



SCHOOL OF COMPUTER SCIENCES  
UNIVERSITI SAINS MALAYSIA  
Semester II, Session 2023/2024

---

CPT113 – Programming Methodology & Data Structure  
Tutorial Week 7  
Pointers and Dynamic Variables

Learning Outcomes:

- Understanding the use of pointers and dynamic variables in class
- 

1. Refer to the given source code, compile and run the program. Follow the instructions in the comments and observe the effect of modifying the code on the output of the program.

```
/* This program demonstrates deep and shallow copy for objects
Instructions:
    1. Remove comment for CopyConstructor, then compile and run the program. Observe
       the output. Relate the code with the output. Discuss your findings.
    2. Comment out CopyConstructor, then compile and run the program. Observe the
       output.
    3. Relate with the code and output. Discuss it.

*/
#include <iostream>
using namespace std;

class ShallowC
{
private:
    int * x;
public:
    ShallowC(int m)
    {
        x = new int; } shallow.
        *x = m;
    }
    ↴ address
    //CopyConstructor → deep
    /*ShallowC(const ShallowC& obj)
    {
        x = new int; /
        *x = obj.GetX();
    } */
    ↴
int GetX() const
{
    return *x;
```

```

    }

    void SetX(int m)
    {
        *x = m;
    }

    void PrintX()
    {
        cout << "Int X=" << *x << endl;
    }

    ~ShallowC()
    {
        delete x;
    }
};

int main()
{
    ShallowC ob1(10);
    ShallowC ob2 = ob1 ;
    ob1.PrintX();
    ob2.PrintX();
    ob1.SetX(12);
    ob1.PrintX();           x = *x
    ob2.PrintX();
}

```

**Output:**

```

Int x=10 10      ob1   ob2
Int x=10 10      10     10
Int x=12 12      12     12
Int x=10

```

//When the copy constructor was uncommented, it performed a deep copy, the value of x from the other object was copied over using the dereferencing operator.

```

Int x=10
Int x=10
Int x=12
Int x=12

```

//When the copy constructor is commented out, the program performed a shallow copy, whereby the pointer in the current object is made to point to the pointer of the other object. //May also lead to a warning (double free) due to the shallow copy. Both objects are pointing to the same memory location. After one object goes out of scope, the destructor is called, and memory is deallocated. The Second object will also try to deallocate the same memory location, but it's already empty. This results in undefined behavior (trying to free the same memory location twice).

2. Refer to the given source code, compile & run a program. Follow the instructions in the comments and observe the effect of modifying the code on the output of the program. Explain what has happened.

```
/* This program demonstrates how assignment operators work with/without dynamic array
```

**Instructions:**

1. Uncomment **Method 1**, then compile and run the program. Observe the output and relate it with the code. Discuss it.
  2. Comment out **Method 1**. Then uncomment **Method 2**.
  3. Compile and run the program. Observe the output and relate it with the code. Discuss it.
- \*/

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

class cwork
{
    private:
        int matric;
        int *p;
    public:
        void createDynamicArray(int size)
        {
            p=new int [size];
        }

        void setData(int m,int t1,int t2)
        {
            matric=m;
            p[0]=t1;
            p[1]=t2;
        }

        cwork()
        {
            matric=0;
            p=NULL;
        }

        ~cwork() { }

//Method 1
/* overloading of assignment operator
   void operator = (const cwork &cw) {
       matric = cw.matric;
       p=cw.p;
   } */

//Method 2
/* const cwork operator=(const cwork& rightObject)
{
    if(this != &rightObject) //avoid self-assignment
    {
        delete [ ] p;
        matric = rightObject.matric;
        p=new int [2];
        for (int i=0;i<2;i++)
            p[i]=rightObject.p[i];
    }

    //return the object assigned
}
```

```

        return *this;
    } */

    void display()
    {
        cout <<"\t" << matric <<": " ; cout <<"\t" << "Test 1 & Test 2:
" << p[0] << "\t" << p[1] << endl;
    }

    void del()
    {
        delete [] p;
    }
};

int main()
{
    cwork student1,student2;
    student1.createDynamicArray(2);
    student1.setData(27913, 75,80);
    cout <<"Display data in object student1: \n";
    student1.display();
    //copy data from object student1 into object student2
    student2=student1; student1.del(); //object student1 delete dynamic array
    cout <<"Display data in object student2: \n";
    student2.display();
}

```

When using Method1, a shallow copy was performed when copying data from student1 to student2. As such, the pointers in both objects are pointing to the same dynamic array. When student1 is deleted, memory for the dynamic array is freed. As such, the pointer in student2 is no longer valid and results in undefined behavior.  
 When using Method2, a deep copy was performed. Therefore, a new dynamic array was created for student2, and the array elements from student1 were copied one by one. Thus, both objects have their own dynamic arrays that are, at this point, equal. When student1 is deleted, this will not affect student 2.

3. Construct a class named Health which consists of two data members: passportNum with type int and a pointer named ptr with type float. Create a dynamic array using pointer ptr to store body temperature and age. You should defined appropriate methods (including copy constructors, overloading of assignment operator, destructors to properly handle dynamic arrays) to complete the class definition. Then write main() function to test your program. \*\*(let student try on their own and submit their answers)

```

#include <iostream>
using namespace std;

class Health{
private:
    int passportNum;
    float *ptr;

```

```
public:
    Health()
    {
        passportNum = 0;
        ptr = new int[2];
    }
    Health(const Health &otherObj);
    ~Health();
    const Health operator=(const Health &right);
    void setDetails();
    void getDetails();
};

//Copy Constructor
Health::Health(const Health &otherObj)
{
    passportNum = otherObj.passportNum;
    ptr = new int[2];
    ptr[0] = otherObj.ptr[0];
    ptr[1] = otherObj.ptr[1];
}

//Destructor
Health::~Health()
{
    delete [] ptr;
}

//Overload
const Health Health::operator=(const Health &right)
{
    if(this != &right)
    {
        delete [] ptr;
        ptr = new int [2];
        passportNum = right.passportNum;
        ptr[0] = right.ptr[0];
        ptr[1] = right.ptr[1];
    }
    return *this;
}

void Health::setDetails()
{
    cout << "Enter passport Number: ";
    cin >> passportNum;
    cout << "Enter body temperature: ";
    cin >> ptr[0];
    cout << "Enter age: ";
    cin >> ptr[1];
}
```

```

void Health::getDetails()
{
    cout << "Passport Number: " << passportNum << endl;
    cout << "Body temperature: " << ptr[0] << endl;
    cout << "Age: " << ptr[1] << endl;
}

int main()
{
    Health person1, person2;
    person1.setDetails();
    person2 = person1;
    person1.getDetails();
    person2.getDetails();
    Health person3 = person1;
    person3.getDetails();
    return 0;
}

```

### True/False

Indicate whether the statement is true or false.

T 4. A pointer variable is a variable whose content is a memory address.

F 5. In C++, no name is associated with the pointer data type.

F 6. In the statement

```
int* p, q;
```

p and q are pointer variables.

T 7. The *address operator* is a unary operator that returns the address of its operand.

F 8. In C++, the asterisk character is only used as the binary multiplication operator.

F 9. In C++, &p, p, and \*p all have the same meaning.

F 10. In C++, the dot operator has a lower precedence than the dereferencing operator.

F 11. Variables that are created during program execution are called `static` variables.

F 12. In C++, new is a reserved word; however, delete is not a reserved word.

T 13. A memory leak is an unused memory space that cannot be allocated.

T 14. The statement `delete p;` deallocates the dynamic variable pointed to by p.

T 15. Two pointer variables of the same type can be compared for equality.

F 16. If p is a pointer variable, the statement `p = p * 2;` is valid in C++.

T 17. If p is a pointer variable, the statement `p = p + 1;` is valid in C++.

F 18. Pointer arithmetic is the same as ordinary arithmetic.

T 19. Given the declarations

```
int list[10];
int *p;
```

the statement

```
p = list;
```

is valid in C++.

### Multiple Choice

*Identify the choice that best completes the statement or answers the question.*

a 20. In C++, you declare a pointer variable by using the \_\_\_\_ symbol.

- |      |      |
|------|------|
| a. * | c. # |
| b. & | d. @ |

c 21. The code `int *p;` declares p to be a(n) \_\_\_\_ variable.

- |        |            |
|--------|------------|
| a. new | c. pointer |
| b. num | d. address |

a 22. In C++, \_\_\_\_ is called the address operator.

- |      |       |
|------|-------|
| a. & | c. #  |
| b. * | d. -> |

a 23. Which of the following correctly declares a pointer variable p?

- |                            |                            |
|----------------------------|----------------------------|
| a. <code>int *p;</code>    | c. <code>int p; q*;</code> |
| b. <code>int* q, p;</code> | d. <code>*int p;</code>    |

d 24. What is the value of x after the following statements execute?

```
int x = 25;
int *p;
p = &x;
*p = 46;
```

- |         |       |
|---------|-------|
| a. NULL | c. 25 |
| b. 0    | d. 46 |

d 25. What is the output of the following statements?

```
int x = 33;
int *q;
```

```
q = &x;  
cout << *q << endl;
```

- a. NULL
- b. 0
- c. 3
- d. 33

**d** 26. What is the output of the following code?

```
int *p;  
int x;  
x = 76;  
p = &x;  
*p = 43;  
cout << x << ", " << *p << endl;
```

- a. 76, 76
- b. 76, 43
- c. 43, 76
- d. 43, 43

**b** 27. What is the output of the following code?

```
int *p;  
int x;  
x = 12;  
p = &x;  
cout << x << ", "  
*p = 81;  
cout << *p << endl;
```

- a. 12, 12
- b. 12, 81
- c. 81, 12
- d. 81, 81

**b** 28. The only number that can be directly assigned to a pointer variable is \_\_\_\_.

- a. -1
- b. 0
- c. 1
- d. 2

**b** 29. Which of the following can be used to initialize a pointer variable?

- a. 1
- b. NULL
- c. "0"
- d. '0'

**b** 30. The C++ operator \_\_\_\_ is used to create dynamic variables.

- a. dynamic
- b. new
- c. virtual
- d. dereferencing

**b** 31. The C++ operator \_\_\_\_ is used to destroy dynamic variables.

- a. destroy
- b. delete
- c. \*
- d. ~

**c** 32. Which of the following operations is allowed on pointer variables?

- a. exp
- b. %
- c. ==
- d. /

**a** 33. Which of the following arithmetic operations is allowed on pointer variables?

- a. Increment
  - b. Modulus
  - c. Multiplication
  - d. Division
- d 34. Given the statement `double *p;`, the statement `p++;` will increment the value of `p` by \_\_\_\_ bytes.
- a. one
  - b. two
  - c. four
  - d. eight
- a 35. Given the declaration `int *a;`, the statement `a = new int[50];` dynamically allocates an array of 50 components of the type \_\_\_\_.
- a. `int`
  - b. `int*`
  - c. pointer
  - d. address
- d 36. An array created during the execution of a program is called a(n) \_\_\_\_ array.
- a. list
  - b. static
  - c. execution
  - d. dynamic
- b 37. In a \_\_\_\_ copy, two or more pointers of the same type point to the same memory.
- a. static
  - b. shallow
  - c. dynamic
  - d. deep
- b 38. In a(n) \_\_\_\_ copy, two or more pointers have their own data.
- a. shallow
  - b. deep
  - c. static
  - d. dynamic
- b 39. A class \_\_\_\_ automatically executes whenever a class object goes out of scope.
- a. constructor
  - b. destructor
  - c. pointer
  - d. exception
- c 40. The default member-wise initialization is due to the constructor, called the \_\_\_\_ constructor.
- a. init
  - b. new
  - c. copy
  - d. pointer
- b 41. The \_\_\_\_ constructor is called when an object is passed as a (value) parameter to a function.
- a. default
  - b. copy
  - c. struct
  - d. class
- a 42. In C++, virtual functions are declared using the reserved word \_\_\_\_.
- a. `virtual`
  - b. `private`
  - c. `public`
  - d. `struct`
- a 43. In \_\_\_\_ binding, the necessary code to call a specific function is generated by the compiler.
- a. static
  - b. dynamic
  - c. shallow
  - d. deep
- c 44. Run-time binding is also known as \_\_\_\_ binding.
- a. static
  - b. shallow
  - c. dynamic
  - d. deep