Chapter 2 - Pointers

- Learning outcomes:

  - *Describe the idea of a pointer*
  - *Apply declaration & initialization of pointers*
  - *Perform assignments & arithmetic of pointers*
  - *Relate pointer & array, discuss array of pointers*

  - *Describe dynamic memory allocation*
  - *Create & apply dynamic variables, and arrays*
  - *Explain memory leak and dangling pointers*

Introduction

- Variable
  - can be seen as a memory cell, accessed using an identifier.
  - has a unique address in the memory, & addresses follows successive pattern.
  - an address of a variable in the memory is known as *reference*
    - e.g., `0x61fe1c` *// 8-bit, hexadecimal value*

- Pointer
  - a variable that stores the address of another variable

Cont'd…

- Some uses of pointers:
  - allow access to memory locations, which regular variables do not
  - allow to work with strings & arrays, more efficiently
  - allow to create arbitrarily-sized array of values in the memory
  - allow to create "dynamic variables", or to dynamically allocate memory

Cont'd…

- Reference operator (**&**)
  - **&** also known as "address of operator"
  - return the address of a variable in the memory

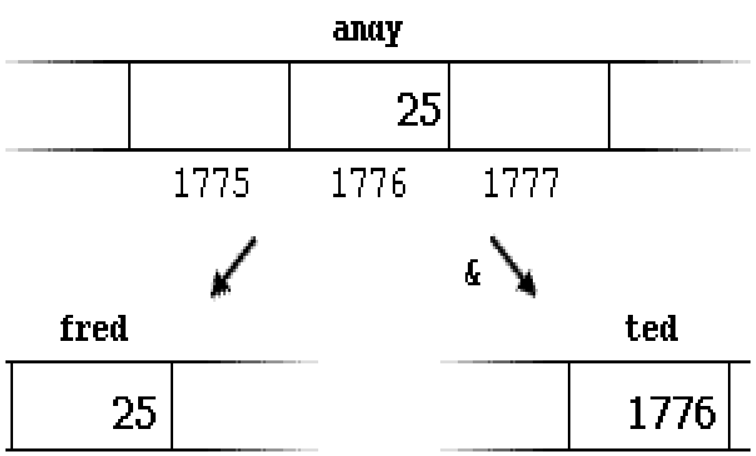
- Examples,
  - `int num=23;`

    `cout<<#` *// output the address of num*

  - `int andy=25;`

    `int fred=andy;`

  // assign the address of andy to ted
  - `ted=&andy;`

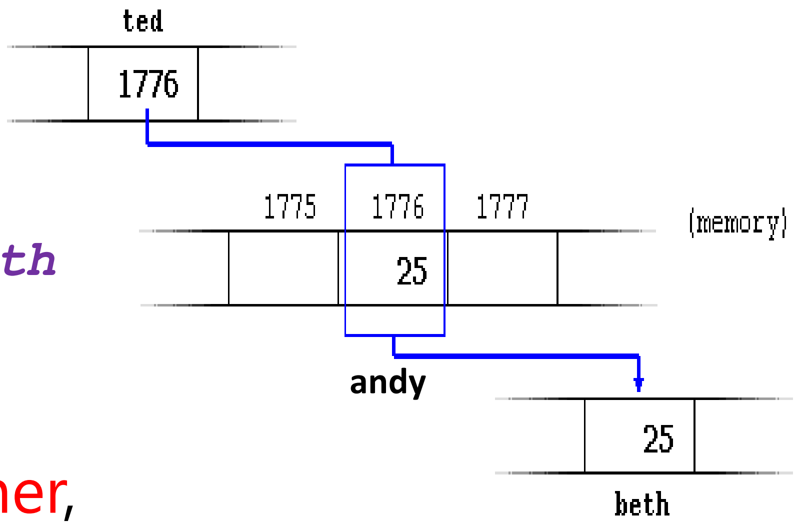


Cont'd…

- Dereference operator (*****)
  - ***** also known as "indirection operator"
  - return the value pointed to by a pointer

- Example
  - `int andy = 25;`
    `ted = &andy;`

  // assign the value pointed by `ted` *to* `beth`
  - `int beth;`
    `beth = *ted;`



- ***** and **&** are inverse of each other,
  - e.g., `cout<<(*)&ted;`
    `cout<<(&)*ted;` *// both outputs the address of* `andy`

Declaring Pointer

- Syntax: `type *pointerName;` *// * indicate a pointer*
  - e.g., `int *myPtr;` *// declare a pointer `myPtr` to `int` type*
- Can declare a pointer to any data type,
  - e.g., `char *x;` *// declare a pointer to a `char`*
- Pointer type: must be of the same type,
  - e.g., `int x=25;`
    `int *p;` *// `p` can point to `x`*
- Declaring multiple pointers, of the same type,
  - e.g., `int *myPtr1, *myPtr2;`

Initializing Pointer

- Syntax: `type *pointerName = intialValue;`
  - `intialValue`: address of a variable or a **`NULL`** value (or **`0`**)
- Example,
  - `int x;`
    `int *y = &x;` *// initialize pointer `y` with the address of `x`*
- Initial value must have the same type,
  - e.g., `float x;`
    `int *ptr = &x;` *// error: can not assign `float` to `int`*
- If not the same type,
  - need explicit "type casting"

Cont'd…

- If not initialized to an address of a variable,
  - good to initialize to a `NULL` value (or `0`)

- Example,
  - ```
    int *ip = NULL;  // ip is a "Null pointer"
    char *cp = NULL; // ip & cp points to null value
    cout<<ip;  // output 0
    ```

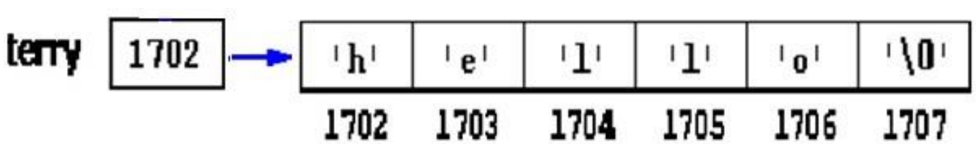
- `NULL` can be initialized, to pointers of any type, but
  - cannot be dereferenced, e.g., `cout<<*ip;` *// cause run time error*

- Pointers can be also initialized with an array name,
  - e.g., `int num[10];`
    `int *ptr = num;` *// num holds address of the first element*
    ```

Cont'd…

- Example,
  - `Cout<<&num[0];`
    `Cout<<ptr;` *// both outputs the same result*

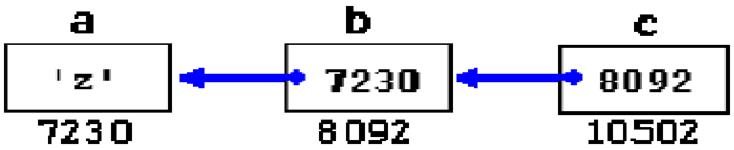
- Pointers can be also initialized to a string constant,
  - e.g., `char *terry = "hello";`



  - create arbitrary-sized array, & store **"hello"** as a string, and
  - create pointer **terry,** & points to the address of the 1st element

Cont'd…

- A pointer to another pointer
  - e.g., `char a='z';`
  - `char *b;`
    `b=&a;`
  - `char **c;`
    `c=&b;`



- Void pointer
  - syntax: `void *ptrName;`
  - also know as "generic" pointer
    - can point to variables of any type
  - cannot be dereferenced *// cause compilation error*

Cont'd…

- Example
  - `int x=5;`
    `void *ptr=&x;`
  - `cout<<ptr;` *// output the address of `x`*
    `cout<<*ptr;` *// error: cannot be dereferenced*
- To dereference,
  - need an explicit "type casting", to another typed pointer
    - e.g., `int *num;` *// because `x` has `int` type*
      `num=(int*)ptr;` *// type cast `ptr` to `int`*
    - `cout<<*num;`
      `cout<<*((int*)ptr);` *// or use it directly*

Constant Pointer

- Constant pointer
  - always point to the same address (or memory location)
  - default for array name *// always point to address of 1st element*
- To declare,
  - syntax: `type *const ptrName = intialValue;`
- Example,
  - `int x, y;`
    `int *const ptr = &x;` *// must be initialized when declared*
  - `*ptr = 7;`
    `ptr = &y;` *// error: cannot change location*

Example 1

```
// Example 1 - Using & and * operators
#include <iostream>

using namespace std;

int main(){

    int a = 7;
    int *ptr;  // pointer to an integer

    ptr = &a;  // ptr assigned address of a

    cout<<"The address of a: "<<&a<<endl;

    cout<<"\nThe value of ptr: "<<ptr<<endl; // address of a

    cout<<"\nThe value of *ptr: "<<*ptr<<endl; // the value pointed by ptr

    cout<<"\nThe address of &ptr: "<<&ptr<<endl; // the address of ptr itself

    cout<<"\n* and & are inverse of each other: "<<endl;

    // go to the value pointed by ptr, and
    // and display the address of that value,
    // which is basically the contents of ptr
    cout<<"\n    (&)*ptr = "<<&*ptr<<endl; // address of a

    // go to the address of ptr, and
    // and display the content of that address,
    // which is basically the address of a
    cout<<"\n    (*)&ptr = "<<*&ptr<<endl; // address of a

    return 0;
}
```

Pointer Assignment

- Pointer to pointer, if the same type,
  - e.g., `int x=2;`
    `int *p1=&x;`
  - `int *p2;`
    `p2=p1;` *// assign the address hold by p1 to p2*
- Do not confuse with,
  - e.g., `*p1=*p2;` *// assign the value pointed by p1 to p2*
- If has no same type, must be type casted
  - e.g., `float *p3;`
    `p3=p1;` *// ~~implicitly~~ upcast p1 to float*

Example 2

```
// Example 2- Pointer assignments

#include <iostream>

using namespace std;

int main(){

    int v1 = 5, v2 = 15;
    int *p1, *p2;

    p1 = &v1;
    p2 = &v2;

    *p1 = 10;
    *p2 = *p1;

    p1 = p2;
    *p1 = 20;

    cout<<"The value of v1: "<<v1<<endl;

    cout<<"\nThe value of v2: "<<v2<<endl;

    return 0;
}
```

Pointer Arithmetic

- Valid operations,
  - increment/decrement pointer (**++** or **--**)
  - add/subtract an integer to/from a pointer(**+** or **-**)
  - subtract one pointer from another pointer
- Pointer arithmetic is meaningful,
  - when performed on a pointer to array
- **`sizeof()`**, unary operator
  - return the size of operand (in bytes)
  - used along with variables, data types, constant values

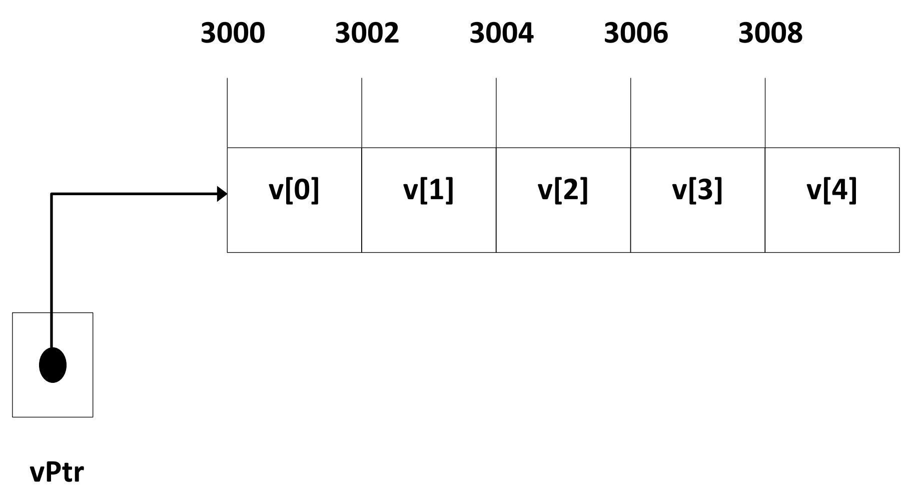
Example 3

```cpp
// Example 3 - Using sizeof ( ) operator
#include <iostream>

using namespace std;

int main(){

     char a;
     long int b;
     int array[20];

     cout<<"The sizeof char: "<<sizeof(a)<<endl;
     cout<<"\nThe sizeof int: "<<sizeof(int)<<endl;
     cout<<"\nThe sizeof  long int: "<<sizeof(b)<<endl;
     cout<<"\nThe sizeof array: "<<sizeof(array)<<endl;

     return 0;
}
```

+/- Integer to Pointer

- Example,
  - `int v[5];` *// assume size of `int` is 2 bytes*

    `int *vPtr = v;` *// vPtr points to `v[0]`*
  - `vPtr += 2;` *// `3000 + 2 (*2) = 3004`*
    - sets `vPtr` to `3004`
    - `vPtr` points to `v[2]`



Pointers Subtraction

- Example,
  - `int *vPtr2;`
    - `vPtr = &v[0];` *// vPtr = 3000*
    - `vPtr2 = &v[2];` *// vPtr2 = 3004*
  - `int n;`

    `n = vPtr2 - vPtr;` *// n equals 2 (elements)*

Pointer ++/--

- Example,
  - `vPtr = v[0];` *// vPtr = 3000*
  - `vPtr++;` *// 3000 + 1*2 = 3002*
    - sets **vPtr** to **3002**
    - **vPtr** points to **v[1]**
  - `vPtr--;` *// 3002 - 1*2 = 3000*
- Also consider,
  - `vPtr = v[0];`

    `cout<<*vPtr++;` *// same as cout<<(*vPtr)++;*

Pointers Comparison

- Equality or relational operators (**</<=, >/>=**)
  - e.g., `if (ptr1 == ptr2)`
    - compare the addresses stored in **ptr1** and **ptr2**
- Pointer comparison is meaningful,
  - when performed on pointers to the same array
- Also commonly used,
  - to check whether a pointer is pointing to **NULL** or not
    - e.g., `if (ptr != NULL)`

Pointer & Array

- Array (name) is like a (constant) pointer,
  - always points to the address of 1st element
- Example,
  - `int vals[]={4, 7, 11};`
  - `cout<<&vals[0];` *// output address of 1st element*

    `cout<<vals;` *// same as `&vals[0]`*
- Array acting-like a pointer variable,
  - e.g., `cout<<*vals;` *// same as `cout<<vals[0]`*

    `cout<<*(vals+2);` *// same as `cout<<vals[2]`*

Cont'd…

- A pointer to an array variable,
  - e.g., `int vals[]={4, 7, 11};`
  - `int *ptr=vals;`

    `cout<<*(ptr+1);` *// output 7*
- Pointer acting-like an array variable,
  - e.g., `cout<<ptr[3];` *// same as* `cout<<vals[3]`
- To get address of individual array elements,

  4
  - e.g., `cout<<&vals[3];` *// output the address the 3rd element*

Cont'd…

- 2D array acting-like a pointer variable,
  - e.g., `int nums[2][3]={{16,18,20}, {25,26,27}};`

    `cout<<*(*(nums+i)+j);` *// same as cout<<nums[i][j]*

	Pointer Notation	Array Notation	Value
i=0	*(*nums + 0)	nums [0][0]	16
	*(*nums + 1)	nums [0][1]	18
	*(*nums + 2)	nums [0][2]	20
i=1	*(*(nums + 1) + 0)	nums [1][0]	25
	((nums + 1) + 1)	nums [1][1]	26
	((nums + 1) + 2)	nums [1][2]	27

Example 4

```cpp
// Example 4 - Pointer and array
#include <iostream>

using namespace std;

int main(){

    int x[5]={5, 10, 15, 20, 25};

    cout<<"The numbers in set are:\n"<<endl;

    for(int i=0; i<5; i++)   //array name as pointer
        cout<<*(x+i)<<" ";

    cout<<endl;

    int *ptr=x, *num=x;

    for(int index=0; index<5; index++) // pointer notation
        cout<<*(ptr+index)<<" ";

    cout<<endl;

    for(int j=0; j<5; j++) // pointer used as array name
        cout<<num[j]<<" ";

    cout<<endl;

    return 0;
}
```

Example 5

```cpp
// Example 5 - Pointer assignment & arithimetic
#include <iostream>

using namespace std;

int main(){

    int num[5];
    int *p;

    p=num;
    *p=10;

    p++;
    *p=20;

    p=&num[2];
    *p=30;

    p=num+3;
    *p=40;

    p=num;
    *(p+4)=50;

    for(int n=0; n<5; n++)
        cout<<num[n]<<" ";

    cout<<endl;
    return 0;
}
```

Example 6

```cpp
// Example 6 - Pointers notation for 2D arrays
#include <iostream>

using namespace std;

int main(){

    int x[2][3]= {{1,2,3}, {4,5,6}};

    for(int i=0; i<2; i++){

    for(int j=0; j<3; j++)
        cout<<*(*(x + i) +j)<<" ";

    cout<<endl;
    }

    return 0;
}
```

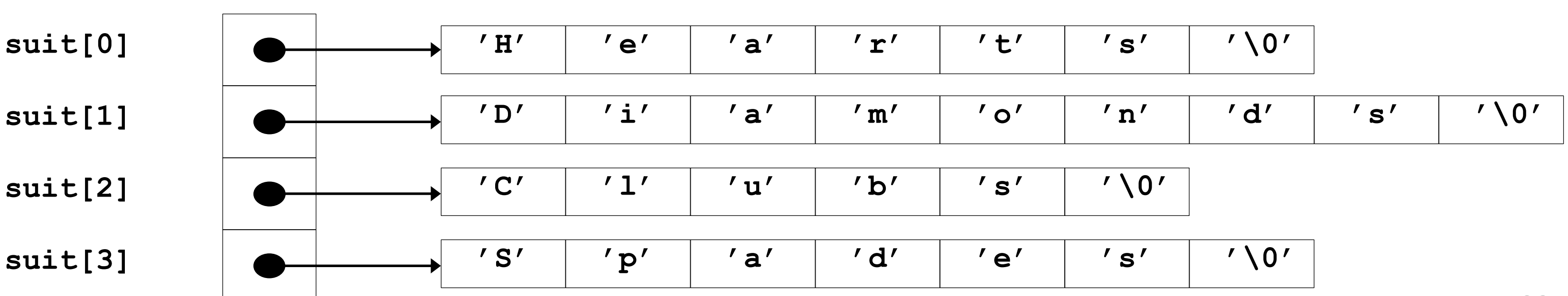
Example 7

```cpp
// Example 7 - declaring a pointer to 2D array
#include <iostream>

using namespace std;

int main(){

    int x[2][3]= {{1,2,3}, {4,5,6}};
    int  (*ptr) [3]= x;

    for(int i=0; i<2; i++){

    for(int j=0; j<3; j++)
        cout<<ptr[i][j]<<" ";

    cout<<endl;
    }
```

??

Array of Pointers

- To declare,
  - syntax: **`type *ptrName[size];`**
- Example,
  - **`char *seasons[4];`**
    - each element point to an address of a **`char`** array
- To initialize,
  - e.g., **`char *suit[4]={"Hearts", "Diamonds‶, "Clubs", "Spades"};`**



Example 8

```
// Example 8 - Array of pointers & string

#include <iostream>

using namespace std;

int main(){

    char *seasons[]={"Winter", "Spring ", "Summer", "Fall"};

    int n, i;

    cout<<"\nEnter a month (use 1 for Jan, 2 for Feb, etc): ";
    cin>>n;

    i=(n%12)/3;

    cout<<"\nThe month entered is: "<<seasons[i]<<endl;

    return 0;
}
```

Dynamic Memory Allocation

- Variables declared,
  - enough amount of memory space reserved, to hold the data type
  - and the space is allocated, during the program compilation time,
    - or before the program execution begins
  - this kind of allocation known as "static" memory allocation,
    - where space is reserved for variables on "stack memory"
- What if we want the user to determine the amount of memory required for a variable, as the program is running?
  - this kind of allocation known as "dynamic" memory allocation,
    - where space is reserved for variables on "heap memory",
    - using a pointer variable

Cont'd…

- Dynamic memory allocation, can be useful,
  - to create an array, where the size was unknown, until run time
  - to create complex data structures, where the size were unknown, and/or need to be constructed as the program was running
  - to create objects of complex structures, where the constructor arguments were unknown, until run time

Dynamic Variable

- To create dynamic variable,
  - need to use a pointer variable
  - created with a **`new`** operator

- Dynamic variables
  - created & destroyed while the program is running
  - **`new`** operator: used to allocate memory dynamically ***// on heap***
  - **`delete`** operator: used to deallocate memory ***// free allocated space***

New Operator

- `new` operator
  - If memory available, allocate memory dynamically
  - & return the address of the new memory location to a pointer
- To create, a dynamic variable
  - syntax: `type *pointerName = new type;`
- Example
  - `int *bobby = new int;`
    - create a new "nameless" variable,
    - and assign **bobby** to "point to" it

Example 1

```
// Example 1 - Create dynamic variable
#include <iostream>

using namespace std;

int main(){

    int *p1, *p2;

    // create a new int variable
    // & assign p1 to point to it
    p1 = new int;

    *p1 = 42;
    p2 = p1;

    cout<<"*p1 = "<<*p1<<endl;

    cout<<"*p2 = "<<*p2<<endl;

    *p2 = 53;

    cout<<"\n*p1 = "<<*p1<<endl;

    cout<<"*p2 = "<<*p2<<endl;

    p1 = new int;

    *p1 = 88;

    cout<<"\n*p1 = "<<*p1<<endl;

    cout<<"*p2 = "<<*p2<<endl;

    return 0;
}
```

Cont'd…

- Stack memory
  - allocated for automatic variables & function variables
- Heap memory
  - allocated for dynamic variables ***// created using `new` operator***
- If sufficient memory not available,
  - **`new`** operator return an exception error, ***// known as "`bad_alloc`"***
  - and the program crash or terminate.
- Warning:
  - always check the requested memory was allocated successfully?

Cont'd…

- C++ standard approaches,
  - handling exceptions
    - using **`try-throw-catch`** statement,
  - or using **`nothrow`** object (with **`new`** operator)
    - **`new`** operator return **`NULL`** value,

  // execution of the program continues

- Exception handling
  - the most used approach
  - not to be discussed here (CH 7)

Cont'd…

- Using the **`nothrow`** object
  - e.g., **`int *bobby;`**

    `bobby = new (nothrow) int [5];`

  - to check the memory was allocated successfully?

```
if (bobby == NULL)
{
    statement;  // print error
     ...... // apply fixes
}
```

Dynamic Array

- Standard array
  - e.g., **`int x[10];`**
  - maximum size must be specified first,
    - actual size may not be known sometimes, until program runs
  - thus, may lead to memory wastage

- Dynamic array
  - size is not specified, at programming time
    - can be determined while the program is running
  - syntax: **`type ptrName = new type [size];`**

Cont'd…

- Example,
  - `int *bobby = new int[i];`
    - assigns space for **`i`** elements of type **`int`**
    - and return a pointer to the 1st element
  - the value of **`i`** is determined by user, as the program is running
  - `cout<<*bobby;` *// same as bobby[0]*

Example 2

```cpp
// Example 2 - Dynamic array
#include <iostream>

using namespace std;

int main(){

    int i, n;
    int *p;

    cout<<"Hw many integers do you want to store? ";
    cin>>i;

    p = new int [i]; // create dynamic array

    for(n=0; n<i; n++){
        cout<<"\nEnter number "<<(n+1)<<" : ";
        cin>>p[n];
    }

    cout<<"\nArray elements: ";

    for(n=0; n<i; n++)
        cout<<p[n]<<" ";

    cout<<endl;

    return 0;
}
```

Delete Operator

- **`delete`** Operator
  - deallocate dynamic variables *// created by **`new`** operator*
    - allocated memory is released to heap memory,
    - the pointer is then become unassigned pointer
- To delete a dynamic variable,
  - syntax: **`delete pointerName;`**
  - e.g., **`int *p = new int;`**

    `delete p;`
  - **`delete [] pointerName;`** *// for 2D dynamic array*

Example 4

```cpp
// Example 4 - Delete dynamic variable
#include <iostream>

using namespace std;

int main(){

    int *ptr;

    ptr = new int;

    *ptr = 22;

    cout<<*ptr<<endl;

    delete ptr;

    return 0;
}
```

Example 5

```cpp
// Example 5 - Delete @D dynamic array
#include <iostream>

using namespace std;

int main(){

    int *grades = NULL;
    int num;

    cout<<"Enter the number of grades: ";
    cin>>num;

    grades = new int [num];

    for(int i=0; i<num; i++)
        cin>>grades[i];

    cout<<"\nGrades: ";

    for(int j=0; j<num; j++)
        cout<<grades[j]<<" ";

    cout<<endl;

    delete [] grades;

    return 0;
}
```

Memory Leak & Dangling Pointer

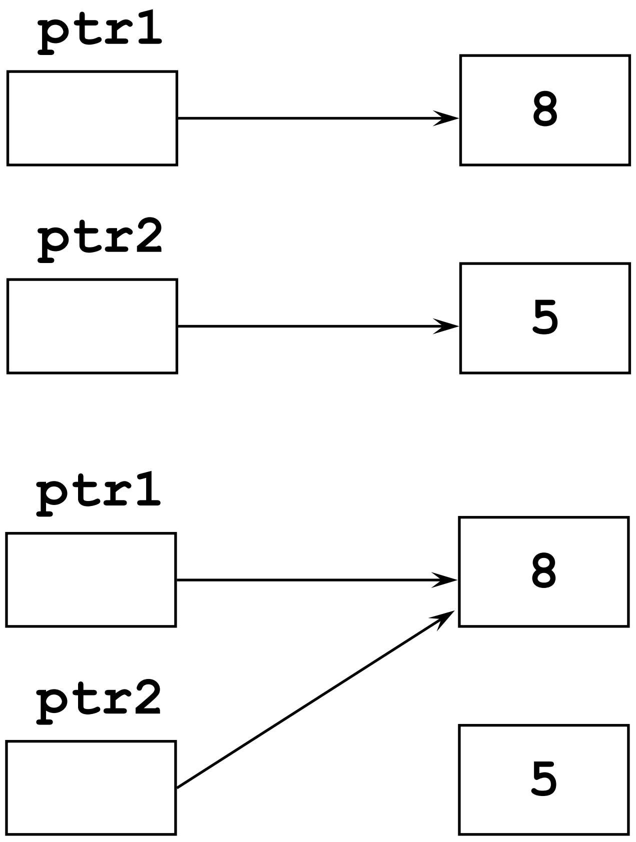
- Memory leak
  - pointer reassigned, without deleting the previous memory
  - result in the loss of memory space
  - solution: free the memory after use, i.e., `delete p;`
- Dangling pointer
  - contain the address of memory that has been deleted
  - dereferencing, may produce unpredictable result
  - solution: assign pointer to **`NULL`**, as soon as the memory is deleted
    - i.e., `delete p;`

      `p = NULL;`

Example 6

```
int *ptr1 = new int;

int *ptr2 = new int;

*ptr1 = 8;

*ptr2 = 5;

ptr2 = ptr1;
```

Solution: __________?

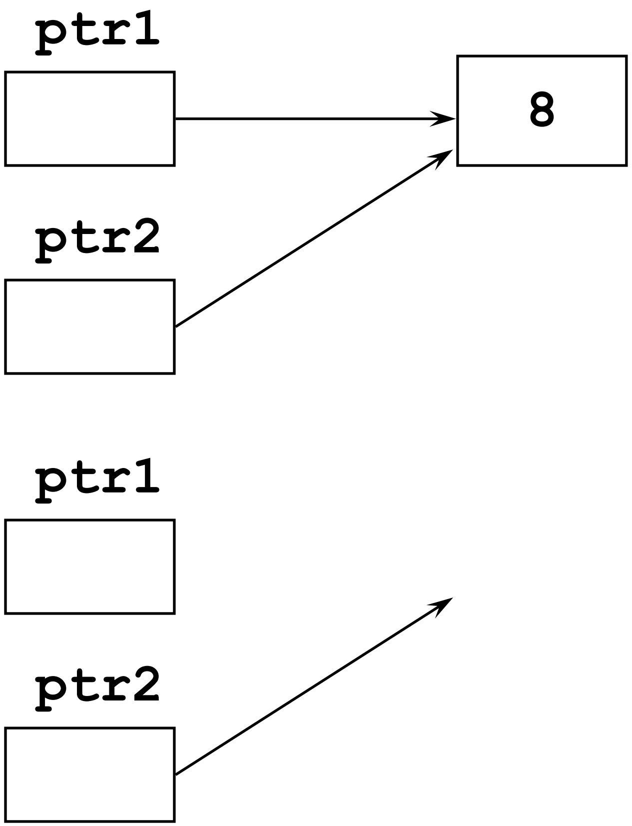


"memory leaked"

Example 7

```
int *ptr1 = new int;

int *ptr2;

*ptr1 = 8;

ptr2 = ptr1;

delete ptr1;
```

Solution: __________?



“dangling pointer”