C++: POINTERS

- Use the keyword new to allocate memory space to a pointer, and delete to clear the memory that the pointer points to.

Example:

```
Passenger *p;
p = new Passenger; // p points to the new Passenger (Allocate memory)
(*p).name = "Pocahontas"; /* change the name of the object that p
points to */
p->name = "Pocahontas"; /* this is an alternative way to change the
name of the object that p point to; think of “->” as an arrow */
p->mealPref = REGULAR; // this is the same as (*p).mealPref = REGULAR
delete p; //clear the memory space that p points to (deallocate memory)
```

C++: REFERENCE

- **Reference:** a variable that refers to another, already-existing variable.
- A variable can be specified as a reference using the ‘&’ operator.

E.g. string choice = “all of the above” ;
string& answer = choice;

/* now, “answer” can be thought of as an alias, i.e., an alternative name for “choice” */

Passing an argument “by value”

```
void f(int x, int y){
   x++; // increase x by 1
   y++; // increase y by 1
}
int main(){
    int n = 5; int m = 10;
    f(n, m);
    cout << n;  // prints 5
    cout << m;  // prints 10
    return EXIT_SUCCESS;
}
```

- Important: The changes that function “f” made to “n” **are not preserved** after “f” terminates.
- This is because “n” was **passed by value** to “f”, i.e., the function was dealing with a *copy* of “n”
- But what if we wanted those changes to be preserved after “f” terminates?

Passing an argument “by reference”:

```
void f(int& x, int y){
   x++; // increase x by 1
   y++; // increase y by 1
}
int main(){
    int n = 5; int m = 10;
    f(n, m);
    cout << n;  // prints 6
    cout << m;  // prints 10
    return EXIT_SUCCESS;
}
```

- When an argument is declared as a reference, e.g., by writing “`int&`” instead of “`int`”, it will be **passed “by reference”** instead of “by value”.
- This way, “f” deals with “n” itself, rather than with a copy of “n”. Thus, any changes that “f” makes to “n” are preserved after “f” terminates

What if we want to pass the variable itself to a function, but without allowing the function to change that variable?

```
void f(int& x){
   cout<< x;
   x++; // f is allowed to change x
}
// Below is the "main" function, which is the part of the code that is executed.
int main(){
    int n = 5;
    f(n); // n is passed by reference.
    return EXIT_SUCCESS;
}
```

Solution: use the keyword "const"

```
void f(const int& x){
   cout<< x;
   x++; // ERROR: f is not allowed to change x, because we added the word "const"
}
// The main function; this is the part of the code that is executed.
int main(){
    int n = 5;
    f(n); // n is passed by reference.
    return EXIT_SUCCESS;
}
```

C++: OPERATOR OVERLOADING

- Operator overloading means defining operators such as:
  - Assignment, e.g., `x = y;`
  - Equality, e.g., `if(x == y)...`
  - Output, e.g., `cout<< x;`

This is for those cases when the type of x is not standard (built-in), i.e., the type of x is defined by the user (e.g. structures and classes)

C++: OPERATOR OVERLOADING

Example 1:

```
Passenger x  = {“John Smith”, LOW_FAT, true, X72199};
Passenger y = {“John Smith”, VEGETARIAN, true, K80006};
if (x==y) cout<< "the two passengers are the same";  /* ERROR: the operator "==" is not
defined for the type "Passenger" */
```

You must first overload the == operator as follows:

```
bool operator==(const Passenger& p1, const Passenger& p2) {
  //write here what you want to happen when writing “p1==p2”
  if (p1.name == p2.name) && (p1.mealPref == p2.mealPref) &&
     (p1.isFreqFlyer == p2.isFreqFlyer) && (p1.freqFlyerNo == p2.freqFlyerNo)
        return true;
  else
        return false; }
```

Now you can use == with variables of type Passenger:

```
if (x==y) cout<< "the two passengers are the same"; // No error :)
```

45

ARRAYS

ARRAYS

- ***Compile time*** is the time when the code you entered is converted to executable
- ***Runtime*** is the time when the executable is running

❖ Static Arrays (created in compile time)

❖ Dynamic Arrays (created in Runtime)

STATIC VS. DYNAMIC ARRAYS

- Static arrays are allocated memory at **compile time** and their size is fixed (their size cannot be changed later)

Example:

```
int x[5];
x[0]=10;  x[1]=20;  x[2]=20;  x[3]=40;

//An alternative way is to write:
int x[] = {10, 20, 30, 40};
```

- If you want to alter the size of your array in **runtime**, use *pointers* and the new operator to create dynamic arrays

Example:

```
int* x; int y; int z;
cin >> y;
x = new int[y]; // Now x is an array of size y
. . .
delete [] x;  // deletes all array elements
cin >> z;
x = new int[z];  // Now x is an array of size z
```

2D ARRAYS

Static 2D Array Example:

```
int m[2][3]= {{1, 2, 3}, {1, 5, 2}} ;
```

Dynamic 2D Array Example:

```
// a double pointer
int** table; //or int **table;
table = new int* [nRows]; //create rows
//create columns
for(int i=0; i<nbRows; i++) {
   table[i] = new int [nCols]; }
// traverse the array rows, then columns
for(int i = 0; i < nRows; i++) {
   for(int j= 0; j < nCols; j++) {
     table[i][j] = (i+1)*(j+1); } }
// deallocation
for (int i=0 ; i<nRows ; i++) {
   delete [] table[i]; }
delete [] table;
```

49

INPUT/OUTPUT FILES

INPUT/OUTPUT FILES

- The fstream library allows us to create, read, and write to files.
- Therefore, load both iostream and fstream header files to be able to create files:

```
#include <iostream>
#include <fstream>
```

CREATE AND WRITE TO A FILE

- Use the keyword ofstream to create a file object
- Use the insertion operator (<<) to write to the file

Example:

```
#include <iostream>
#include <fstream>

// Create and open a text file
ofstream myFile("filename.txt");
// Write to the file
myFile << "Hello world";

// Close the file
myFile.close();
```

READ A FILE

- Use the keyword ifstream to read from a file
- getline() function fetches a line from the file

Example:

```
#include <iostream>
#include <fstream>
// Read from the text file
ifstream myFile("filename.txt");
// Use a while loop along with getline() to read the file line by line
string myText;
while (getline(myFile, myText))
   cout << myText; // Output the text from the file

myFile.close(); // Close the file
```

53

CLASSES

CLASS OVERVIEW

- A class is a **user-defined type**. It has data members, aka, **“attributes”**, just like structures, but the class may also include functions, aka, “**methods**”
- Example:

```
class Passenger {
  public:
    string name; // a public "data member"
    string getName(); /* public "method", also known as a
                         public "member function" */
  private:
    string freqFlyerNumber; // a private "data member"
    string setFreqFlyerNumber(); /* private "method", also
                         known as a private "member function" */
};
```

Public members can be accessed from anywhere, unlike **private members** that can be accessed only from inside the class

- The body of a method is typically defined outside the class. If it consists of one line of code, it can defined inside the class.

CONSTRUCTORS

- An "**object**" is a variable whose data type is a class
- A **constructor** in C++ is a special method that is automatically called when an object/instance of a class is created.
- A constructor is basically a method that has the same name as the class and does not specify a return type (not even void!)
- A constructor can also take parameters (just like regular functions), which are typically used for *setting initial values of the attributes.*

CLASS EXAMPLE

```
class Counter {
   public:
      Counter(int x);  // a constructor (a function has the same name as the class)
      int getCount();
      void increaseBy(int x);
   private:
      int count;
};
// Below, we define the methods of the class “Counter”
Counter::Counter(int x) { count = x; }
int Counter::getCount() { return count; }
void Counter::increaseBy(int x) { count += x; }

int main(){
    Counter c1(5);   // here, the constructor of “Counter” is called; it initializes “c1”
    cout << c1.getCount();  // prints 5
    c1.increaseBy(3);  // calling the method “increaseBy”, which increase “c1.count”.
    cout << c1.count; // ERROR: “count” is private and can’t be accessed outside Counter.
    return EXIT_SUCCESS;
}
```

c1 is an **object** of type Counter

- Notice that the constructor of Counter was automatically called when we defined c1.
- If you want, you can define multiple constructors, each with a different set of arguments

THE "COPY CONSTRUCTOR"

```
class Person {
   public:
      Person( string x, int y );  // A constructor that takes as input an integer and a string
      Person( Person p );  // A copy constructor; it takes as input an object of type Person
      string getName( );
      int getAge();
   private:
      string name;
      int age;
};

// Below, we define the functions that are members of the "Counter"
Person::Person( string x, int y ) { name = x; age = y; }
Person::Person( Person p ) { name = p.name; age = p.age; }
string Person::getName() { return name; }
int Person::getAge() { return age; }


int main (){
     Person p1("Jim", 25);   // here, the constructor of "Person" is called; it initializes "p1"

     Person p2( p1 ); /* here, the copy constructor of "Person" is called; it initializes "p2" such
                        that its members have the same values as those of p1, i.e., it copies p1 */
     return EXIT_SUCCESS;
}
```

POINTERS & CLASSES

You can create a pointer that points to an object of a particular class. As with dynamically-allocated arrays, you can use the keywords “new” and “delete”.

```
int main(){
   int* y;
   y = new int[100]; // remember, this is how we create a dynamically-allocated array
   delete [] y;  // this is how we free the memory

   // similarly, we can create a pointer to an object of a class.
   Counter* z = new Counter(5);  // Now z points to an object of “Counter”

   // Now, we can access the object that “z” points to
   z->getCount(); // remember, “z->getCount()” is the same as “(*z).getCount()”

   // This is how we free the memory that z points to
   delete z;

   return EXIT_SUCCESS;
}
```

Be careful when deleting an object!

MEMORY LEAKS

The keyword "**delete**" clears the memory in which the members of the object are stored, but it does not clear the memory that those members point to!

```
class MyList {
   public:
      MyList(int n);  // A constructor of class X
   private:
      int* list;
};

MyList::MyList( int n ) { list = new int[n]; } /* This constructor creates a
                                  dynamically-allocated array of n integers */


int main(){
     MyList*  x = new MyList(100); /* here, the constructor of "MyList" is called, which
                          allocates memory for x.list to be an array of 100 integers */

     delete x; /* This clears the memory of the object that x points to, but does not
            clear the memory in which "x.list" is stored, leading to a memory leak */
     return EXIT_SUCCESS;
}
```

Solution: Use a "destructor"!

DESTRUCTORS

A destructor of class “MyList” is a function called “~MyList” that does not have a return type (not even “void”). It is **automatically called** when using the keyword **“delete”**.

```
class MyList {
   public:
      MyList(int n);  // A constructor of class X
      ~MyList(); // A destructor of class X
  private:
      int* list;
};

MyList::MyList( int n ) { list = new int[n]; } // Defining the constructor
MyList::~MyList() { delete [] list; } /* Defining the destructor, which clears the memory
                                         of “list” */
int main(){
     MyList*  x = new MyList(100);   // here, the constructor of “MyList” is called
     delete x; // The destructor of “MyList” is called, which clears the memory of x.list
     return EXIT_SUCCESS;
}
```

The destructor is **also automatically called** when an object of type MyList goes **out of scope**, i.e., if:

- The object was defined in a function, and the **function ended**.
- The object was a local variable defined in a block, and the **block ended**.
- The object was defined in a program, and the **program ended**.