

Module 11: Dynamic Memory Allocation

Learning Objectives

1. Trace code that uses pointers.
2. Explain memory management issues with raw pointers.
3. Write code that creates and dereferences shared pointers.

Process Skills

1. Information processing. Extract structural patterns from sample code.
2. Oral and written communication. Describe a technical concept expressed in a written form.

Please fill in the roles for each member of your team. Take a look at the description of each role to see its responsibilities. If there are only three people in the group, please assign the same person to the **Presenter** and **Reflector** role. It is a good idea to select roles that you have not recently taken.

Team name: _____

Date: _____

Role	Team Member Name
Manager. Keeps track of time and makes sure everyone contributes appropriately.	
Presenter. Talks to the facilitator and other teams.	
Reflector. Considers how the team could work and learn more effectively.	
Recorder. Records all answers and questions and makes the necessary submission.	

For virtual activities: Once you select your roles, [change your Zoom name](#) using the format and example below.

Format: Group X: First name, Last name initial / Role

Example: Group 1: Paul I / Presenter



Model 1. Pointers and addresses (18 min)

Start time: _____

```

int main() {
    std::string name = "Tuffy Titan";
    std::cout << "Value: " << name << "\n";
    std::cout << "Address: " << &name << "\n";

    std::string* name_ptr = &name;
    std::cout << "Pointer Value: " << name_ptr << "\n";
    std::cout << "Dereferenced value: " << *name_ptr << "\n";

    *name_ptr = "Tuffy";
    std::cout << "Value from variable: " << name << "\n";
    std::cout << "Value from pointer: " << *name_ptr << "\n";

    std::cout << "Dereferenced member function call 1: "
                << (*name_ptr).size() << "\n";
    std::cout << "Dereferenced member function call 2: "
                << name_ptr->size() << "\n";

    return 0;
}

```

Screen output*

```

Value: Tuffy Titan
Address: 0x7ffe9b0cd10
Pointer Value: 0x7ffe9b0cd10
Dereferenced value: Tuffy Titan
Value from variable: Tuffy
Value from pointer: Tuffy
Dereferenced member function call 1: 5
Dereferenced member function call 2: 5

```

* Memory addresses are usually different every time you run the program because it depends on the system's memory availability.


Symbol Table

Variable Name	Scope	Type	Memory address*
name	main()	std::string	0x7ffe9b0cd10
name_ptr	main()	std::string*	0x7ffe9b0cd08

* memory addresses are often represented in [hexadecimal format](https://en.wikipedia.org/wiki/Hexadecimal) (<https://en.wikipedia.org/wiki/Hexadecimal>)



Memory Visualization (Shows memory addresses in the Stack)

Memory Address	Value
0x7ffe9b0cd08	
0x7ffe9b0cd09	
0x7ffe9b0cd0a	
0x7ffe9b0cd0b	
0x7ffe9b0cd0c	
0x7ffe9b0cd0d	
0x7ffe9b0cd0e	
0x7ffe9b0cd0f	
0x7ffe9b0cd10	
0x7ffe9b0cd11	
0x7ffe9b0cd12	
0x7ffe9b0cd13	
0x7ffe9b0cd14	
0x7ffe9b0cd15	
0x7ffe9b0cd16	
0x7ffe9b0cd17	
0x7ffe9b0cd18	
0x7ffe9b0cd19	
0x7ffe9b0cd1a	
0x7ffe9b0cd1b	
0x7ffe9b0cd1c	
0x7ffe9b0cd1d	
0x7ffe9b0cd1e	<div> name: std::string </div> <div> "Tuffy Titan" </div>
0x7ffe9b0cd1f	
0x7ffe9b0cd20	
0x7ffe9b0cd21	
0x7ffe9b0cd22	
0x7ffe9b0cd23	
0x7ffe9b0cd24	
0x7ffe9b0cd25	
0x7ffe9b0cd26	
0x7ffe9b0cd27	
0x7ffe9b0cd28	
0x7ffe9b0cd29	
0x7ffe9b0cd2a	
0x7ffe9b0cd2b	
0x7ffe9b0cd2c	
0x7ffe9b0cd2d	
0x7ffe9b0cd2e	
0x7ffe9b0cd2f	

1. According to the symbol table, where can we find the first memory address occupied by name variable in the memory?

0x7ffe9b0cd10

2. Each memory address represents one byte. According to the memory visualization, how much space in bytes does the name std::string use?

32 bytes

3. What is the data type of the name_ptr variable?

std::string*

4. According to the symbol table, where can we find the first memory address occupied by the name_ptr variable in the memory?

0x7ffe9b0cd08

5. According to the memory visualization, how much space in bytes does the name_ptr std::string pointer use?

8 bytes

6. The & symbol can also be used to get the memory address of a variable. In this case, we use the term *address operator*. What value did we get when the compiler ran the expression &name? Place a check (✓) beside your answer.

- a. 0x7ffe9b0cd10 ✓
- b. 0x7ffe9b0cd08
- c. "Tuffy Titan"

7. What kind of value does a std::string pointer contain? Place a check (✓) beside your answer.

- a. std::string
- b. memory address of an int
- c. memory address of a std::string ✓



8. Why does the program display 0x7ffe9b0cd10 when we ask it to output name_ptr? Place a check (✓) beside your answer.
- We assigned the address of name to name_ptr. ✓
 - This is a random initial value assigned o the name_ptr.
 - The system actually displays "Tuffy Titan" not an address.
9. The pointer operator (*) is used to *dereference* a pointer variable. *Dereference* means to access the object in the memory address stored by the variable. In other words, access the value pointed to by the pointer. What value do we expect to get when the compiler runs the expression *name_ptr? Place a check (✓) beside your answer.
- 0x7ffe9b0cd10
 - 0x7ffe9b0cd08
 - "Tuffy Titan" ✓
10. When we dereference an object to call its member function, we need to place the pointer operator and the variable name inside a parentheses. Without the parentheses, the compiler will try to apply the dot notation on the pointer before dereferencing (which we do not want). We can use the arrow operator (->) to shorten the code. It is a shorthand of dereferencing then accessing a member function. Complete the table below to call the size member function of the std::string pointed to by city_ptr using the pointer and arrow operators.

pointer variable used	Acces using pointer operator	Access using arrow operator
std::string* name_ptr;	(*name_ptr).size()	name_ptr->size()
std::string* city_ptr;	(*city_ptr).size()	city_ptr->size()

 **STOP HERE AND WAIT FOR FURTHER INSTRUCTIONS**

Model 2. Dynamic memory allocation (12 min)

Start time: _____

```
int main() {
    std::string* city_ptr = new std::string("Fullerton");
    std::cout << "Pointer address: " << &city_ptr << "\n";
    std::cout << "Pointer value: " << city_ptr << "\n";
    std::cout << "Dereferenced value: " << *city_ptr << "\n";
    delete city_ptr;
    city_ptr = nullptr;
    return 0;
}
```

Screen output*

Pointer address: 0x7ffe40b1ac30
 Pointer value: 0xb832c0
 Dereferenced value: Fullerton

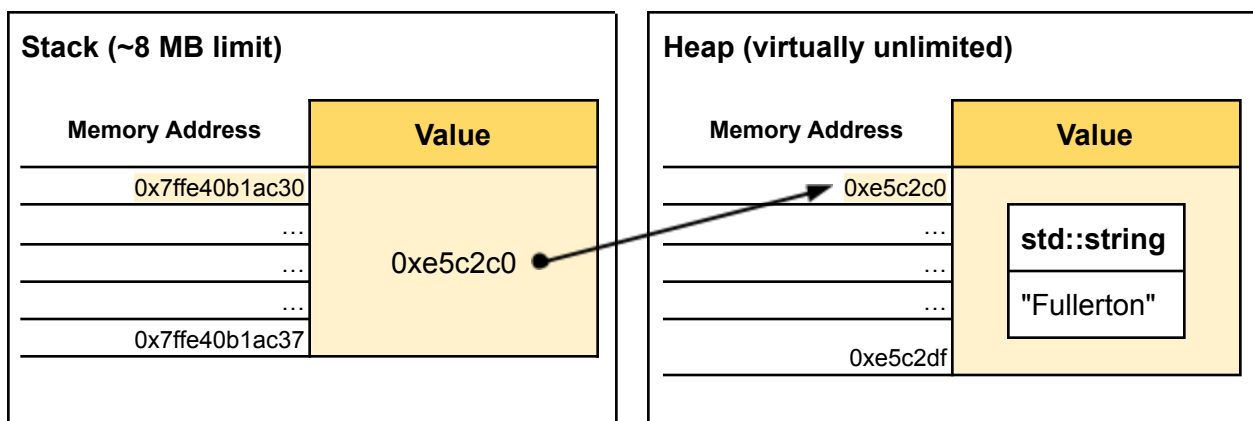
* Memory addresses are usually different every time you run the program because it depends on the system's memory availability.

Symbol Table

Variable Name	Scope	Type	Memory address*
city_ptr	main()	std::string*	0x7ffe40b1ac30

* memory addresses are often represented in [hexadecimal format](https://en.wikipedia.org/wiki/Hexadecimal)
<https://en.wikipedia.org/wiki/Hexadecimal>

Memory Visualization



11. All variables and functions your program uses are stored in a designated part of the memory called a *stack*. The stack is limited to a certain amount. In our class' programming environment, this is only about 8 MB. If your program exceeds the space allocated, it will cause a *stack overflow*. To avoid this issue, we use another part of the memory called the *heap*, virtually unlimited. According to the code, where is the `city_ptr` variable stored? Place a check (✓) beside your answer.

- a. Stack ✓
- b. Heap

12. The `new` keyword is used to find and reserve space in the heap for your program to use. The size reserved is based on the size of the data type provided after the `new` keyword. In our example, how much space do you think is reserved in the heap when the compiler runs the expression `new std::string("Fullerton");` ?

32 bytes

13. The syntax for reserving space in the heap is shown below. Write code to reserve space that will hold a `Food` object using its default constructor. *Note: We use the same rules in providing parentheses (or not) when calling constructors during object instantiation.*

```
<data type>* <variable name> = new <data type><constructor call>;  
std::string* city_ptr          = new std::string("Fullerton");
```

`Food* apple = new Food;`

14. We use the same dereferencing rules regardless if we point to an object in the stack or heap. Complete the code to display the size of the `std::string` pointed to by `city_ptr`.

```
std::cout << "City name size: " << (*city_ptr).size();  
std::cout << "City name size: " << city_ptr->size();
```

15. The *delete* keyword tells the compiler that the address pointed to by the pointer will no longer be used. What do you think will happen if we access a deleted memory location? This is called *dangling reference*. Place a check (✓) beside your answer.

```
delete city_ptr;  
std::cout << "City name after deletion: " << *city_ptr << "\n";
```

- a. The program may crash if another program used the deleted memory address. ✓
 - b. The system guarantees there will be no errors.
 - c. The memory address remains reserved for our program so no other program can use it (wasted resource).
16. If we forget to call delete on a memory address, it will remain reserved in the heap. What is a potential issue when a memory address is not deleted? See the code below for an example. This is called a *memory leak*. Place a check (✓) beside your answer.

```
for (int i = 0; i < 10; i++) {  
    std::string* temp = new std::string("garbage");  
}
```

- a. The program may crash if another program used the memory address.
- b. The system guarantees there will be no errors.
- c. The memory address remains reserved for our program so no other program can use it (wasted resource). ✓

 **STOP HERE AND WAIT FOR FURTHER INSTRUCTIONS**

Model 3. Shared Pointers (18 min)

Start time: _____

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<std::string> safe_space =
        std::make_shared<std::string>("safe space");

    std::cout << "Shared pointer location: " << &safe_space << "\n";
    std::cout << "Object location: " << safe_space << "\n";
    std::cout << "Object value: " << *safe_space << "\n";
    std::cout << "Object member function call: "
        << safe_space->size() << "\n";

    std::cout << "Shared pointers pointing to object: "
        << safe_space.use_count() << "\n";

    std::shared_ptr<std::string> safe_space_copy = safe_space;
    std::cout << "Shared pointers pointing to object: "
        << safe_space.use_count() << "\n";
    safe_space_copy = nullptr;

    std::cout << "Shared pointers pointing to object: "
        << safe_space.use_count() << "\n";
    safe_space = nullptr;
    return 0;
}
```

Screen output*

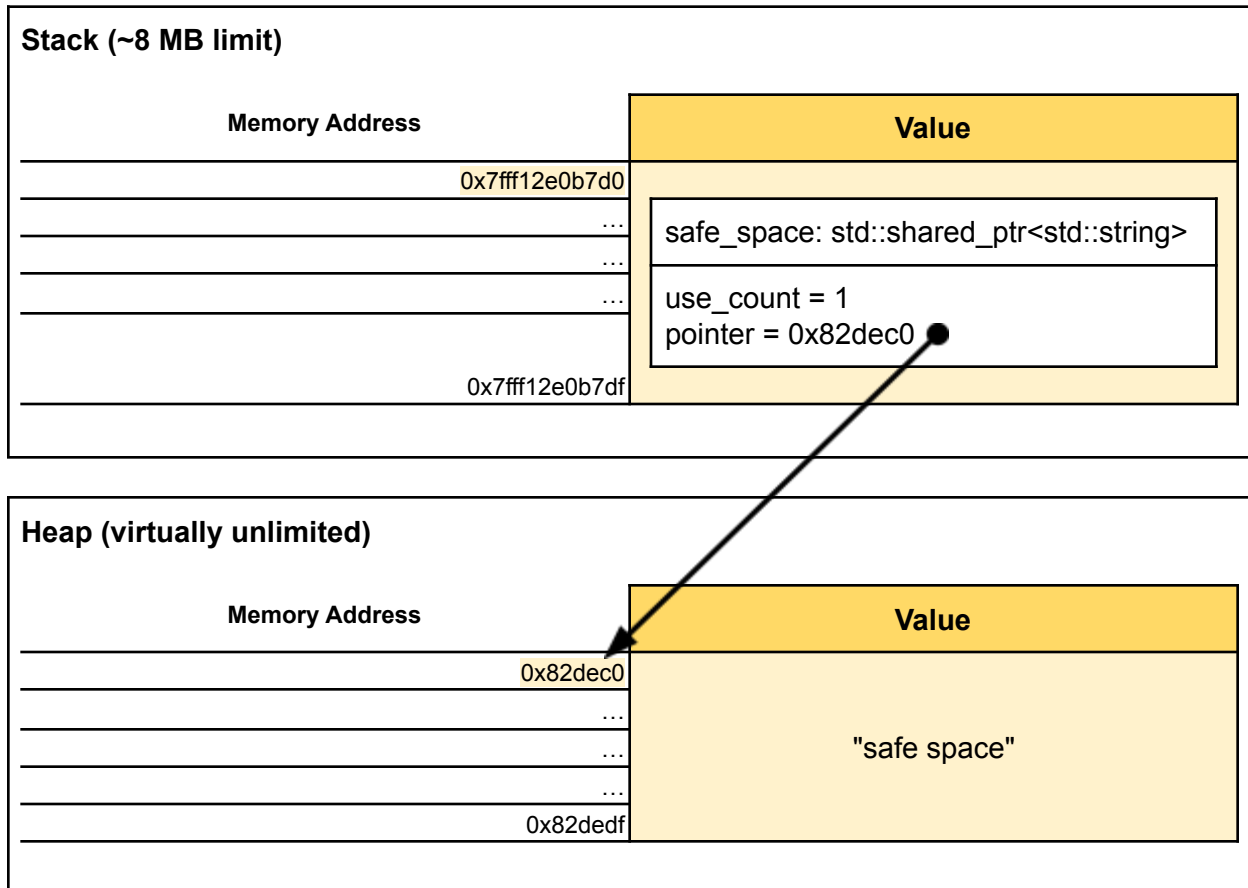
```
Shared pointer location: 0x7fff12e0b7d0
Object location: 0x82dec0
Object value: safe space
Object member function call: 10
Shared pointers pointing to object: 1
Shared pointers pointing to object: 2
Shared pointers pointing to object: 1
```

* Memory addresses are usually different every time you run the program because it depends on the system's memory availability.

Symbol Table

Variable Name	Scope	Type	Memory address*
safe_space	main()	std::shared_ptr<std::string>	0x7fff12e0b7d0

* memory addresses are often represented in [hexadecimal format](https://en.wikipedia.org/wiki/Hexadecimal_format)
<https://en.wikipedia.org/wiki/Hexadecimal>

Memory Visualization

17. `std::shared_ptr` is a class that helps address potential raw pointer issues. We call the `std::make_shared` function to reserve space in the heap and return a `std::shared_ptr` object. The pattern below shows how to call the function. What was the name of the `std::shared_ptr` variable we instantiated using the `std::make_shared` function in the Model 3 code?

```
std::shared_ptr<<data type>> <variable name>
    = std::make_shared<<data type>><constructor call>;
```

safe_space

18. Where is the `std::shared_ptr` object stored? Place a check (✓) beside your answer.

- a. Stack: 0x7fff12e0b7d0 ✓
- b. Heap: 0x82dec0
- c. Stack: 0x7fff12e0b7df

19. Where is the `std::string` object pointed to by `safe_space` stored? Place a check (✓) beside your answer.

- a. Stack: 0x7fff12e0b7d0
- b. Heap: 0x82dec0 ✓
- c. Stack: 0x7fff12e0b7df

20. We use the pointer and arrow operators on `std::shared_ptr` variables much like raw pointers for dereferencing. Write code to clear the `std::string` pointed to by `safe_space`. Refer to the `clear` member function below.

```
void std::string::clear();
```

```
(*safe_space).clear();  
safe_space->clear();
```

21. `std::shared_ptr` provides a `use_count` member function that tells us how many `std::shared_ptr` point to the same object. Why does assigning `safe_space` to `safe_space_copy` increase the `use_count`? Place a check (✓) beside your answer.

- a. There is only one `std::shared_ptr` pointing to the `std::string` in the heap.
- b. `safe_space` and `safe_space_copy` point to the same `std::string` in the heap. ✓
- c. There are no `std::shared_ptr` pointing to a `std::string` in the heap.

22. A `std::shared_ptr`'s `use_count` decreases by one when it is assigned a `nullptr` or when it goes out of scope. A variable goes out of scope when a function containing it exits or a code block containing the variable exits (e.g., loop body, if statement body). When there are no `std::shared_ptr` pointing to the object in the heap, it is automatically deleted. Consider the code below. How does the `std::shared_ptr` solve the memory leak problem we saw in question 16 that used raw pointers?

```
for (int i = 0; i < 10; i++) {  
    std::shared_ptr<std::string> temp  
        = std::make_shared<std::string>("garbage");  
}
```

When the iteration starts, a new `shared_ptr` is created in memory with a use count of 1. When the loop exits, the use count is reduced to 0, so it is automatically deleted from memory. When the code runs, there are no memory leaks, unlike the code in question 16.

Reflector questions

1. What was the most useful thing your team learned during this session?

2. Pointers are one of the most confusing topics in the course. Did you discover any strategies that helped you understand the topic?

3. Reflect on your explanation about code (e.g., question 22). Do you think you and your group have improved in explaining code compared to before the midterms? List ways you think can help you further improve your explanation skill.



