHeap = new int[10000000];
```

Why Use the Heap?

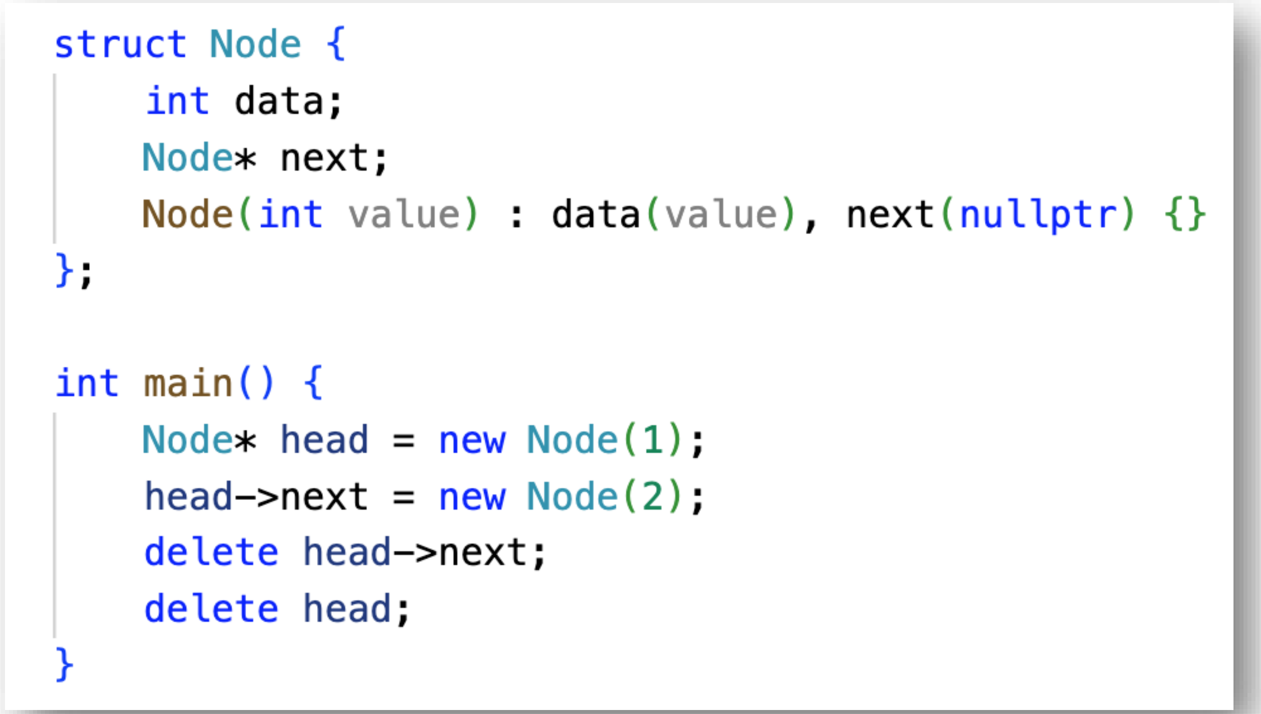
- **Sharing Memory Between Scopes**
  - **Heap memory is accessible from multiple scopes**
    - Variables allocated on the heap are not tied to any particular scope, making it easier to share memory between functions, objects, or threads.



```
void initializeArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;
    }
}

int main() {
    int* bigArray = new int[500000];
    initializeArray(bigArray, 500000);
    return 0;
}
```

Why Use the Heap?

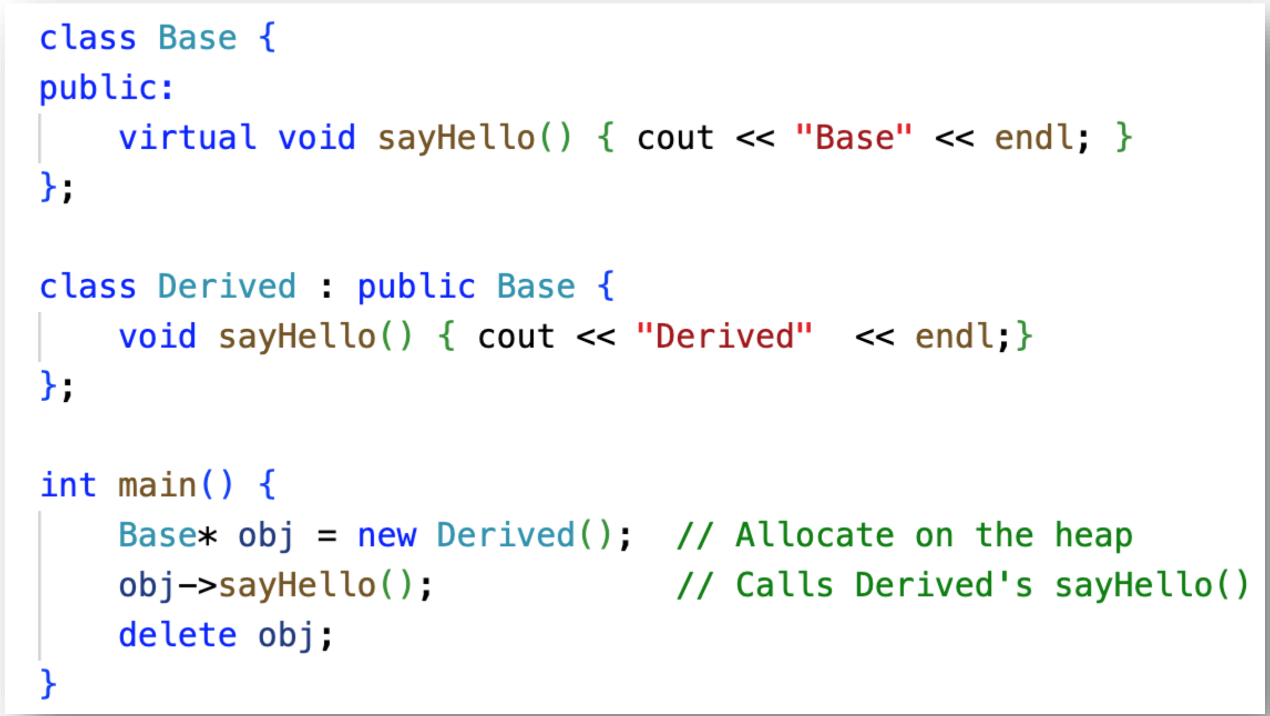
- **Dynamic Data Structures**
  - Data structures like linked lists, trees, graphs, and other dynamically growing data structures require memory allocation during runtime, which makes heap allocation essential.



```
struct Node {
    int data;
    Node* next;
    Node(int value) : data(value), next(nullptr) {}
};

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    delete head->next;
    delete head;
}
```

Why Use the Heap?

- **Object-Oriented Programming (Polymorphism)**
  - **Heap allocation is used for polymorphic behavior**
    - In object-oriented programming, objects are often created on the heap to allow for runtime polymorphism.



```cpp
class Base {
public:
    virtual void sayHello() { cout << "Base" << endl; }
};

class Derived : public Base {
    void sayHello() { cout << "Derived"  << endl;}
};

int main() {
    Base* obj = new Derived();  // Allocate on the heap
    obj->sayHello();            // Calls Derived's sayHello()
    delete obj;
}
```

When Not to Use the Heap

- **Performance cost**
  - Allocating and deallocating memory on the heap is slower than on the stack because it requires interacting with the operating system or memory manager.
- **Risk of memory leaks**
  - If you allocate memory on the heap and forget to free it using delete, you create a memory leak.
- **Fragmentation**
  - Over time, frequent heap allocations and deallocations can lead to memory fragmentation, reducing performance.

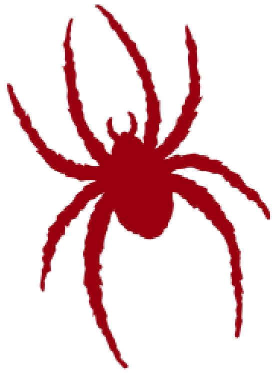
Code Demo

Ask a question

Today

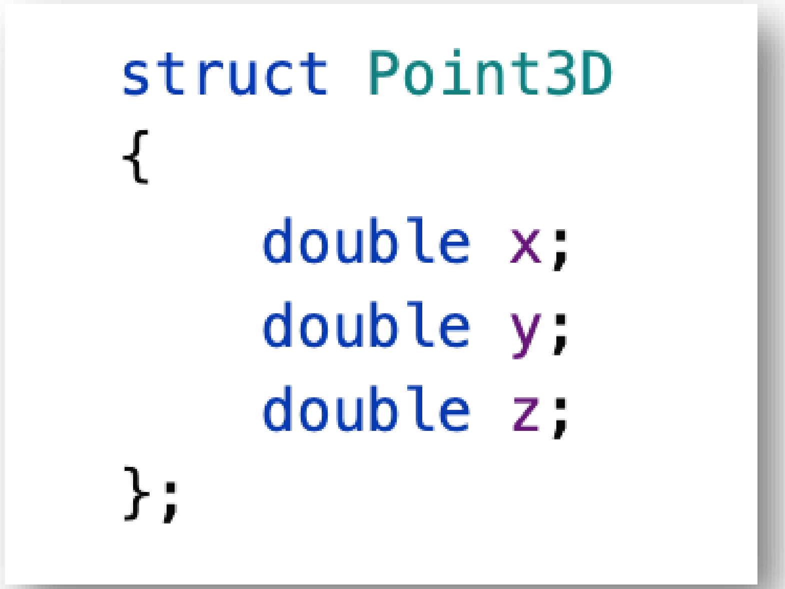
- ~~Code Style~~
- ~~References~~
- ~~Heap (Free Store)~~
- Structs

- In-class exercise

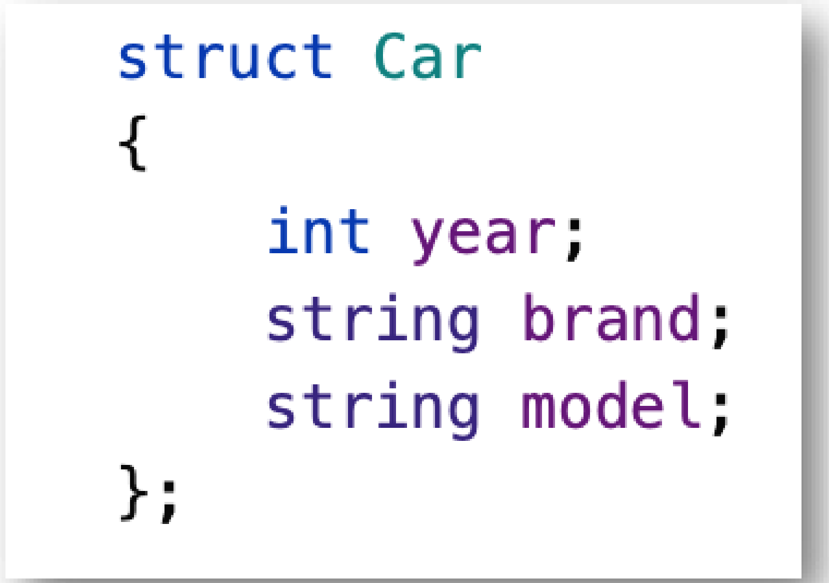


Structs

- 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;
};
```

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

// Declare a structure named "Car"
struct Car
{
    int year;
    string brand;
    string model;
};

int main()
{
    // A Car variable (named object).
    Car dreamCar;
    dreamCar.year = 1969;        // use . to access fields
    dreamCar.brand = "Ford";
    dreamCar.model = "Mustang";

    // Another Car object.
    Car myCar = {2006, "Honda", "CRV"};

    cout << "Dream car: " << dreamCar.year
                          << " " << dreamCar.brand
                          << " " << dreamCar.model << endl;

    return 0;
}
```

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

// Declare a structure named "Car"
struct Car
{
    int year;
    string brand;
    string model;
};

void printCar(Car carToPrint)
{
    cout << "Car: " << carToPrint.year
                    << " " << carToPrint.brand
                    << " " << carToPrint.model << endl;
}

int main()
{
    // Object of type Car
    Car myCar = {2006, "Honda", "CRV"};

    printCar(myCar);

    return 0;
}
```

Use structs to pass around grouped information

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

// Declare a structure named "Car"
struct Car
{
    int year;
    string brand;
    string model;
};

Car getCorollaByYear(int modelYear)
{
    Car corolla;
    corolla.year = modelYear;
    corolla.brand = "Toyota";
    corolla.model = "Corolla";

    return corolla;
}

int main()
{
    // Get a Toyota
    Car toyota = getCorollaByYear(2023);

    return 0;
}
```

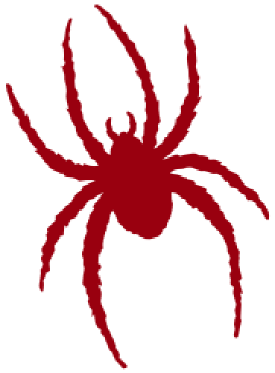
Use structs to return grouped information

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

// Declare a structure named "Car"
struct Car
{
    int year;
    string brand;
    string model;
};

int main()
{
    // Create a new struct on the heap.
    // Using a pointer to a Car struct.
    Car* dreamCar = new Car;

    // Use . to access fields after dereferencing pointer.
    // Notice the use of the * "contents of" operator.
    (*dreamCar).year = 1969;
    (*dreamCar).brand = "Ford";
    (*dreamCar).model = "Mustang";

    cout << "Dream car: " << (*dreamCar).year
                          << " " << (*dreamCar).brand
                          << " " << (*dreamCar).model << endl;

    return 0;
}
```

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

// Declare a structure named "Car"
struct Car
{
    int year;
    string brand;
    string model;
};

int main()
{
    // Create a new struct on the heap.
    // Using a pointer to a Car struct.
    Car* dreamCar = new Car;
    dreamCar->year = 1969;          // using the -> operator to access the fields
    dreamCar->brand = "Ford";
    dreamCar->model = "Mustang";

    cout << "Dream car: " << dreamCar->year
                          << " " << dreamCar->brand
                          << " " << dreamCar->model << endl;

    return 0;
}
```

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