Practice 12
Pointers

2018 second semester
Computer Engineering Programming
CONTENTS

Pointers

New / delete
Pointing to ...

• Recall:
  \[
  \text{int } *p1, *p2, v1, v2; \\
p1 = \&v1;
  \]

• Two ways to refer to \(v1\) now:
  – Variable \(v1\) itself:
    \[
    \text{cout }<< v1;
    \]
  – Via pointer \(p1\):
    \[
    \text{cout }<<*p1;
    \]

• Dereference operator, \(*\)
  – Pointer variable "derereferenced"
  – Means: "Get data that \(p1\) points to"
```cpp
#include <iostream>
using namespace std;

int main()
{
    int *ptr1, *ptr2, val1=10, val2=20;
ptr1 = &val1;
ptr2 = &val2;
*ptr2 = 50;

cout << "val1:" << val1 << " val2:" << val2 << endl;
cout << "*ptr1:" << *ptr1 << " *ptr2:" << *ptr2 << endl;

cout << "\n\n";
*ptr1 = *ptr2;
cout << "val1:" << val1 << " val2:" << val2 << endl;
cout << "*ptr1:" << *ptr1 << " *ptr2:" << *ptr2 << endl;

cout << "\n\n";
val1 = 10;
ptr2 = ptr1;
cout << "val1:" << val1 << " val2:" << val2 << endl;
cout << "*ptr1:" << *ptr1 << " *ptr2:" << *ptr2 << endl;

return 0;
}
```
Pointer example (via diagram)

```
val1 = 10;
ptr2 = ptr1;
```

```
*ptr1 = *ptr2;
```

```
val1: 10   val2: 50
*ptr1: 10   *ptr2: 50
val1: 50   val2: 50
*ptr1: 50   *ptr2: 50
```

```
val1: 10   val2: 50
*ptr1: 10   *ptr2: 10
계속하려면 아무 키나 누르십시오 . . .
```
The new Operator

- Since pointers can refer to variables...
  - No "real" need to have a standard identifier
- Can dynamically allocate variables
  - Operator new creates variables
    - No identifiers to refer to them
    - Just a pointer!
- \$p1 = \texttt{new int;}$
  - Creates new "nameless" variable, and assigns \$p1\$ to "point to" it
  - Can access with \$*p1\$
    - Use just like ordinary variable
The new Operator example

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

int main()
{
    int *p1, *p2, v2 = 20;
    
p1 = new int;
    cout << "p1:" << p1 << "  p1:" << *p1 << endl;
    // We didn't initialize p2...
    // cout << "p2:" << p2 << "  p2:" << *p2 << endl;
    
cout << "\n\n";
    *p1 = 40;
    cout << "p1:" << p1 << "  p1:" << *p1 << endl;
    
p2 = &v2;
    cout << "p2:" << p2 << "  p2:" << *p2 << endl;
    
cout << "\n\n";
    p2 = p1;
    p1 = new int;
    *p1 = 100;
    cout << "p1:" << p1 << "  p1:" << *p1 << endl;
    cout << "p2:" << p2 << "  p2:" << *p2 << endl;
    
    return 0;
}
```
More on new Operator

• Creates new dynamic variable
• Returns pointer to the new variable
• If type is class type:
  – Constructor is called for new object
  – Can invoke different constructor with initializer arguments:
    ```
    MyClass *mcPtr;
    mcPtr = new MyClass(32.0, 17);
    ```
• Can still initialize non-class types:
  ```
  int *n;
  n = new int(17); //Initializes *n to 17
  ```
Memory Management

• Heap
  – Also called "freestore"
  – Reserved for dynamically-allocated variables
  – All new dynamic variables consume memory in freestore
    • If too many → could use all freestore memory

• Future "new" operations will fail if freestore is "full"
delete Operator

• De-allocate dynamic memory
  – When no longer needed
  – Returns memory to freestore
  – Example:
    ```
    int *p;
    p = new int(5);
    ... //Some processing...
    delete p;
    ```
  – De-allocates dynamic memory "pointed to by pointer p"
    • Literally "destroys" memory
The new / delete Operator example

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

class Money {
public:
    Money(int theDollars, int theCents);
    void printMoney();
private:
    int dollars;
    int cents;
};

int main()
{
    // Method 1 (using new), mPtr1
    Money *mPtr1;
    mPtr1 = new Money(5, 20);
    if (mPtr1 == NULL)
    {
        cout << "Error: Insufficient memory in heap space." << endl;
        exit(1);
    }
    mPtr1->printMoney();
    delete mPtr1; // Return memory to freestore(heap)
    // Destroys dynamic memory but mPtr still points above address!
    mPtr1 = NULL; // Avoid dangling pointers

    // Method 2 (not using new), mPtr2
    Money mPtr2(4, 30);
    mPtr2.printMoney();

    return 0;
}
```

```cpp
Money::Money(int theDollars, int theCents)
    : dollars(theDollars), cents(theCents) {}

void Money::printMoney()
{
    cout << "We have " << dollars << " dollars," << cents << " cents." << endl;
}
```

```
C:\Windows\system32\cmd.exe
We have 5 dollars, 20 cents.
We have 4 dollars, 30 cents.
계속하려면 아무 키나 누르십시오 ...
```
Why do we use ‘new’ operator?

Method 1 (using new)
• Allocates memory for the object on the free store (This is frequently the same thing as the heap)
• Requires you to explicitly delete your object later. (If you don't delete it, you could create a memory leak)
• Memory stays allocated until you delete it. (i.e. you could return an object that you created using new)
• The example in the question will leak memory unless the pointer is deleted; and it should always be deleted, regardless of which control path is taken, or if exceptions are thrown.

Method 2 (not using new)
• Allocates memory for the object on the stack (where all local variables go) There is generally less memory available for the stack; if you allocate too many objects, you risk stack overflow.
• You won't need to delete it later.
• Memory is no longer allocated when it goes out of scope. (i.e. you shouldn't return a pointer to an object on the stack)
• As far as which one to use; you choose the method that works best for you, given the above constraints.
Malloc / New

- malloc() and new are both do “dynamic allocation”.

- Are they same? Then change left code to use malloc instead of new.
### malloc / free vs. new / delete

<table>
<thead>
<tr>
<th>malloc / free</th>
<th>new / delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should specify size</td>
<td>Automatically calculate size</td>
</tr>
<tr>
<td>Allocate memory (O)</td>
<td>Allocate memory (O)</td>
</tr>
<tr>
<td>Initialization (X)</td>
<td>Initialization (O)</td>
</tr>
<tr>
<td>Realloc (O)</td>
<td>Realloc (X)</td>
</tr>
<tr>
<td>Object (X)</td>
<td>Object (O)</td>
</tr>
<tr>
<td>Constructor (X)</td>
<td>Constructor (O)</td>
</tr>
<tr>
<td>Deconstructor (X)</td>
<td>Deconstructor (O)</td>
</tr>
</tbody>
</table>