OntarioTech Science

Kourosh Davoudi
kourosh@uoit.ca

Week 3: Modular Programming
Pointers
Dynamic Memory Allocation

CSCI 1061: Programming Workshop II

Learning Outcomes

In this week, we learn:

- Modular programming
- Pointers and arrays
- Dynamic Memory Allocation

Modular programming

- Facilitates organizing a large-scale programs
- Header files *.h contains
  - Declaration of classes/structures
  - Prototype of functions
  - External variables
- Implementation files *.cpp contains
  - Definition of methods
  - Definition of functions

Example

```cpp
// main.cpp

#include <iostream>
using namespace std;

#include "main.h"
#include "student.h"

int main(int argc, char const *argv[])
{
    Student myClass[ClassNo];
    for (int i = 0; i < ClassNo; ++i)
    {
        readStudent(myClass[i]);
    }
    printAvg(myClass,ClassNo);
    return 0;
}

void printAvg(Student s[], int size)
{
    int sum = 0;
    for (int i = 0 ; i < size; ++i)
    {
        sum += s[i].grade;
    }
    cout << "The class average is " << static_cast<double>(sum)/size << endl;

}
```

```cpp
// main.h

#ifndef MAIN_H
#define MAIN_H

#include "student.h"

const int ClassNo = 2;
void printAvg(Student [], int );

#endif
```

```cpp
// student.h

#ifndef STUDENT_H
#define STUDENT_H

struct Student{
    char name[100];
    int grade;
};

void display(Student & s);
void readStudent(Student & s);

#endif
```

Example

```cpp
// student.cpp

#include <iostream>
#include "student.h"

using namespace std;

// Implementation of functions

void display(const Student & s)
{
    cout << s.name << endl;
    cout << s.grade << endl;
}

void readStudent(Student & s)
{
    cout << "Enter a name: ";
    cin >> s.name;

    cout << "Enter grade: " ;
    cin >> s.grade;
}
```

```cpp
// student.h

#ifndef STUDENT_H
#define STUDENT_H

struct Student{
    char name[100];
    int grade;
};

void display(Student & s);
void readStudent(Student & s);

#endif
```

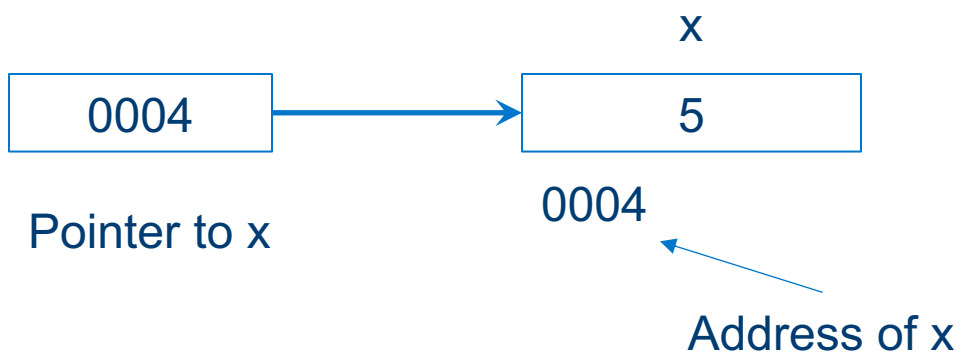
```
g++ student.cpp -c // compilation=> student.o
g++ main.cpp  -c  // compilation=> main.o
g++ main.o student.o -o myprog // link
./myprog
```

Learning Objectives

- Pointers
  - Pointer variables
  - Memory management
- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic

Pointer Introduction

- What is a pointer?
  - A variable holding an address of a variable
- Recall:
  - Each byte in a memory has its own address
  - We can access a variable using its name or address



Pointer Variables

- Pointers are "typed"
  - Can store address of certain variable type
- Example:

```
double *p;   // p is pointer to double
             // p can store an address of a double
int *q;      // q is pointer to int
             // q can store an address of a integer
```

Pointing to and & operator

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- Sets pointer variable p1 to "point to" int variable v1
- Operator &
  - Determines "address of" variable
- Read like:
  - "p1 equals address of v1"
  - "p1 points to v1"

Pointing to …

- Recall:

```
int *p1, *p2, v1, v2;

p1 = &v1; // now we can say that p points to v1
```

- Two ways to refer to v1 now:
  - Variable v1 itself:

```
cout << v1;
```

  - Via pointer p1:

```
cout  << *p1;
*p1 = 7;
```

- Dereference operator *
  - If p1 points to v1 then *p1 IS v1 (You to say ‘is’ and not ‘value of v1’)

Pointers and arrays

- There is close relationship between arrays and pointers in C++
- How?
  - In C++ each array name is the pointer to first element of the array
  - This pointer is constant!

That is all about this relationship

```
int  a[5];
```

- Note that it is different from

```
int *a;
```

There is no storage that a points to !!!

Pointers Arithmetic

Example: int *p, *q;

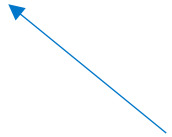
- ++/-- or adding to integer:
  - ++: add pointer to size of object that p points to (e.g. p++ adds p to sizeof(int)
- ==, >= , …
  - Two pointer should have the same type
- p-q
  - Is an integer showing the the number of objects (here, integers) and NOT byte between p and q
- Compare to nullptr/0
  - P==nullptr //by definition if it is correct it shows that p does not poin to //any variable

What is the two main usage of pointers

- Call by reference
  - C++ eliminates this need by introducing the reference concepts
  - DO NOT confuse yourselves by making relationship with reference and pointers!!

They are different !!

- Dynamic Memory Allocation



Very important !

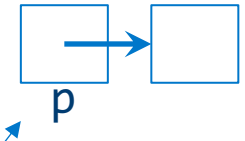
Dynamic Memory Allocation

- What is dynamic memory allocation?
  - A memory which allocated in run time (rather than compile time)

- Why do we need it?
  - It helps us allocate (and free) memory as we need

The new Operator

- Can dynamically allocate variables
  - Operator *new* creates variables and gives the its address




```
int *p;
p = new int;   // new allocates an integer
               // and returns its address
delete p;      // free the memory that p
               //pointes to
```

The new Operator

- Can dynamically allocate variables
  - Operator *new* creates variables and gives the its address


```
int *p;
p = new int[5];     // new allocates array of
                    // integers and returns its
                    // address (address of 1’st
                    // element)
delete [] p;     // free the memory
```

Checking new Success

- Test if null returned by call to *new:*

```
int *p;
p = new int;
if (p == nullptr)   // NULL represents empty pointer
{
        cout << "Error: Insufficient memory.\n";
        exit(1);
}
```

- If new succeeded, program continues

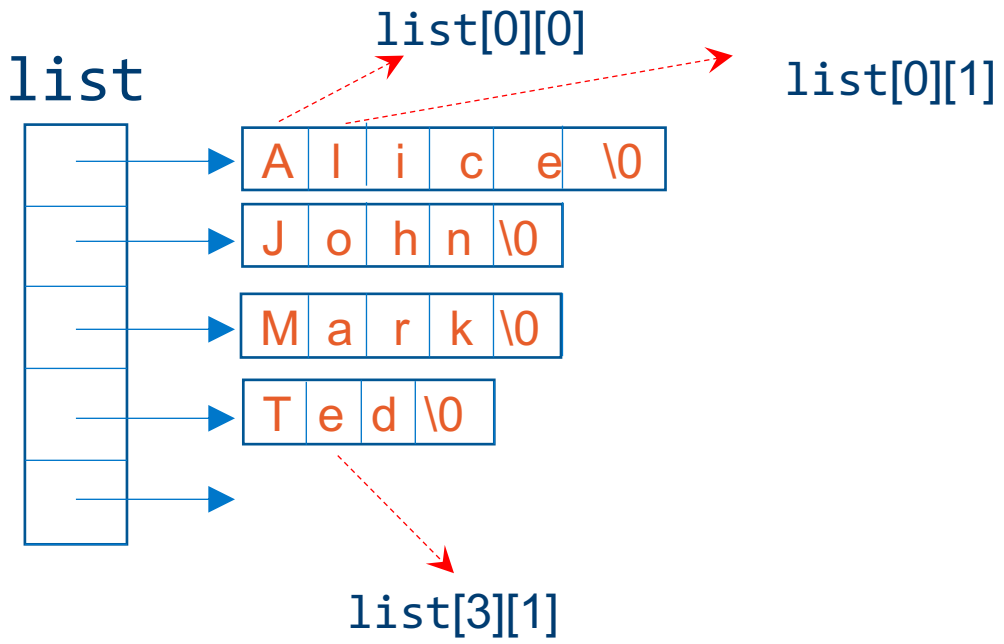
Access to Dynamic Array

```
int *p;
p = new int[5];

*p = 7          // or p[0] = 7
*(p+1) = 8      // or p[1] = 8
```

Array of Pointers

How to create such an structure? (see arr_ptr.cpp)




```
char *list[100];

// list is an array of
// pointers to char
```

Note: List[0] is like a name for the first array

Pointer to Functions

- It is possible to define s pointer to function
  - This allows us to pass a function to another function and ask it to call it

```
int  square(int x); // prototype
int  neg(int x); // prototype

int (*p)(int);     // p is a pointer to a function that
                   // gets an integer and returns an integer
p = square;
cout << p(2);      // call square(2)
p = neg;
Cout << p(3);      // call neg(3)
```