CPT-182 - Programming in C++

Module 8

Pointers

Dayu Wang

- **Reference Variables**

→ **'&' is the address-of operator, which returns the memory address where the variable is stored.**

```cpp
ofstream fout("output.txt");
int x = 5;
int y = x;
fout << &x << endl;
fout << &y << endl;
```

output.txt
0022FAAC 0022FAA0

Although the value of variables x and y are both 5, they are stored in different memory locations.

```cpp
ofstream fout("output.txt");
int x = 5;
int &y = x;   // Reference variable
fout << &x << endl;
fout << &y << endl;
```

output.txt
003FFAB4 003FFAB4

Memory locations of variables x and y are the same.

- **They point to the same integer object in the memory.**

- **Storing the Memory Address of a Variable in Another Variable**

→ **Can we store the memory address of the current variable in another variable?**

```
/*
    Let x = 25
    Let y = &x
    Let z = x
*/
```

				x		y		z
Memory	**'H'**	**−3**	**...**	**25**	**...**	**1016**	**...**	**25**
Address	**1000**	**1004**		**1016**		**1024**		**1032**

→ **The variable (y) that stores the address of another variable is called a pointer.**

- **Pointer is a very powerful feature that has many uses in lower level programming (close to hardware level).**
- **A pointer is said to "point to" the variable whose address it stores.**

→ **Pointers can be used to access the variable they point to directly.**

- **This is done by preceding the pointer name with the dereference operator '*'.**
- **In this example,** `x == *y`**.**

- **'*' and '&' have opposite meanings:** `x == *(&x)`**.**

- **Pointer Declaration**

```
string* p;  // A pointer that points to a string variable.
```

→ **Types** `string*` **and** `string` **are different data types.**

```
/* The two declarations below are equivalent.
    string*  p;
    string  *p;
*/
```

```
// What I am declaring?
int  *x, y;
```

```
// What I am declaring?
int*  x, y;
```

Variable `x` **is a pointer to an integer;** `y` **is an integer.**

Common Mistake

```
// What I am declaring?
int *x, *y;
```

Variable `x` **is a pointer to an integer;** `y` **is also a pointer to an integer.**

- **Pointer Assignment**

→ A pointer only stores the address of a variable, not the address of a literal (constant).

```
int* x = &3;  // Compiler error
```

→ If a pointer is not initialized, what address it points to?

```
int* x;
```

Console

```
0xcccccccc {???}
```

- If a pointer is not initialized, it will point to a memory address that cannot be controlled by the user.
- It is dangerous (unreliable programming), since the user may overwrite the important data stored in that memory address.

→ It is a good practice to always initialize a pointer to safe value.

```
int *x = NULL;
```

```
int *x = nullptr;
```

NULL	nullptr
• (Pros) It can be used in any version of C++. • (Cons) You need to include <stddef.h> or <iostream>.	• (Pros) It is a reserved keyword (no need to include any library). • (Cons) This keyword is only available in C++ 11 or later.

- **The NULL Pointer**

→ Expression "x == NULL" is equivalent to "!x == true".

→ Expression "x != NULL" is equivalent to "!x == false".

```
int *x = NULL;
if (!x) { fout << "x is NULL." << endl; }
```

output.txt

```
x is NULL.
```

```
int y = 3;
int *x = &y;
if (x) { fout << "x is not NULL." << endl; }
```

output.txt

```
x is not NULL.
```

- **Letting a pointer point to a variable**

  → **If y is a pointer to integer, then you can set y's value to the address of an integer variable to let y point to that integer.**

```cpp
int x = 2;
int* y;
y = &x;  // y will point to x.
```

```cpp
int x = 2;
int* y = &x;  // Initialize a pointer to let it point to a variable.
```

- **Changing the value of the variable a pointer points to**

```cpp
int x = 3;
int *y = &x;
*y += 3;
fout << x << endl;
```

output.txt
6

  - **Pointer y points to x, so expression *y returns a reference to x (not a copy of x).**
  - **This is why when *y changes, x also changes.**

- **[Challenge] Can you tell the effect of the highlighted statements?**

```cpp
int x = 3;
int *y = &x;
(*y)++;
fout << x << endl;
fout << &x << endl;
fout << y << endl;
```

output.txt
4 0029FC98 0029FC98

```cpp
int x = 3;
int *y = &x;
*(y++);
fout << x << endl;
fout << &x << endl;
fout << y << endl;
```

output.txt
3 002AFC94 002AFC98

```cpp
int x = 3;
int *y = &x;
*y++;
fout << x << endl;
fout << &x << endl;
fout << y << endl;
```

output.txt
3 002DFA10 002DFA14

- **[Challenge] Can you tell the effect of the highlighted statements?**

```
int x = 3;
int *y = &x;
++*y;
fout << x << endl;
fout << &x << endl;
fout << y << endl;
```

output.txt

```
4
0029FC98
0029FC98
```

```
int x = 3;
int *y = &x;
*++y;
fout << x << endl;
fout << &x << endl;
fout << y << endl;
```

output.txt

```
3
003FFB7C
003FFB80
```


- **[Exercise] Show how p1 and p2 are changed in each step.**

```
ofstream fout("output.txt");
int x = 5, y = 15;
int *p1 = &x, *p2 = &y;
*p1 = 10;
*p2 = *p1;
p1 = p2;
*p1 = 20;
fout << x << endl;
fout << y << endl;
```

output.txt

```
10
20
```

- **Pointers and Regular Arrays**

→ **In C++, a regular array (not vector) is always passed by reference (address).**

Therefore, an array is assigned to a pointer without '&'.

```
int arr[5] = { 1, -1, 1, 2, 3 };
int* p = arr;  // Not "int* p = &arr;"
```

- **When a regular array is assigned to a pointer, the pointer will point to the beginning of the array by default.**
- **The "beginning" of the array means the first element of the array.**

→ **[Example] Moving pointer around a regular array**

```
int arr[5] = { 11, 12, 13, 14, 15 };
int* p = arr;  // p points to 11.
p++;  // p points to 12.
*(p + 2) = -14;  // 14 is changed to -14.
p = arr + 2;  // p points to 13.
p = &arr[4];  // p points to 15.
arr[4] = 0;  // 15 is changed to 0.
*(arr + 4) = 1;  // 0 is changed to 1.
```


- **Pointers and Vectors**

→ **What are the differences among p1 to p4 in the code below?**

```
vector<int>   p1;
vector<int*>  p2;
vector<int>*  p3;
vector<int*>* p4;
```

→ **The "->" dereferencing operator**

- **If variable p1 is a vector object (not a pointer), then to access a vector class-member function of p1 (e.g., size()), we use '.' operator (e.g., p1.size()).**
- **If variable p3 is a pointer to vector, then to access a vector class-member function of p3 (e.g., size()), we use "->" operator (e.g., p3->size()).**

```
vector<int> p1 = { 11, 12, 13, 14, 15 };
vector<int>* p3 = &p1;
p3->push_back(16);
if (!p3->empty()) { p3->pop_back(); }
```

- **Pointer and User-Defined Class**

In "Rectangle.h"

```
class Rectangle {
public:
    // Constructor
    Rectangle(unsigned int = 0, unsigned int = 0);
    // Getters
    unsigned int get_width() const;
    unsigned int get_height() const;
    // Setters
    void set_width(unsigned int);
    void set_height(unsigned int);
    // Class-member functions
    unsigned int area() const;
    unsigned int perimeter() const;
    // Operator
    friend ostream& operator << (ostream&, const Rectangle&);

private:
    // Data fields
    unsigned int width, height;
};
```


In "Rectangle.cpp"

```
// Constructor
Rectangle::Rectangle(unsigned int width, unsigned int height) :
    width(width), height(height) {}

// Getters
unsigned int Rectangle::get_width() const { return width; }
unsigned int Rectangle::get_height() const { return height; }

// Setters
void Rectangle::set_width(unsigned int w) { width = w; }
void Rectangle::set_height(unsigned int h) { height = h; }

// Class-member functions
unsigned int Rectangle::area() const { return width * height; }
unsigned int Rectangle::perimeter() const {
    return 2 * (width + height);
}

// Stream insertion operator
ostream& operator << (ostream& out, const Rectangle& rect) {
    out << "Width: " << rect.width << endl;
    out << "Height: " << rect.height;
    return out;
}
```

In "Main.cpp"

```cpp
int main() {
    Rectangle rect;
    Rectangle* p = &rect;
    p->set_width(5);
    p->set_height(10);
    cout << *p << endl;
    system("pause");
    return 0;
}
```

→ **Although you can create pointers to user-defined classes (the example above), it is not a typical way to use C-style pointers on C++ classes.**

You need to use C++ pointers (the `new` operator) on C++ classes.

In "Main.cpp"

```cpp
int main() {
    Rectangle* p = new Rectangle();
    p->set_width(5);
    p->set_height(10);
    cout << *p << endl;
    delete p;
    system("pause");
    return 0;
}
```


- **The `const` Pointers**

→ **Using keyword `const`, you can make a pointer a `const` pointer.**

```cpp
const int* q;
```

→ **A `const` pointer cannot change the variable the pointer is pointing to, but the pointer itself can be changed to point to another variable.**

```cpp
int arr[5] = { 11, 12, 13, 14, 15 };
const int* p = arr;
fout << *p << endl;  // Correct
*p = 20;  // Incorrect
p = arr + 1;  // Correct
p++;  // Correct
(*p)++;  // Incorrect
```

- **A `const` pointers cannot be cast to a pointer.**
- **A pointer can be cast to a `const` pointer.**

You can treat your smartphone as a calculator; you cannot treat your calculator as a smartphone.

→ **[Pitfall] Creating a pointer to a const variable**

```cpp
const int t = 5;
const int* p1 = &t;  // Correct
int* p2 = &t;  // Incorrect
```

→ **If a variable is a `const` variable, the you can only create a `const` pointer to point to the variable.**

- **<u>Pointer and String Literal</u>**

```cpp
char* p1 = "Hello World";
const char* p2 = "Hello World";
```

→ **The string literal will be treated as <u>an array of characters</u>.**

- **The pointer must point to character.**
- **The string literal will be treated as a <u>C-string</u>.**

- **<u>Pointer to Pointer?</u>**

```cpp
int** p = NULL;  // What is the data type of p?
```

- **In C-style pointers, `p` is <u>a pointer to pointer to integer</u>.**
- **In C++ pointers, `p` is <u>a two-dimensional dynamic array</u>.**

- **<u>Pointer `this`</u>**

→ **In C++, when a <u>class object</u> is created, a <u>`this` pointer</u> is also initialized.**

`this` pointer <u>points to the current class object</u> (not the ~~current class~~).

In Python, we have something called `self`; in Visual Basic, we have something called `Me`.

→ **In C++, `this` is a <u>pointer</u> that points to the <u>current class object</u>.**

In "Rectangle.cpp"

```cpp
// Overloading the "less-than" operator
bool Rectangle::operator < (const Rectangle& other) const {
    return this->area() < other.area();
}
```

In this case, "`this->`" can be omitted, since "`area()`" itself calls the class-member function of the current class object.

In "Rectangle.cpp"

```cpp
// Setter of "width"
void Rectangle::set_width(unsigned int width) { this->width = width; }
```

In this case, "`this->`" cannot be omitted, since there are <u>two different variables named `width`</u>. One is a class data field and the other is the function argument.

→ Since this is a pointer to the current class object, so *this is a reference (not a ~~copy~~) to the current class object.

→ this pointer cannot be called in ~~static functions~~, since static functions belong to classes, not ~~objects~~.

→ this pointer cannot be called in ~~friend functions~~, since friend functions are non-class-member functions.

- C++ Dynamic Memory Allocation

→ The key difference between C++ pointers and C-style pointers is that C++ pointers support dynamic memory allocation, using the new keyword.

```
int* p;  // A pointer to an integer
p = new int(4);  // Dynamic memory allocation
```

- In this example, the new keyword let the program allocate "a chunk of memory" that is just enough (no ~~more~~ no ~~less~~) to store an integer.
- Then, integer 4 is stored in the allocated chunk of memory.
- Finally, p is a pointer that points to the dynamically-allocated memory that stores integer 4.

→ [Exercise] Describe the effects of the statements below.

```
string* p = new string("xyz");
Rectangle* q = new Rectangle(3, 5);
```


→ After using the dynamically-allocated memory, the user must release the memory so that the chunk of memory can be assigned to other processes.

The delete keyword is used to free up dynamically-allocated memory.

```
int* p = new int(4);
cout << *p << endl;
delete p;  // Dynamic memory pointer "p" is pointing to is freed up.
p = NULL;  // Set "p" to safe value after delete it.
```

- The delete keyword can only delete dynamically-allocated memory (C++ pointers) created via the new keyword; it cannot delete any ~~static memory~~, no matter the variable is a pointer or not.

```
int* p = new int(3);
delete p;
p = NULL;
```

Correct

```
int x = 10;
int* q = &x;
delete x;  // Not dynamic memory
delete q;  // Not dynamic memory
```

Incorrect

- After deleting the dynamic memory a pointer points to, the pointer will point to a random location in the memory (not ~~safe value~~).
- [Good Habit] Always set a pointer to safe value after deleting it.

→ Two versions of delete statement:

delete (used to delete a pointer that is not a ~~dynamic array~~)
delete[] (used to delete a pointer that is a dynamic array)

- **Two important issues with dynamic memory allocation in C++**

→ **Shallow copy and deep copy**

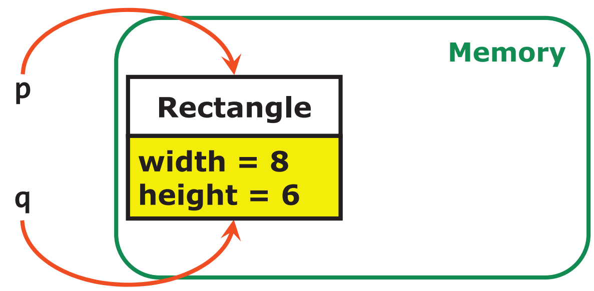
Shallow copy

```cpp
Rectangle* p = new Rectangle(5, 9);
Rectangle* q = p;  // "q" is a shallow copy of "p".
```

p and q point to the same object in the memory.

If either pointer is changed, then the other one is also changed.

```cpp
// Before change
cout << *p << endl;
cout << *q << endl;
// Change pointer "p".
p->set_width(8);
p->set_height(6);
// After change
cout << *p << endl;
cout << *q << endl;
```



Console

```
Rectangle(width=5, height=9)
Rectangle(width=5, height=9)
Rectangle(width=8, height=6)
Rectangle(width=8, height=6)
```

- **Advantage of shallow copy: fast process (no ~~actual data copied~~)**
- **Disadvantage of shallow copy: may be unreliable in execution**

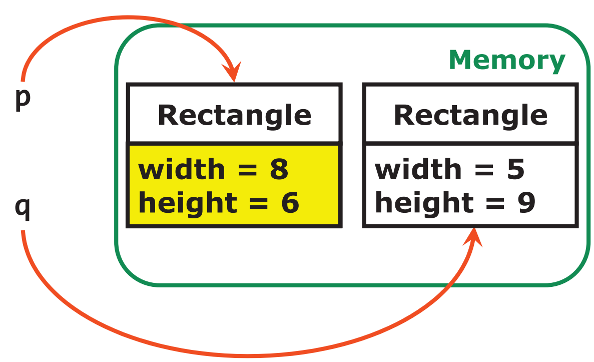
Deep copy

```cpp
Rectangle* p = new Rectangle(5, 9);
Rectangle* q = new Rectangle(p->get_width(), p->get_height());
// "q" is a deep copy of "p".
```

p and q point to different objects in the memory.

If either pointer is changed, it will not affect the other one.

```cpp
// Before change
cout << *p << endl;
cout << *q << endl;
// Change pointer "p".
p->set_width(8);
p->set_height(6);
// After change
cout << *p << endl;
cout << *q << endl;
```



Console

```
Rectangle(width=5, height=9)
Rectangle(width=5, height=9)
Rectangle(width=8, height=6)
Rectangle(width=5, height=9)
```

- **Advantage of deep copy: safe programming (reliability enhanced)**
- **Disadvantage of deep copy: slow process (actual data copied)**

Both shallow copy and deep copy are widely used in computer programming.

→ **Memory leak**

```
Rectangle* p = new Rectangle(5, 9);
p = new Rectangle(8, 6);  // "p" point to another dynamic memory.
```

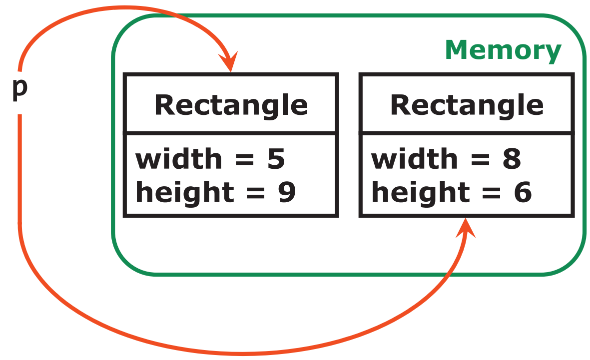
Initially, a chunk of memory is allocated and a `Rectangle` object is stored in it.

`p` points to the `Rectangle` object.

Then, another chunk of memory is allocated and a `Rectangle` object is stored in it.

`p` points to the new `Rectangle` object.

How about the old `Rectangle` object?



- The allocated memory storing the old `Rectangle` object is not ~~released~~ (no ~~delete statement~~).
- However, we lose the reference to ~~access that chunk of memory~~, because `p` was the only reference to access that chunk of memory and it is now pointing to another chunk of memory.
- The chunk of memory storing the old `Rectangle` object is both unusable and unreleased, which means it cannot be assigned to some other process (it stays there forever and memory space is wasted).

This issue is called memory leak that must be avoided in the program.

→ **How to avoid memory leak?**

You need to use the `delete` statement to release the dynamic memory a pointer points to before let it point to another dynamically-allocated memory.

→ **C++ does not do ~~garbage collection~~ for the user.**

It is the user's responsibility to take good care of the allocation and release of all dynamic memory.

→ **Do not forget the "house-keeping" in your program.**

In your assignments/projects, if your program has memory leak in it, you will lose points.

- **Pointers in User-Defined Classes**

→ User-defined classes discussed before did not have ~~pointers~~ involved.

→ If a class data fields have variables that are pointers, then before we can use the class as other classes (that do not have ~~pointers~~), some extra work must be done (see below):

1) Overload the assignment operator '=' (deep-copy assignment).

2) Overload the copy constructor.

3) Overload the destructor.

→ These are called "the big three" or "rule of three".

- **Let's take the Banner class as example.**

→ **[Data Fields] A banner has a `width`, `height`, and some `text`.**

`width` and `height` are **unsigned integers**.

`text` is a **pointer to a string**.

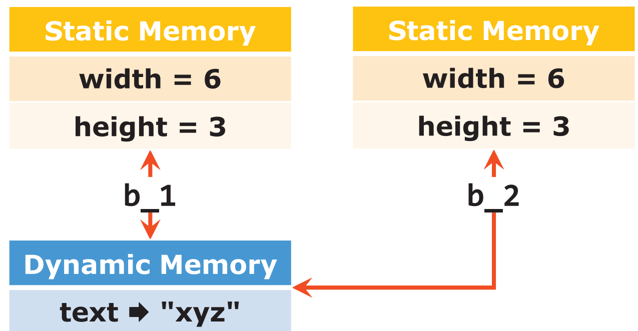
In "Banner.h"

```
private:
    // Data fields
    unsigned int width;  // Stores the width of the banner
    unsigned int height;  // Stores the height of the banner
    string* text;  // Stores the text to show on the banner
```

- **Deep-Copy Assignment Operator**

→ **Even if there is no ~~assignment operator~~ overloaded, we can still use operator '='.**

```
Banner b_1(6, 3, "xyz");
Banner b_2 = b_1;
```



However, for pointer data fields, only shallow copies are made.

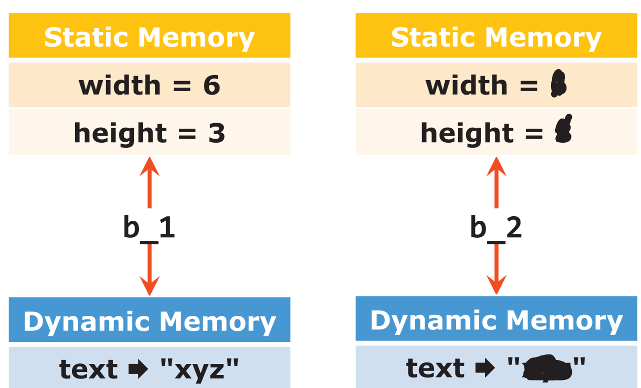
We want b_1 and b_2 to be independent to each other.

Another issue here is that if b_2 already has some dynamic memory associated with it, after the assignment, we will have memory leak.

→ **Basic idea of overloading the deep-copy assignment operator**

```
b_2 = b_1;  // The assignment makes b_2 an identical deep copy of b_1.
```

- **Release the dynamic memory currently associated with b_2 (~~memory leak~~ avoided).**
- **Copy static data from b_1 to b_2.**
- **In b_2, allocate a chunk of memory which is the same size as b_1's dynamic memory.**
- **Copy the data in b_1's dynamic memory to b_2's dynamic memory.**



→ **[Important] Five steps to overload deep-copy assignment operator:**

1) **Avoid ~~self-assignment~~ (if "b_2 = b_2", then do nothing).**
2) **Release dynamic memory.**
3) **Copy static data.**
4) **Copy dynamic data.**
5) **Return.**

In "Banner.cpp"

```cpp
// Deep-copy assignment operator
const Banner& Banner::operator = (const Banner& rhs) {
    // Step 1: avoid self-assignment.
    if (this != &rhs) {
        // Step 2: release currently associated dynamic memory.
        if (text) {
            delete text;
            text = NULL;  // Set to safe value after deleting it.
        }
        // Step 3: copy static data fields.
        width = rhs.width;
        height = rhs.height;
        // Step 4: copy dynamic data fields.
        if (rhs.text) { text = new string(*rhs.text); }
    }
    // Step 5: return.
    return *this;
}
```

- Assignment operator takes a `const` reference as the only argument.
- Assignment operator returns a `const` reference (return type).
- Assignment operator is not ~~a const function~~.
- Return value of assignment operator always `*this`.

- **Copy Constructor**

→ Copy constructor makes a deep copy from another class object.

→ Basic idea of overloading the copy constructor

- Set all the pointers to `NULL`.
- Use the overloaded assignment operator to make a deep copy.

In "Banner.cpp"

```cpp
// Copy constructor
Banner::Banner(const Banner& other) {
    text = NULL;  // Step 1: set all pointers to NULL.
    *this = other;  // Step 2: set "*this" equal to "other".
}
```

- **Destructor**

→ Destructor will be called automatically when the class object is going out of scope (e.g., at the end of the program).

→ Destructor releases all dynamically memory associated with the object.

In "Banner.cpp"

```cpp
// Destructor
Banner::~Banner() { if (text) { delete text; } }
```

- **Dynamic Arrays**

→ **What are the limitations of regular arrays in C++?**

- In `arr[size]` declaration, `size` must be an integer literal or `const` integer variable (cannot be a ~~variable~~).
- An array cannot be ~~resized~~ after declaration.
- Compiler does not ~~check for boundaries~~.

→ **How to use a pointer to create a dynamic array?**

Syntax of creating a one-dimensional dynamic array

```
Item_Type* arr = new Item_Type[size]
```

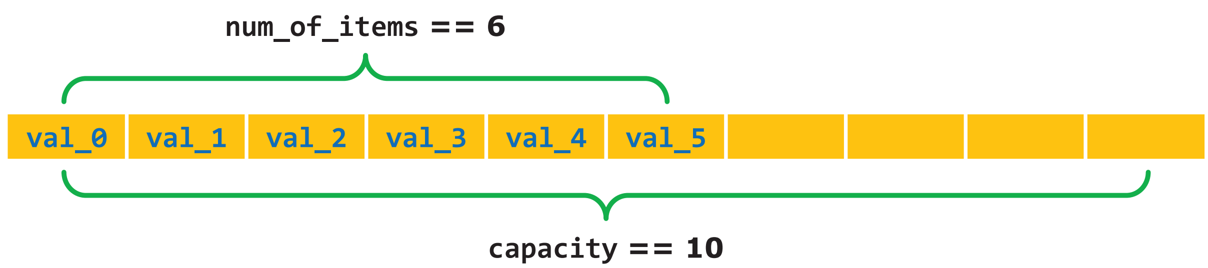
- `size` can be an integer literal or integer variable.
- `arr` is a pointer that points to the dynamic array.
- Elements in the dynamic array can be accessed using `arr[i]`, where `i` is the index of the element being accessed.

- **Let's take a dynamic array of integers as example.**

In "Dynamic_Array.h"

```
static const size_t DEFAULT_CAPACITY;
size_t capacity;  // Stores the capacity of the array.
size_t num_of_items;  // Stores the number of elements in the array.
int* data;  // Stores the elements in the array.
```


→ **What is the difference between `capacity` and `num_of_items`?**



→ **Default capacity**

In "Dynamic_Array.cpp"

```
const size_t Dynamic_Array::DEFAULT_CAPACITY = 10;
```

→ **Default constructor**

In "Dynamic_Array.cpp"

```
Dynamic_Array::Dynamic_Array() :
capacity(DEFAULT_CAPACITY), num_of_items(0) { data = new int[capacity]; }
```

Default constructor creates an empty array.

→ Deep-copy assignment operator

In "Dynamic_Array.cpp"

```
const Dynamic_Array& Dynamic_Array::operator = (const Dynamic_Array& rhs) {
    // Step 1: avoid self-assignment.
    if (this != &rhs) {
        // Step 2: release currently associated dynamic memory.
        if (data) {
            delete[] data;
            data = NULL;  // Set to safe value after deleting it.
        }
        // Step 3: copy static data fields.
        capacity = rhs.capacity;
        num_of_items = rhs.num_of_items;
        // Step 4: copy dynamic data fields.
        data = new int[capacity];
        for (size_t i = 0; i < num_of_items; i++) { data[i] = rhs.data[i]; }
    }
    // Step 5: return.
    return *this;
}
```

You need to use "delete[]" to <u>delete a dynamic array</u>, even if there is only one element in the array.

→ Copy constructor

In "Dynamic_Array.cpp"

```
Dynamic_Array::Dynamic_Array(const Dynamic_Array& other) {
    data = NULL;
    *this = other;
}
```

→ Destructor

In "Dynamic_Array.cpp"

```
Dynamic_Array::~Dynamic_Array() {
    if (data) { delete[] data; }
}
```

→ Subscript operator "[]"

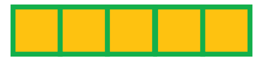
In "Dynamic_Array.cpp"

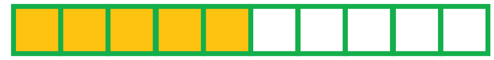
```
// Subscript operator (lvalue)
int& Dynamic_Array::operator [] (size_t index) { return data[index]; }
// Subscript operator (rvalue)
const int& Dynamic_Array::operator [] (size_t index) const {
    return data[index];
}
```

You must overload the "[]" operator twice (lvalue and rvalue).

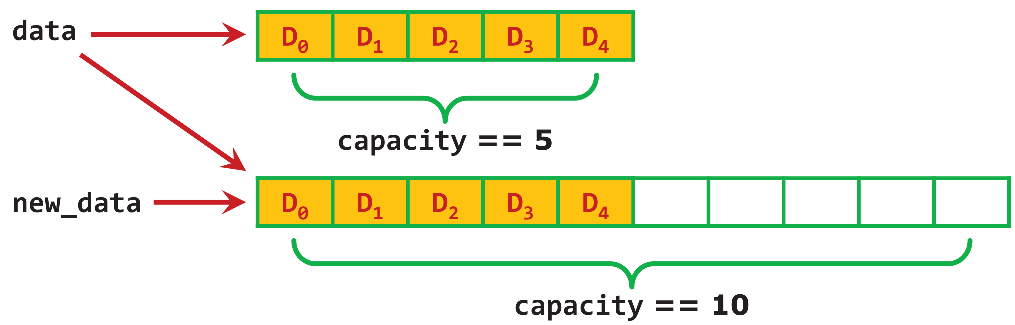
→ The `.resize()` function

The `.resize()` function doubles the capacity of the array but keep the current elements unchanged.

Before: `num_of_items == capacity == 5` 

After: `num_of_items == 5`, `capacity == 10` 

What is the algorithm of `.resize()`?



The `.resize()` function is in the private section of the class, since it is not expected to be called outside the class.

In "Dynamic_Array.cpp"

```
void Dynamic_Array::resize() {
    capacity *= 2;
    int* new_data = new int[capacity];
    for (size_t i = 0; i < size(); i++) { new_data[i] = at(i); }
    delete[] data;
    data = new_data;
}
```

→ The `.push_back()` function

The `.push_back()` function appends a new element to the rear end of the array.

In "Dynamic_Array.cpp"

```
/** Appends a new element to the rear end of the array.
    @param value: new element to append to the array
*/
void Dynamic_Array::push_back(int value) {
    if (size() == capacity) { resize(); }
    data[num_of_items++] = value;
}
```

Please see the lecture sample code for the full class implementation.

C++ uses dynamic array to implement the `vector` class.

- **Two-Dimensional Dynamic Arrays**

```
int* arr_1;  // One-dimensional dynamic array
int** arr_2;  // Two-dimensional dynamic array
```

→ **Initialize a two-dimensional dynamic array with** `num_of_rows` **and** `num_of_columns`**.**

```
arr = new int*[num_of_rows];  // Create the outer array.
for (int i = 0; i < row_of_rows; i++) {
    arr[i] = new int[num_of_columns];  // Create the inner arrays.
}
```

→ **Delete a two-dimensional dynamic array.**

"`delete[] arr;`" **is incorrect.**

```
for (int j = 0; j < row; j++) {
    delete[] arr[j];  // Delete the inner arrays.
}
delete[] arr;  // Delete the outer array.
```