

C++: Pointers & Dynamic Memory

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

CV Kunjeti

Lang Qin

Felix Sun

Sean Chuang

Heyi Liu

Serena Chen

Kevin Bernat

Yuna Shao



pollev.com/tqm

❖ How is HW0 going?

Administrivia

- ❖ HW0 is due on Friday
 - Can already setup your docker environment, please do that.
 - Should have everything you need after this lecture.

- ❖ Pre-semester survey out today on canvas
 - For credit, but answers are anonymous
 - Due Wednesday January 31st at 11:59 pm

- ❖ HW1 to be released on Friday or Monday
 - should have everything you need either after Wednesday's or Monday's lecture

Lecture Outline

- ❖ **HW0 demo**
- ❖ Pointers
- ❖ Dynamic memory
- ❖ `std::array`

Lecture Outline

- ❖ HW0 demo
- ❖ **Pointers**
- ❖ Dynamic memory
- ❖ `std::array`

Poll Everywhere

pollev.com/tqm

❖ What does this code print?

```
int main(int argc, char** argv) {
    int x{5}; x z ||
    int y {10}; 15
    int& z {x}; // binds the name "z" to x

x += 1;
    x += 1;
x z = y; ←
x z += 1;
    y += 5;

    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "z: " << z << endl;

    return EXIT_SUCCESS;
}
```

Pointers

❖ Variables that store addresses

- It stores the address to somewhere in memory
- Must specify a type so the data at that address can be interpreted

❖ Generic definition: `type* name;` or `type *name;`

equivalent

- Example: `int *ptr;`

- Declares a variable that can contain an address
- Trying to access that data at that address will treat the data there as an int

❖ Pointers can be thought of as references, but you can reassign what it refers to.

Pointer Operators

❖ *Dereference* a pointer using the unary ***** operator

- Access the memory referred to by a pointer
- Can be used to read or write the memory at the address

▪ Example:

```
int *ptr = ...; // Assume initialized
int a = *ptr; // read the value
*ptr = a + 2; // write the value
```

❖ Get the address of a variable with **&**

- `&foo` gets the address of `foo` in memory

▪ Example:

```
int a = 595;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```


Pointers: assigning

- ❖ There are two ways you can interact with a pointer
- ❖ Assigning/changing what variable the pointer is “referring” to

```
int a = 5930;
int b = 5950;
int *ptr = &a; // ptr refers to a
ptr = &b; // ptr now "refers" to b
```

- ❖ Assigning/changing the value of the thing it is referring to

```
int a = 5950;
int *ptr = &a; // ptr refers to a
*ptr = 3333; // *ptr and 'a' hold the value 3333
```

Pointer Example

```


int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

Initial values
are garbage



0x2001	a	--
0x2002	b	--
0x2003	c	--
0x2004	ptr	--

Assuming that integers and pointers
each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    → a = 5;
    → b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	5
0x2002	b	3
0x2003	c	--
0x2004	ptr	--

Assuming that integers and pointers
each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	5
0x2002	b	3
0x2003	c	--
0x2004	ptr	0x2001

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    → *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	7
0x2002	b	3
0x2003	c	--
0x2004	ptr	0x2001

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}

```

0x2001	a	7
0x2002	b	3
0x2003	c	10
0x2004	ptr	0x2001

Assuming that integers and pointers
each fit into a single memory location

 **Poll Everywhere**pollev.com/tqm

❖ What does this code print?

```
int main(int argc, char** argv) {
    int x {5};
    int y {10};
    int* z {&x};

    *z += 1;
    x += 1;

    z = &y;
    *z += 1;

    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "z: " << *z << endl;

    return EXIT_SUCCESS;
}
```

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```

int main(int argc, char** argv) {
    int x = 5, y = 10;
    int* z = &x;

    *z += 1;
    x += 1;

    z = &y;
    *z += 1;

    return EXIT_SUCCESS;
}
    
```



Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

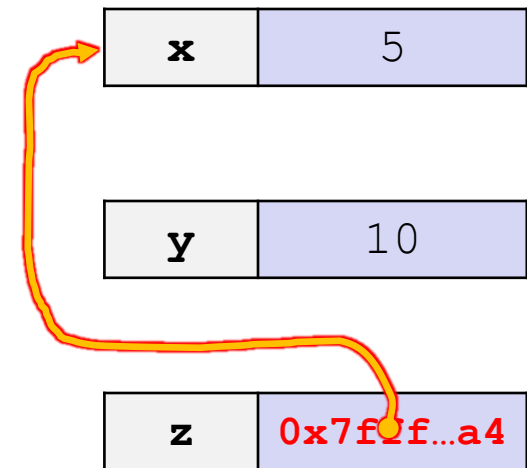
```

int main(int argc, char** argv) {
    int x = 5, y = 10;
    int* z = &x;

    *z += 1;
    x += 1;

    z = &y;
    *z += 1;

    return EXIT_SUCCESS;
}
    
```



Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

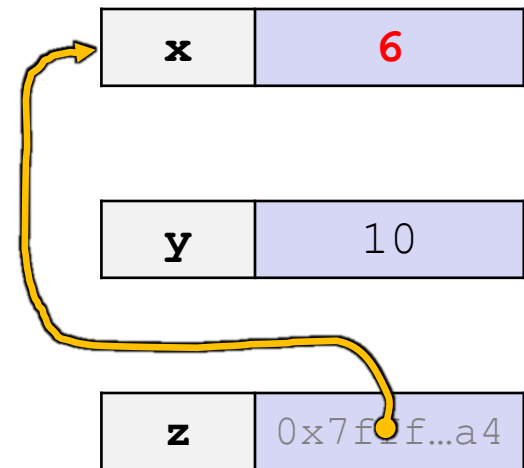
```

int main(int argc, char** argv) {
    int x = 5, y = 10;
    int* z = &x;

    *z += 1; // sets x to 6
    x += 1;

    z = &y;
    *z += 1;

    return EXIT_SUCCESS;
}
    
```



Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

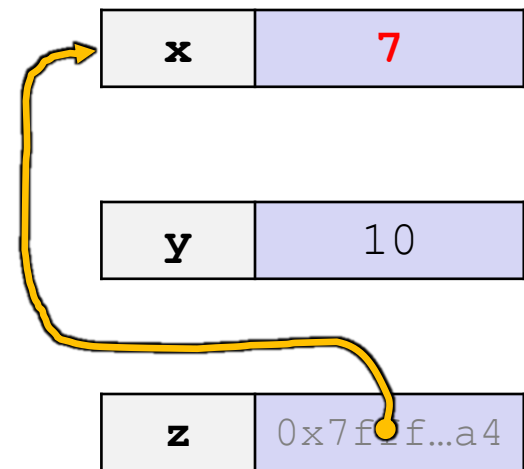
```

int main(int argc, char** argv) {
    int x = 5, y = 10;
    int* z = &x;

    *z += 1; // sets x to 6
    x += 1; // sets x (and *z) to 7

    z = &y;
    *z += 1;

    return EXIT_SUCCESS;
}
    
```



Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

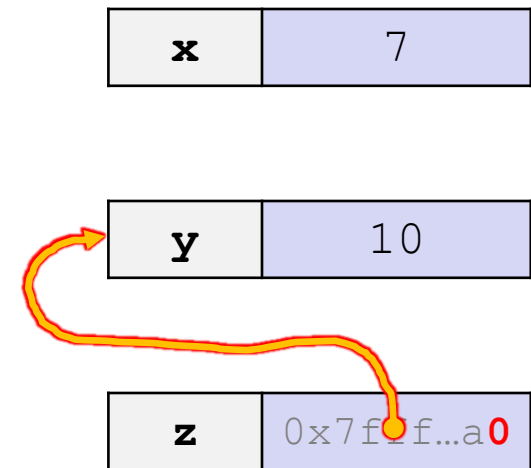
```

int main(int argc, char** argv) {
    int x = 5, y = 10;
    int* z = &x;

    *z += 1; // sets x to 6
    x += 1; // sets x (and *z) to 7

    z = &y; // sets z to the address of y
    *z += 1;

    return EXIT_SUCCESS;
}
    
```



Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

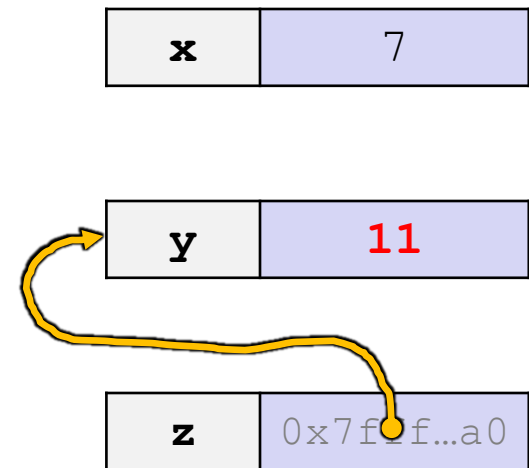
```

int main(int argc, char** argv) {
    int x {5}, y {10};
    int* z {&x};

    *z += 1; // sets x to 6
    x += 1; // sets x (and *z) to 7

    z = &y; // sets z to the address of y
    *z += 1; // sets y (and *z) to 11

    return EXIT_SUCCESS;
}
    
```



C++ `nullptr`

- ❖ C++ can have pointers that refer to nothing by assigning pointers the value `nullptr`
- ❖ `nullptr` is a useful indicator to indicate that the pointer is currently uninitialized or not in use.
- ❖ Trying to dereference or “access the value at” a pointer holding `nullptr`, will guarantee* your program to crash

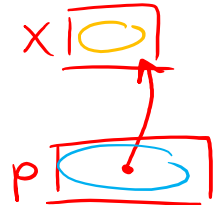
const and Pointers

❖ Pointers can change data in two different contexts:

1) You can change the value of the pointer

```
int x;
int *p = &x;
```

2) You can change the thing the pointer points to
(via dereference)



❖ `const` can be used to prevent either/both of these behaviors!

■ `const` next to pointer name means you can't change the value of the pointer

```
int *const p;
```

Handwritten red annotations: 'int' and '*' are circled, and 'const' is underlined. Arrows point from 'const' to both 'int' and '*'.

■ `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to

```
const int *p;
```

Handwritten red annotations: 'const int' is circled, and '*' is underlined. An arrow points from '*' to 'const int'.

■ Tip: read variable declaration from *right-to-left*

const and Pointers



yes



no

- ❖ The syntax with pointers is confusing:

```
int main(int argc, char** argv) {
    int x {5};           // int
    const int y {6};    // (const int)
    ✗ y++;
    const int *z {&y};  // pointer to a (const int)
    ✗ *z += 1;
    ✓ z = nullptr;
    int *const w {&x};  // (const pointer) to a (variable int)
    ✓ *w += 1;
    ✗ w = nullptr;
    const int *const v {&x}; // (const pointer) to a (const int)
    ✗ *v += 1;
    ✗ v = nullptr;
    return EXIT_SUCCESS;
}
```


Lecture Outline

- ❖ HW0 demo
- ❖ Pointers
- ❖ **Dynamic memory**
- ❖ `std::array`

Stack Example:

```

#include <iostream>
#include <cstdlib>

→ int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int sum = sum(3);
    cout << "sum: " << sum;
    cout << endl;
    return EXIT_SUCCESS;
}
    
```

```
int sum;
```

Stack frame for
main()

```
int i;
```

```
int sum;
```

```
int n;
```

Stack frame for
sum()

Stack Example 1:

```

#include <iostream>
#include <cstdlib>

int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int sum = sum(3);
    cout << "sum: " << sum;
    cout << endl;
    return EXIT_SUCCESS;
}
    
```

```
int sum;
```

Stack frame for
`main()`

`sum()`'s stack frame
goes away after
`sum()` returns.

`main()`'s stack frame
is now top of the stack
and we keep executing
`main()`

Stack Example:

```

#include <iostream>
#include <cstdlib>

int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int sum = sum(3);
    cout << "sum: " << sum;
    cout << endl;
    return EXIT_SUCCESS;
}
    
```

int sum;

Stack frame for
main()

????

Stack frame for
cout << string

Stack

- ❖ Grows, but has a static max size
 - Can find the default size limit with the command `ulimit -all`
(May be a different command in different shells and/or linux versions. Works in bash on Ubuntu though)
 - Can also be found at runtime with `getrlimit(3)`

- ❖ Max Size of a stack can be changed
 - at run time with `setrlimit(3)`
 - At compilation time for some systems (not on Linux it seems)
 - (or at the creation of a thread)

 **Poll Everywhere**pollev.com/tqm

- ❖ Does this code compile? If so, what does it print? If not, what are the compiler errors? (compiler warnings can be ignored for now)

```
string& get_string() {
    string greeting{"hello world!"};
    return greeting;
}

int main(int argc, char** argv) {
    string& s = get_string();
    cout << s << endl;

    return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- ❖ Does this code compile? If so, what does it print? If not, what are the compiler errors? (compiler warnings can be ignored for now)

main()'s stack frame

```
string& get_string() {
    string greeting{"hello world!"};
    return greeting;
}

int main(int argc, char** argv) {
    string& s = get_string();
    cout << s << endl;

    return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- ❖ Does this code compile? If so, what does it print? If not, what are the compiler errors? (compiler warnings can be ignored for now)

main()'s stack frame

get_string()'s stack frame

greeting hello world!

```
string& get_string() {
    string greeting{"hello world!"};
    return greeting;
}

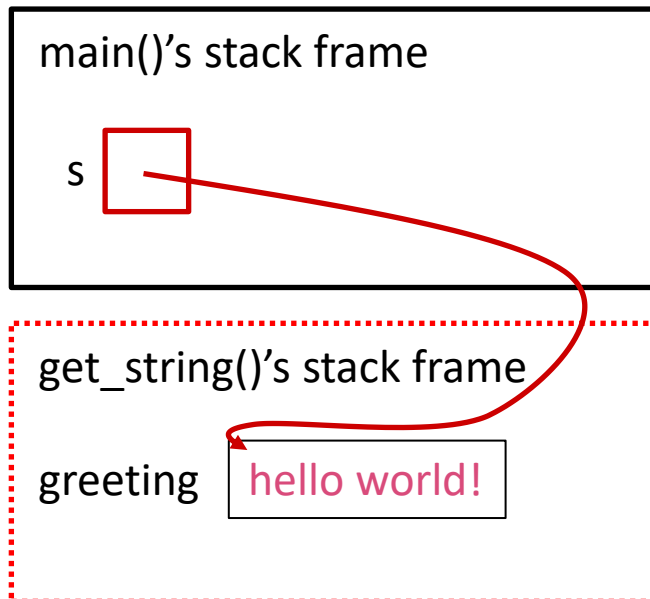
int main(int argc, char** argv) {
    string& s = get_string();
    cout << s << endl;

    return EXIT_SUCCESS;
}
```


Poll Everywhere

pollev.com/tqm

- ❖ Does this code compile? If so, what does it print? If not, what are the compiler errors? (compiler warnings can be ignored for now)



```
string& get_string() {
    string greeting{"hello world!"};
    return greeting;
}

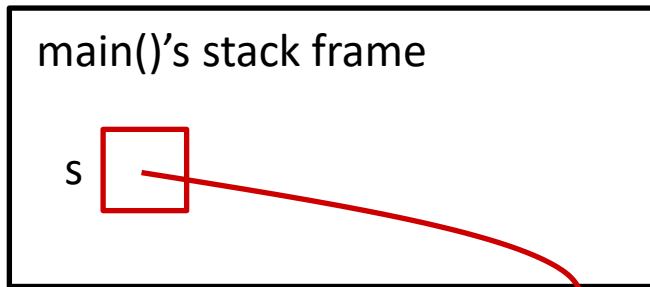
int main(int argc, char** argv) {
    string& s = get_string();
    cout << s << endl;

    return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- ❖ Does this code compile? If so, what does it print? If not, what are the compiler errors? (compiler warnings can be ignored for now)



????????

```
string& get_string() {
    string greeting{"hello world!"};
    return greeting;
}

int main(int argc, char** argv) {
    string& s = get_string();
    cout << s << endl;

    return EXIT_SUCCESS;
}
```

Memory Allocation

❖ So far, we have seen two kinds of memory allocation:

```
int counter = 0;    // global var

int main() {
    counter++;
    cout << "count = " << counter;
    cout << endl;
    return 0;
}
```

- counter is **statically**-allocated
 - Allocated when program is loaded
 - Deallocated when program exits

```
int foo(int a) {
    int x = a + 1;    // local var
    return x;
}

int main() {
    int y = foo(10); // local var
    cout << "y = " << y << endl;
    return 0;
}
```

- a, x, y are **automatically**-allocated
 - Allocated when function is called
 - Deallocated when function returns



What is Dynamic Memory Allocation?

- ❖ We want Dynamic Memory Allocation
 - **Dynamic means “at run-time”**
 - The compiler and the programmer don't have enough information to make a final decision on how much to allocate or how long the data “should live”.
- ❖ **Dynamic memory can be of variable size:**
 - Your program explicitly requests more memory at run time
 - The language allocates it at runtime, probably with help of the OS
- ❖ **Dynamically allocated memory persists until either:**
 - A garbage collector collects it (automatic memory management)
 - Your code explicitly deallocates it (manual memory management)

The Heap

- ❖ The Heap is a large pool of available memory to use for Dynamic allocation
- ❖ This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.

C++ keyword: new

- ❖ C++ keyword `new` is used to allocate space on the heap.
 - We specify a type and initial value which will be constructed and/or initialized for us.

```
string *get_string() {  
    string *greeting = new string("hello world!");  
    return greeting;  
}  
  
int main(int argc, char** argv) {  
    string *s = get_string();  
    cout << *s << endl;  
  
    return EXIT_SUCCESS;  
}
```

Dynamic Memory Deallocation

- ❖ Dynamic memory has a dynamic “lifetime”
 - Stack data is deallocated when the function returns
 - Heap data is deallocated when our program deallocates it
- ❖ In high level languages like Java or Python, garbage collection is used to deallocate data
 - This has significant overhead for larger programs
- ❖ C requires you to manually manage memory
 - And so is easy to screw up
- ❖ C++ and Rust have RAI (more on this later this lecture)
 - Harder to screw-up, and much less overhead

Dynamic Memory Deallocation

- ❖ When is the string we allocate deallocated?

```
string *get_string() {  
    string *greeting = new string("hello world!");  
    return greeting;  
}  
  
int main(int argc, char** argv) {  
    string *s = get_string();  
    cout << *s << endl;  
  
    return EXIT_SUCCESS;  
}
```


C++ keyword: delete

- ❖ C++ keyword delete is used to deallocate space on the heap.

```
string *get_string() {  
    string *greeting = new string("hello world!");  
    return greeting;  
}  
  
int main(int argc, char** argv) {  
    string *s = get_string();  
    cout << *s << endl;  
    delete s;  
    return EXIT_SUCCESS;  
}
```

The Heap

KEY TAKEAWAY: allocating on the heap is not free, it has overhead

- ❖ The Heap is a large pool of available memory to use for Dynamic allocation
- ❖ This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- ❖ **new:**
 - searches for a large enough unused block of memory
 - marks the memory as allocated.
 - Returns a pointer to the beginning of that memory
- ❖ **delete:**
 - Takes in a pointer to a previously allocated address
 - Marks the memory as free to use.

Free Lists

- ❖ One way that allocation can be implemented is by maintaining an implicit list of the space available and space allocated.
- ❖ Before each chunk of allocated/free memory, we'll also have this metadata:

```
// this is simplified  
// not what malloc/new really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```

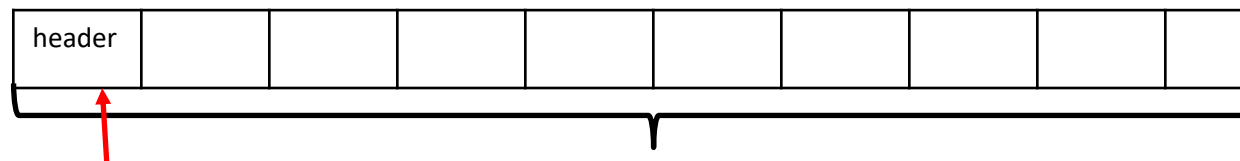
Dynamic Memory Example

```

int main() {
    short* ptr = new short(16);
    double* ptr2 = new double(3.14);
    ...           // do stuff with ptr
    delete ptr;
    delete ptr2;
}
    
```

This diagram is not to scale

❖ free_list ->



```

{
    NULL,
    NULL,
    false,
    1024
}
    
```

The metadata is at the beginning of the chunk of memory

Dynamic Memory Example

```

int main() {
short* ptr = new short(16);
double* ptr2 = new double(3.14);
...           // do stuff with ptr
delete ptr;
delete ptr2;
}

```

Free chunks can be split to allocate blocks of specific size

new gets a pointer to just after the metadata

❖ free_list

"new"
return
value

```

{
  NULL,
  0x...,
  true,
  4
}
{
  0x...,
  NULL,
  false,
  1020
}

```

free_list
points to first
free chunk

Dynamic Memory Example

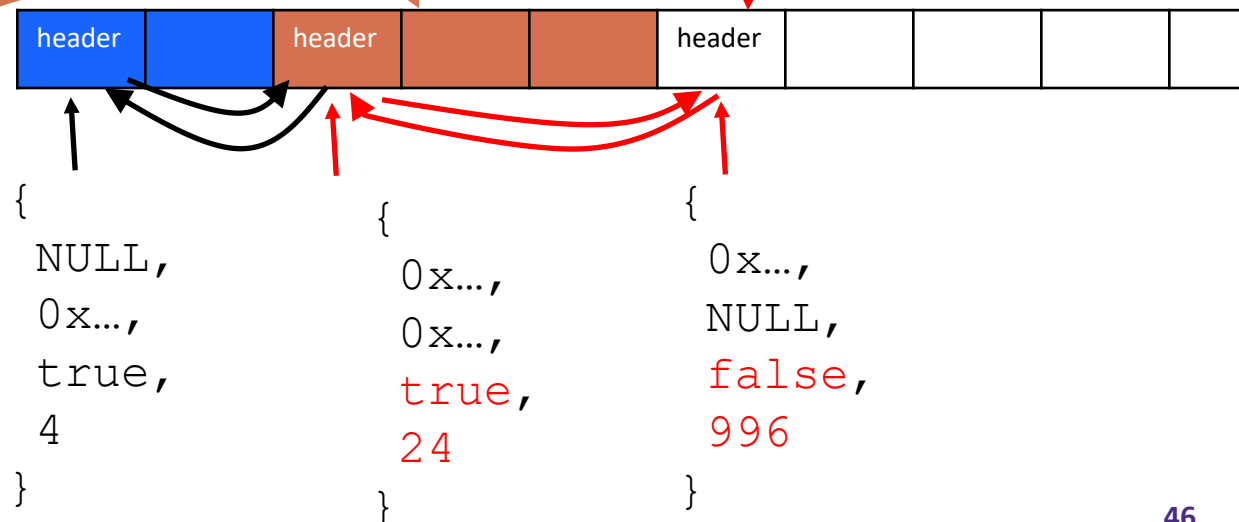
```

int main() {
    short* ptr = new short(16);
    double* ptr2 = new double(3.14);
    ...           // do stuff with ptr
    delete ptr;
    delete ptr2;
}

```

❖ free_list

"new"
return
value

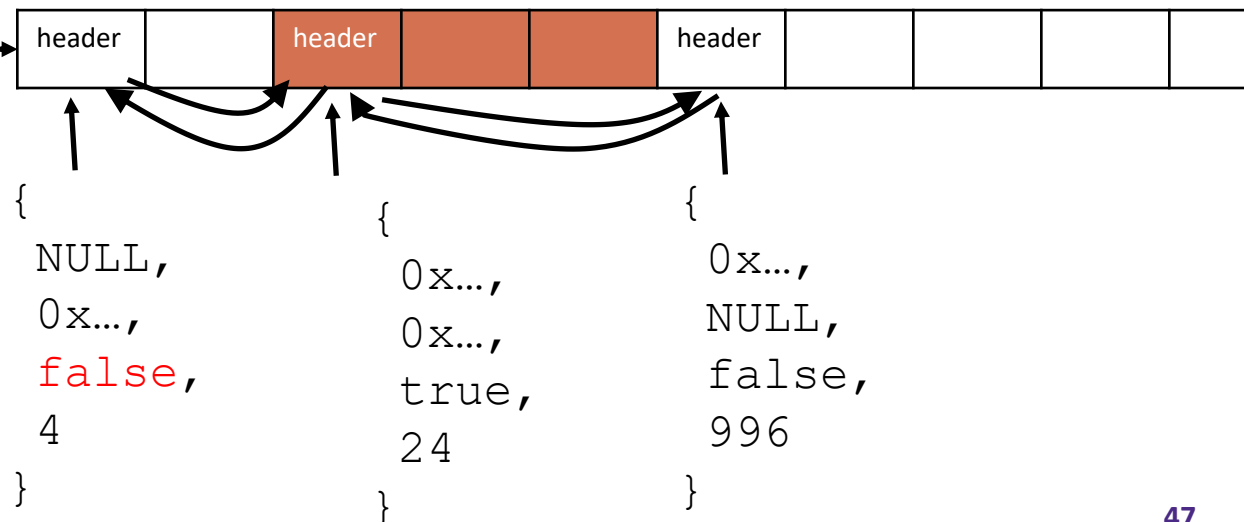


Dynamic Memory Example

```

int main() {
    short* ptr = new short(16);
    double* ptr2 = new double(3.14);
    ...           // do stuff with ptr
    delete ptr;
    delete ptr2;
}
    
```

❖ free_list →

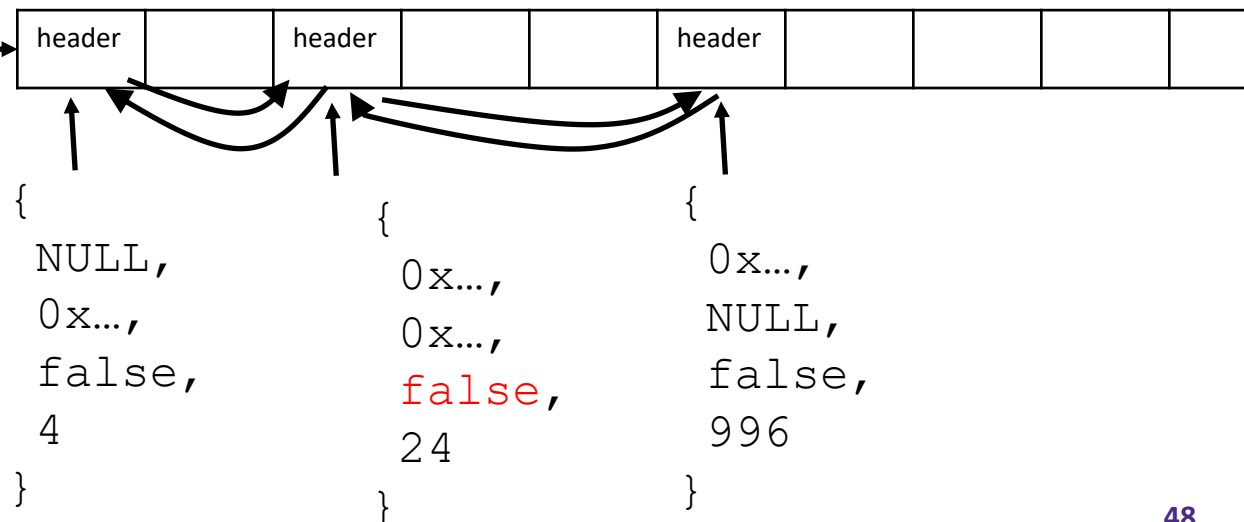


Dynamic Memory Example

```

int main() {
    short* ptr = new short(16);
    double* ptr2 = new double(3.14);
    ...           // do stuff with ptr
    delete ptr;
    delete ptr2;
}
    
```

❖ free_list



Dynamic Memory Example

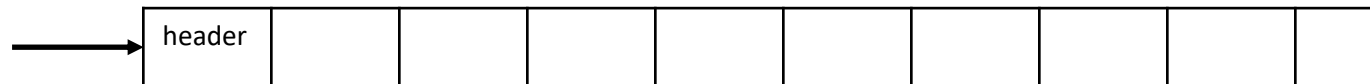
```

int main() {
    short* ptr = new short(16);
    double* ptr2 = new double(3.14);
    ...           // do stuff with ptr
    delete ptr;
    delete ptr2;
}
    
```

Once a block has been freed, we can try to "coalesce" it with their neighbors

The first delete couldn't be coalesced, only neighbor was allocated

❖ free_list



```

{
    NULL,
    0x...,
    false,
    1024
}
    
```

Key Takeaway

- ❖ **Dynamic memory allocation is not free and can have considerable overhead**
- ❖ Performant C++ code minimizes the number of dynamic allocations and/or custom allocators

Lecture Outline

- ❖ HW0 demo
- ❖ Pointers
- ❖ Dynamic memory
- ❖ **std::vec allocations and std::array**

Why would I use new?

- ❖ In “real” or “modern” C++ code, you would not explicitly use new or delete yourself.
- ❖ In most cases, a vector or other data structure can be used, and you never have to allocate memory yourself
- ❖ whenever you are using objects from the C++ standard library (like vector), those objects will do memory allocation.

vector Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char** argv) {

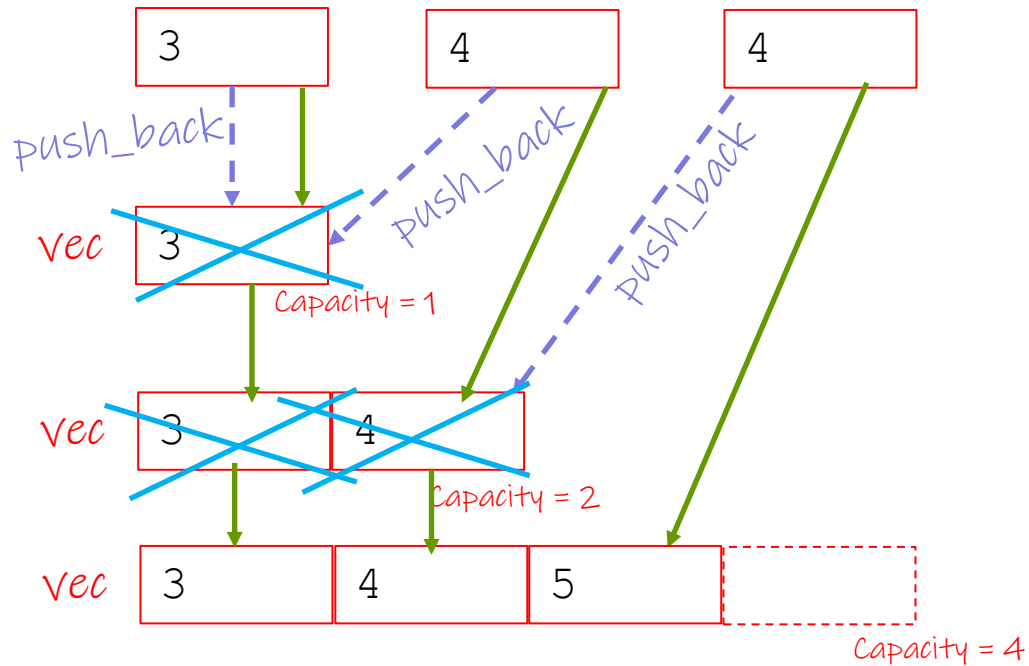
    vector<int> vec; ← Construct empty vector

    cout << "vec.push_back " << 3 << endl;
    vec.push_back(3); ← Add elements to
    cout << "vec.push_back " << 4 << endl; ← end of vector
    vec.push_back(4);
    cout << "vec.push_back " << 5 << endl;
    vec.push_back(5);

    cout << "vec[0]" << endl << vec.at(0) << endl;
    cout << "vec[2]" << endl << vec.at(2) << endl;

    return EXIT_SUCCESS;
}
```

Where is the allocation?



Note:

- Capacity doubles each time capacity is reached

Where is the deletion?

- ❖ This code has allocation, where is the deallocation?

```
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char** argv) {

    vector<int> vec;

    cout << "vec.push_back " << 3 << endl;
    vec.push_back(3);
    cout << "vec.push_back " << 4 << endl;
    vec.push_back(4);
    cout << "vec.push_back " << 5 << endl;
    vec.push_back(5);

    cout << "vec[0]" << endl << vec.at(0) << endl;
    cout << "vec[2]" << endl << vec.at(2) << endl;

    return EXIT_SUCCESS;
}
```

Destructors

- ❖ C++ has the notion of a **destructor (dtor)**
 - Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
 - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
 - Standard C++ idiom for managing dynamic resources
 - Slogan: “*Resource Acquisition Is Initialization*” (RAII)

tilde

No parameters

More on RAII in a later lecture

```

MyObj::~~MyObj() { // destructor
    // do any cleanup needed when a "MyObj" object goes away
    // (nothing to do here since we have no dynamic resources)
}
    
```

When a destructor is invoked:

1. run destructor body
2. Call destructor of any data members

Destructor Example

```

class Integer {
public:
    Integer(int val) : val_(new int(val)) {           // Constructor
    }

    ~Integer() { delete val_; }                       // Destructor
    int get_value() { return *val_; }                // inline member function
private:
    int* val_; // data member
}; // class Integer
    
```

Allocates memory in the constructor
 Without destructor, the memory wouldn't be freed

Integer.h

```

#include "Integer.h"
#include <iostream>

int main(int argc, char** argv) {
    Integer best_course{5950};
    cout << best_course.get_value() << endl;
    return EXIT_SUCCESS;
}
    
```

Destruct the object when it falls out of scope (when we return)

Default Destructor

- ❖ 9 out of ten times, most objects do not need to create an explicit destructor.
- ❖ Destructors can be specified to be a default the C++ generates for you
- ❖ The default destructor just runs the destructor of any data member (fields) the object has.
 - So, if your custom object has a vector or a map, then those data structures will automatically get destructed/"cleaned-up"

Default Destructor Example

```

#ifndef POINT_HPP_
#define POINT_HPP_

class Point {
public:
    Point(int x, int y);           // constructor
    ~Point() = default;           // Default destructor since we
                                  // don't do any allocation in Point
    int get_x() { return x_; }    // inline member function
    int get_y() { return y_; }    // inline member function
    double Distance(Point p);     // member function
    void SetLocation(int x, int y); // member function

private:
    int x_; // data member
    int y_; // data member
}; // class Point

#endif // POINT_HPP_
    
```

std::array

- ❖ Similar to vector, we have array
 - Both contain a sequence of data that we can index into
- ❖ Main differences: the size
 - Vector is resizable (grows to whatever length we need)
 - Array is a static size (size is determined at compile time)
- ❖ Main differences: the allocation
 - To support being resizable, vector uses a lot of dynamic allocation
 - **Array does not use any dynamic allocation**

array example

```
int main(int argc, char* argv[]) {
    array<int, 3> arr {6, 5, 4};
    // arr.push_back(3); push_back does not exist!

    cout << arr.size() << endl; // prints 3
    cout << arr.at(2) << endl;  // prints 4

    // iterates through all elements and prints them
    for (const auto& element : arr) {
        cout << element << endl;
    }

    return EXIT_SUCCESS;
}
```