Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

July 6, 2018

Pointers

- In this lecture we shall learn about **pointers.**
- The subjuct of pointers is vast, and we shall study only a few basic topics.
- **Warning:**
  1. Pointers are useful but also tricky things.
  2. **The misuse of pointers is the single largest source of bugs in C++ programs.**
  3. In fact, it was (at least) partly for this reason that references were introduced into C++.

1 Introduction

- ***What is a pointer?***
- A pointer is a variable which **holds the memory address of another variable.**
- For every data type `char`, `int`, `double`, etc., C++ supports the pointer types **pointer to `char`, pointer to `int`, pointer to `double`,** etc.
- Recall the syntax to declare uninitialized variables is as follows.

```
char   c;
int    i;
double d;
double x, y;                    // multiple variables on same line
```

- The syntax to declare uninitialized pointers to `char`, `int` and `double` is as follows

```
char   *c_ptr;                  // pointer to char
int    *i_ptr;                  // pointer to int
double *d_ptr;                  // pointer to double
double *px, *py;                // multiple pointers declared on same line
```

- **The "*" is associated with the pointer.** Hence the notation is "`double *px, *py`, etc.
- There are many rules to learn about pointers, but here is the first rule.
  1. **A pointer can only hold the address of a variable of the same type.**
  2. A pointer to `int` can only hold the memory address of a variable of type `int`.
  3. Similarly, a pointer to `double` can only hold the memory address of a variable of type `double`, etc.
  4. We cannot "mix and match" pointers to different data types.
- **Can we declare a pointer to a pointer?**
  1. *Oh yes!*
  2. And we can declare a pointer to a pointer to a pointer, without limit.

```
double **ppx;                   // pointer to pointer to double
double ***pppx;                 // pointer to pointer to pointer to double
```

- But before we get lost in silliness, let us learn how to declare and initialize pointers.

2 Pointers: declaration and initialization

- **The "&" symbol is the "address of" operator.**
- It returns the memory address of a variable.

```
char   c;
int    i;
double d;

char   *c_ptr = &c;            // points to address of c
int    *i_ptr = &i;            // points to address of i
double *d_ptr = &d;            // points to address of d
double *px    = &d;            // multiple pointers can point to same address
double *py    = &d;            //   " "
```

- **"NULL" is a special symbol for a zero address. We can also use "0" instead.**

```
char   *c_ptr = NULL;          // initialized to NULL
int    *i_ptr = 0;             // initialized to zero, equivalent to NULL
double *d_ptr = NULL;
double *px = 0, *py = 0;       // all initialized to zero on same line
```

- **Assignment of pointers.**
  1. The "*" notation to declare a pointer is a bit confusing.
  2. The names of pointers are really `px`, `py` and `pz`.

     ```
     double d = 1.2345;
     double *px = 0, *py = 0;   // initialized to zero

     px = &d;                   // px points to address of d
     py = px;                   // py points to same address as px (= address of d)
     ```

  3. The "*" is used on the declaration line to tell the compiler that `px` and `py` are pointers.
  4. However, in assignment statements, we write "`px = ...`" etc.
  5. **We can assign one pointer to another.** The statement "`py = px`" is legal.

3 Pointers: dereference

- We know how to declare a pointer and assign a memory address to it.
- **How do we read the data contained at the memory address held by a pointer?**
- The operation of reading the data contained at the memory address held by a pointer is called **dereferencing a pointer.**
- The deferencing operator is again the "∗" symbol *(and yes, in this context it is an operator).*
- The following code fragment demonstrates the dereferencing of pointers.

```
int    i = 4;
double d = 1.2345;
int    *i_ptr = &i;            // declare and initialize
double *d_ptr = &d;            // declare and initialize

*i_ptr = 3;                    // dereference, value of i is now 3
*d_ptr = 7.2;                  // dereference, value of d is now 7.2
```

- The notation "`*i_ptr`" is called **dereferencing the pointer `i_ptr`.**
  1. "Dereference" means **the variable at the memory address pointed to by `i_ptr`.**
  2. The variable is i.
  3. **Therefore the value of i is set to 3.**
- Similarly "`*d_ptr`" means **dereferencing the pointer `d_ptr`.**
- Because `*d_ptr` points to the address of d, therefore the value of d is set to 7.2.
- **Warning: dereference of NULL pointer.**

```
double d = 1.2345;
double *px = &d;               // px points to address of d
double *py = NULL;             // py set to NULL

*px = *py;                     // ERROR  *** dereference of NULL pointer ***
```

- This illustrates the danger of using pointers: *they can be NULL.*
- Attempting to dereference a NULL pointer causes a run-time fault.
- The converse is also an error.

```
double d = 1.2345;
double *px = &d;               // px points to address of d
double *py = NULL;             // py set to NULL

*py = *px;                     // ERROR  *** dereference of NULL pointer ***
```

- The error here is that we attempt to store the value of d at a NULL memory address.

3.1 Example program #1

- The following C++ program illustrates the above ideas.
- The values of i, j, p_1, p_2, $*p_1$ and $*p_2$ should be clear at every step.

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

int main()
{
  int    i = 4;
  int    j = 7;
  int    *p1 = &i;                      // p1 points to address of i
  int    *p2 = 0;                       // initialized to NULL

  p2 = &j;                              // p2 points to address of j

  cout << "i = " << i << endl;          // value is 4
  cout << "j = " << j << endl;          // value is 7
  cout << "*p1 = " << *p1 << endl;      // p1 points to i, output is 4
  cout << "*p2 = " << *p2 << endl;      // p2 points to j, output is 7
  cout << endl;

  p1 = p2;                              // *** BOTH p1 AND p2 POINT TO j ***

  cout << "i = " << i << endl;          // value is 4
  cout << "j = " << j << endl;          // value is 7
  cout << "*p1 = " << *p1 << endl;      // p1 points to j, output is 7
  cout << "*p2 = " << *p2 << endl;      // p2 points to j, output is 7
  cout << endl;

  p1 = &i;                              // p1 points to address of i
  p2 = &j;                              // p2 points to address of j
  j  = 6;
  *p1 = *p2;                            // dereference of p2 copied to dereference of p1
                                        // equivalent to assignment i = j
                                        // both i and j now equal to 6

  cout << "i = " << i << endl;          // value is 6
  cout << "j = " << j << endl;          // value is 6
  cout << "*p1 = " << *p1 << endl;      // p1 points to i, output is 6
  cout << "*p2 = " << *p2 << endl;      // p2 points to j, output is 6

  return 0;
}
```

4 Pointer & arrays

- **There is a close connection in C++ between pointers and arrays.**
- **In C++, an array is allocated as a block of memory and the name of the array is implemented as a pointer to the start of that block of memory.**
- Suppose we have an array a, and for variety suppose it is of type `double`.
- Then a pointer to `double` can be assigned to a.
- The syntax is as follows.

```
int n = 3;
double a[n];
double *p = a;                      // p points to start of array "a"
```

- Equivalently, we can write the following.

```
int n = 3;
double a[n];
double *p = 0;                      // initialize to NULL
// ...
p = a;                              // p points to start of array "a"
```

- **The syntax is "$p = a$" and not $p = \&a$.** This is because "a" itself is an array.
- **To access the array elements, the notation "$p[i]$" is the same as $a[i]$, for $i = 0, 1, \ldots$**
- *What about the dereference $*p$?*
  1. Recall that p points to the start of the array a.
  2. **Therefore p holds the memory address of $a[0]$.**
  3. **Therefore the dereference $*p$ is the array element $a[0]$.**

4.1 Example program #2

- The following C++ program illustrates the above ideas.
- The pointer p points to the start of the array a, therefore `p[i]` is the same variable as `a[i]`.

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

int main()
{
  int n = 3;
  double a[] = {1.1, 2.2, 3.3};       // "a" is array of length 3
  double *p = a;                      // p points to *** start of array "a" ***

  for (int i = 0; i < n; ++i) {
    cout << a[i] << endl;             // printouts of a[i] and p[i] are same
    cout << p[i] << endl;
  }
  cout << endl;

  *p = 1.7;                           // *p same as p[0] same as a[0]
  p[1] = 33.8;                        // p[1] same as a[1]
  a[2] = 7.6;                         // a[2] same as p[2] therefore p[2] = 7.6

  for (int i = 0; i < n; ++i) {
    cout << a[i] << endl;             // printouts of a[i] and p[i] are same
    cout << p[i] << endl;
  }

  return 0;
}
```

5 Pointers in function calls, part 1

- **We can pass pointers as arguments to functions.**
- Consider the following function to swap two numbers of type `double`.

```
void swap_using_ptr(double *u, double *v)
{
  double tmp = *u;                    // save temporary value (dereference)
  *u = *v;                            // copy dereference
  *v = tmp;                           // finish the swap
}
```

- Here u and v are pointers (to `double`, in this example).
- Then u and v **point to memory addresses of variables in the calling application.**
  1. The dereferences of u and v inside the function **operate on the variables in the calling application.**
  2. Hence "$*u = *v$" **is an assignment of the variables in the calling application.**
  3. Next "$*v =$ `tmp`" **is also an assignment of a variable in the calling application.**
- **The above function swaps the values of two variables in the calling application.**
- Note that we could also perform the swap using references.

```
void swap_using_ref(double &a, double &b)
{
  double tmp = a;
  a = b;
  b = tmp;
}
```

- The references a and b are bound to variables in the calling application.
- There are analogies between pointers and references.

6 Pointers in function calls, part 2

- **Next consider the following function to swap two arrays, also to print two arrays.**

```
void swap_array(int n, double *u, double *v)
{
  for (int i=0; i < n; ++i) {
    double tmp = u[i];              // save temporary value (array element)
    u[i] = v[i];                    // copy element
    v[i] = tmp;                     // assign element
  }
}

void print(int n, const double *p1, const double *p2)
{
  for (int i=0; i < n; ++i) {
    cout << p1[i] << "     " << p2[i] << endl;
  }
  cout << endl;
}
```

- Here u and v **hold the memory addresses of arrays in the calling application.**
- Hence $u[i]$ and $v[i]$ **are elements of the arrays in the calling application.**

  1. First "tmp $= u[i]$" **copies the value of $u[i]$ from the array in the calling application.**
  2. Next "$u[i] = v[i]$" **is an assignment of array elements in the calling application.**
  3. Next "$v[i] =$ tmp" **is an assignment to $v[i]$ in the array in the calling application.**

- **The above function swaps the values of two arrays in the calling application.**
- The "`print`" function is relatively obvious.

  1. The input pointers are tagged "`const`" to indicate the print function will not change the values of the elements of (`p1` and `p2`).
  2. This is analogous to the notion of "`const` references" in function arguments.
  3. The `const` keyword is a promise to the compiler that the code will not make changes that would alter the values of variables in the calling application.

- However the inputs to `swap_array` cannot be tagged `const` because the function *will* change the values of the array elements.

7 Pointers as return values of functions

- **The return value of a function can be a pointer.**
- Consider the following function.

```
double* get_element(vector<double> &v, int n)  // return type = pointer
{
    if ((n >= 0) && (n < v.size()))
      return &v[n];                            // return value = pointer
    else
      return NULL;                             // return value = pointer (NULL)
}
```

- **The return value of the function is a pointer.**
  1. If `0 < n < v.size()`, then we return the **address of** $v[n]$**.**
  2. Else we return NULL.
- Here is an example main program to call the above function.

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

double* get_element(vector<double> &v, int n)   // etc

int main()
{
  int len = 5;
  vector<double> v;

  for (int i = 0; i < len; ++i)
    v.push_back( i + 1.2 );

  for (int i = -1; i < len+2; ++i) {
    double *p = get_element(v, i);
    if (p == NULL)
      cout << "null, i = " << i << endl;
    else
      cout << *p << endl;
  }

  return 0;
}
```

8 Dynamic memory allocation

8.1 Introduction

- All the pointers up to now pointed to variables which already exist in the program.
- We now come to one of the most useful features of pointers.
- **We shall use pointers to *create new variables which did not exist before.***
- The process of doing so is called **dynamic memory allocation.**
- Dynamic memory allocation is performed using the operators **new** and **delete.**
- The procedure is slightly different for single variables and arrays.
- We begin with single variables and deal with arrays later.

8.2 Dynamic allocation of single variables

- The dynamic memory allocation and deallocation is illustrated in the following program.

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

int main()
{
  int *ip = new int;             // dynamic memory allocation for single item

  *ip = 6;
  cout << *ip << endl;

  delete ip;                     // memory release for single item

  // SECOND PASS
  ip = new int;                  // allocate again
  *ip = 3;
  cout << *ip << endl;
  delete ip;                     // release again

  return 0;
}
```

- The memory is allocated using the operator **new** as shown above.
- Obviously for a pointer to `double` we would write **new double**, etc.
- *The allocated variable does not have a name.* It is accessed via the dereference "`*ip`" but has no explicit name of its own.
- Dynamically allocated memory is **released**, or **deallocated**, by invoking the operator **delete** as shown above.
- Note that although the memory was deallocated, **the pointer `ip` is still in scope.**
  1. Therefore we can allocate fresh memory by calling `operator new` again.
  2. **The variable allocated on the second pass is not the same as the variable allocated on the first pass.**
  3. The variable allocated on the first pass was deallocated by the first call to `operator delete` and no longer exists.
  4. *We must deallocate the memory on the second pass by calling* `operator delete` *again.*
- **Important:**
  For every call to `operator new`, there must be a partner call to `operator delete`.

8.3 Dynamic allocation of array

- The dynamic memory allocation and deallocation for an array is shown in the program below.

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

int main()
{
  int n = 5;
  int *p_array = new int[n];           // dynamic memory allocation for array

  for (int i = 0; i < n; ++i)
    p_array[i] = i*i;                  // set some values
  for (int i = 0; i < n; ++i)
    cout << p_array[i] << endl;        // print

  delete [] p_array;                   // memory release for array

  return 0;
}
```

- **The operators `new` and `delete` are invoked with a slightly different syntax.**
- For operator `new`, we must obviously specify the length of the array.
- The syntax for operator `delete` is peculiar.
  1. We require a pair of empty brackets `delete []`.
  2. *The length of the array is not specified.*
  3. The run-time system knows how much memory was allocated.
- Obviously, for every call to operator `new` to allocate an array, there must be a partner call to operator `delete []`.
- **It is an error to invoke operator `delete []` (with brackets) for a single variable or `delete` (without brackets) to deallocate an array.**
- *This is terribly inconvenient and causes many bugs.*
- I have no idea why C++ was designed this way.

8.4 Static and dynamic allocation

- For ordinary variables, the compiler allocates the required memory at compile time.
- This is called **static allocation.**
- Statically allocated memory remains in place until a variable goes out of scope, at which time it is deallocated by the compiler.
- The memory for dynamically allocated variables is **allocated at run time.**
  1. That is why it is called **dynamic allocation.**
  2. The compiler does not know how much memory is requested.
- **A very important fact is that dynamically allocated memory is not deallocated until we call operator `delete` or `delete []`, as appropriate.**
- **Another important fact is that the length of a dynamically allocated array is not known at compile time.**
- If a statement such as "`double pd = new double[n];`" is executed multiple times, *the value of n could be different every time.*
- We shall see an example below, using strings.

8.5 Variable length arrays

- Consider the following program.
- The function `stringToArray` receives an input string, and dynamically allocates an array of type `char` to hold the individual characters, and returns a pointer to the dynamically allocated array.

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

char* stringToArray(const string &s)
{
  int n = s.size();
  char *p = new char[n];                 // dynamic memory allocation
  for (int i=0; i < n; ++i) {
    p[i] = s[i];
  }
  return p;                              // return pointer
}

int main()
{
  string s1("abcd");
  string s2("alpha");
  char *p1 = stringToArray(s1);          // memory allocated inside function
  char *p2 = stringToArray(s2);          // memory allocated inside function

  for (int i = 0; i < s1.size(); ++i)
    cout << p1[i] << endl;

  cout << endl;
  for (int i = 0; i < s2.size(); ++i)
    cout << p2[i] << endl;

  delete [] p1;                         // release memory
  delete [] p2;                         // release memory

  return 0;
}
```

- **See next page(s).**

- There are several significant features in the above program.
- First, *the length of the input string is not known inside the function, at compile time.*
  - The value of `n = s.size()` can be different on each function call.
- **Second, a fresh array is allocated on each function call.**
  - It is not the same array, being reused.
- **Third, a fact of very great significance is that the dynamically allocated memory does not go out of scope at the function exit.**
  1. Recall that dynamically allocated memory is deallocated only when a call to operator `delete` is made (actually `delete []` because it is an array).
  2. The pointers `p1` and `p2` in the main program therefore point to valid memory.
- **Fourth (also very significant) is that the memory allocation and deallocation take place in different subprograms.**
  1. The memory is allocated in the function `stringToArray`.
  2. The memory is released in the main program (calling application).
  3. There is no reason why dynamic memory allocation and release must take place in the same scope.
  4. This is different from static allocation.
- An additional issue is that of a **memory leak.**
- Memory leaks will be discussed in a separate subsection.

8.6 Memory leak

- Let us modify the previous main program slightly, as shown below.

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

char* stringToArray(const string &s)
{
  // etc
}

int main()
{
  string s1("abcd");
  string s2("alpha");
  char *p = stringToArray(s1);

  for (int i = 0; i < s1.size(); ++i)
    cout << p[i] << endl;

  p = stringToArray(s2);                   // memory leak occurs

  cout << endl;
  for (int i = 0; i < s2.size(); ++i)
    cout << p[i] << endl;

  delete [] p;                             // only one deallocation
  return 0;
}
```

- The problem occurs when the pointer p is reassigned in the statement `p = stringToArray(s2);`
- Once the point p is reassigned to point to the second array, there is nothing in the program which points to the first dynamically allocated array.
- **The address of the first dynamically allocated array has been lost.**
- **There is no way to deallocate its memory.**
- This situation is called a **memory leak.**
- The resulting inaccessible memory is called **garbage.**
- If the quantity of garbage grows to an unacceptable level, other applications may not be able to obtain enough memory to run correctly.

8.7 Array of pointers

- It is perfectly possible to declare an array of pointers.

```
const int n = 10;
int *array_ptr[n];
```

- Observe that the array length is n, and the data type is **pointer to `int`**.
- It is also perfectly possible to dynamically allocate an array of pointers.
- We require a **pointer to pointer to `int`**.

```
int **pp = new int*[n];
// etc
delete [] pp;
```

- We shall see an example when we dynamically allocate a multi-dimensional array.

8.8 Multi-dimensional arrays, part 1

- Suppose we wish to declare a rectangular array of type `int` and size $m \times n$.
- For an ordinary array, the syntax is as follows.

```
const int m = 2;
const int n = 3;
int a[m][n];
```

- **We can also dynamically allocate a multidimensional array using pointers.**
  1. Note that each row is an array (of length n).
  2. There are m rows, i.e. totally m arrays, each of length n.
  3. **We require a pointer to a pointer to `int`.**
  4. The dynamic memory allocation is performed in two steps.

     ```
     int **pp = new int*[m];          // *** array of pointers (length = m) ***

     for (int i=0; i < m; ++i) {
       pp[i] = new int[n];            // allocate each element (length = n)
     }
     ```

  5. **First we dynamically allocate an array of length m of *pointers to* `int`.**
  6. Then we execute a loop and allocate `pp[i]` to an array of length m, of type `int`.
- We access an array element via `a[i][j]` or `pp[i][j]`.
- Remember to release the memory at the end.

```
for (int i=0; i < m; ++i) {
  delete [] pp[i];              // release memory
}
delete [] pp;                   // release memory
```

- **It is not adequate to simple write "`delete [] pp;`" because that does not release all the memory. There must be one call to operator `delete []` to partner every call to operator `new`.**

8.9 Multi-dimensional arrays, part 2

- A statically allocated array must be rectangular.
- **A dynamically allocated array need not be rectangular.**
  1. Suppose we need to do a calculation using an array, but the only elements required in the calculation are a_{ij} where $j \leq i$.
  2. Declaring a rectangular array wastes memory, especially if the size of the array is large.
  3. **The following code demonstrates how to dynamically allocate a triangular array.**

```
const int m = 4;
int **pp = new int*[m];                // array of pointers

for (int i=0; i < m; ++i) {
  pp[i] = new int[i+1];                // allocate triangular array
}
```

  4. **The arrays `pp[i]` are allowed to have unequal lengths.**
  5. Once again, remember to release the memory at the end.

```
for (int i=0; i < m; ++i) {
  delete [] pp[i];                // release memory
}
delete [] pp;                     // release memory
```

9 Pointer arithmetic

- The concept of "**pointer arithmetic**" exists, but it has some peculiar rules.
- First of all, it does not make sense to add or multiply two pointers, or to multiply a pointer by an integer or double. This will generate a compilation error.
- However, **we can add a pointer and an integer.** The result is a new memory address.
- Let `int *ip` and `double *dp` be pointers to `int` and `double`, respectively.

```
ip + 1  = memory address of ip plus one integer (= 4 bytes)
dp + 1  = memory address of dp plus one double  (= 8 bytes)
```

- Consider arrays `int a[100]` and `double d[100]`. Then we have the following.

```
int a[100];
a + n  = memory address of a[n],   n = 0,1,2,...,99
*(a+n) = same meaning as a[n]

double d[100];
d + n  = memory address of d[n],   n = 0,1,2,...,99
*(d+n) = same meaning as d[n]
```

- **We can subtract pointers of the same type.** The result is an integer.

```
int a[100];
int *ip_m = &a[m];      // address of a[m]
int *ip_n = &a[n];      // address of a[n]

ip_n - ip_m  == n - m  = distance between ip_n and ip_m (divided by sizeof(int))
```

- **We can compare pointers of the same type.** Use the example above.

```
ip_n >  ip_m            // true if n - m > 0
ip_n == ip_m            // true if n - m == 0
ip_n <  ip_m            // true if n - m < 0
```

- Similarly for pointers to `double`.

```
double d[100];
double *dp_m = &d[m];   // address of d[m]
double *dp_n = &d[n];   // address of d[n]

dp_n - dp_m  == n - m  = distance between dp_n and dp_m (divided by sizeof(double))

dp_n >  dp_m            // true if n - m > 0
dp_n == dp_m            // true if n - m == 0
dp_n <  dp_m            // true if n - m < 0
```