Advanced Programming
CSCI251

C++ Foundations: Pointers, classical arrays, and dynamic memory allocation

Outline

Compound types: References

Compound type

A compound type is a one that is defined in terms of another type. Two important compound types in C++ are references and pointers

Reference

A reference defines an alternative name for an object (another variable). We write a declarator &d; d is the name being declared.

```
int ival = 1024;
int &refVal = ival; // refVal refers to ival
int &refVal2; // error; reference must be initialized
```

- When reference is defined, it binds reference to its initializer.
- For example: iVal is NOT copied into refVal
- Once initialized, a reference remain bound to its initializer.
- A reference is not an object; just another name for an existing object

Compound types: References

Caution!

```
int one28B = 1024, two56B = 2048; // both ints
int &rf1 = one28B, rf2 = two56B; //rf1 bound to one28B; rf2 is int
int i3 = 1024, &ri = i3; //i3 is int, r1 is reference bound to i3
int &r3 = i3, &r4 = two56B; // both r3 and r4 are references
```

- A reference may be bound only to an object, not a literal or result of a general expression.

  For example

```
auto &val = 42; // error: cannot bind non-const
                // lvalue reference of type
                // 'int&' to an rvalue of type 'int'
int jelly = 100;
int beans = 24;
int &dentist = jelly + beans; // error: cannot bind
                              // non-const lvalue reference
                              // of type 'int&' to an rvalue
                              // of type 'int'
```

Passing variables to functions

- C++ has 2 ways to pass variables to functions:
  1. Pass by value;the function doesn't need to change the arguments

     ```
     return_type function_Name (type var1, type var2, ...);

     int get_larger (int A , int B);
     ```

  2. Pass by reference; the function may change the arguments.

     ```
     return_type function_Name (type &var1, type &var2, ...);

     int sort (int &A , int &B);
     ```

But we can mix these ...

```
return_type function_Name (type var1, type &var2, ...);

int add_rate(int rate, int &value);
```

Practice 1

Functions: Default Arguments

- Default values; trailing arguments can be omitted.
- For example, a function declaration with default arguments:

```
void DrawString(char Text[],
                int Style = 0, int Size = 12,
                int HSet = 0, int VSet = 0);
```

  Valid calls to this function include:

```
DrawString("Enter your amount");// 4 default values assumed
DrawString("You won", 3, 24); // 2 default values assumed
DrawString("Increase your bid? ", 3); //3 default values assumed
```

- Note that there is an order to the assumed default values

Practice 2

Compound types: Pointers

Pointers

A pointer is a compound type that points to another type. Pointers can be used for indirect access to other objects. We define a pointer by writing a declarator of the form `*d` where `d` is the name being defined.

```
int *ip1, *ip2; // both ip1 and ip2 are pointers to int
double dp, *dp2; //dp2 is a pointer, dp is a double
```

- Unlike references, pointers are objects; can be copied, and assigned
- Pointer need not be initialized at the time it is defined.

Note this!

A pointer holds the address of another object. That address can be retrieved by using the address-of operator, &

```
int iVal = 100;
int *pInt = &iVal; // pInt holds  the address of ival;
                   //pInt is pointer to iVal

double dval;
double *pd = &dval; // ok. pointer pd initialized to
                    // the address of dval
double *pd2 = pd;   // ok. pd is pointer used to initialize pd2
```

Compound types: Pointers

Continued from previous slide ...

```
int *pInt2 = pd; // error: types of pInt2 and pd differ
pInt2 = &dval; // error: assigning the address of double
               // to a pointer to int
```

Compound types: Pointers

Using Pointers to Access an Object

When a pointer points to an object we can use the dereference operator (*) at access the object.

```
int iVal = 78;
int *intPtr = &iVal; // intPtr holds the address of iVal,
                     // intPtr is a pointer to iVal
std::cout << *intPtr; // * yields the object
                      // to which intPtr points
                      // and cout prints 78
*intPtr = 100;        // * yields the object;
                      // allows us to change its value
std::cout << *intPtr; // prints 100
```

You may only dereference a valid pointer that points to an object.

```
double *dPtr; // does not point to any object
std::cout << *dPtr; // error: dereferencing a pointer
                    // that does not point to any object.
```

Compound types: Pointers

How to read codes with & and *

Read from RIGHT to LEFT

```
int i = 42;   // value 42 is used to
              // initialize variable i of type int
int &r = i;   // value of i initializes reference variable r
int *p;       // p is pointer to int
p = &i;       // address of i is used to initialize pointer p
*p = i;       // value of i is assigned to what p points to
```

Compound types: Pointers

Null pointer: nullptr

- A null pointer does not point to any object.
- Modern C++ introduced the literal nullptr and it should be used when initializing a pointer to null.
- Older code may use NULL; do not use in new code.
- It is illegal to assign an int variable to a pointer

```
int krill = 0;
int *ptr = krill; // error: cannot assign an int to a pointer
int *ptr = nullptr; // correct method of null initialization
int *ptr = 0; // correct, direct initialization to literal o
```

Practice 3

Arrays

Arrays - another compound type

Arrays are collections of variables of the same type, and of fixed size, that we access by position.

- With fixed size the operations on elements are highly optimized.
- Dynamic arrays are possible when unsure of size; memory/space allocated dynamically.
- A `std::vector` might be better in this case

Setting up arrays:

```
type  array_name[dimension];
double  house_prices[50];
```

- dimension → size.
- The array name represents a memory address

Arrays
Example

```
const int limit = 1;
int someValue[limit]; // Array someValue has 1 element
int m;
cout << "Enter the size of the container: ";
cin >> m;
int someArray[m]; // Array someArray has m elements
cout<<sizeof(someArray);
```

Code outputs the value m given by user, multiplied by the bytes representing an int; $(4 \times m)$

- What is this keyword const?
- We use it when defining a variable that must not change. For example:

```
const double myPi = 3.142;
const int numOfSides = 165;
const int numPlates;   // error: uninitialized
                       // const 'numPlates'
numOfSides = 45; // error: assignment
                 // of read-only variable 'numPlates'
```

- Useful when declaring/defining an array.

Arrays

Initializing an array

- Declare an array:

```
const int postCodeLength = 4;
int postCode[postCodeLength];
```

- Initialize as ..

```
int postCode[4]={0}; // All elements initialized to 0
```

- Access the elements as (0 -indexed)...

```
postCode[2]= postCode[1] + 25;
```

Arrays

More on initialization

- There are a few different ways to initialise.
- For an array of three ints with values 0, 1, and 2.

```
const unsigned sz =3;
int ia1[sz] = {0, 1, 2};
int a2[] = {0, 1, 2}; // size inferred from the initialiser
int a3[5] ={0, 1, 2}; // need not have all the initial values
std::string a4[3] = {"hi", "bye"}; // 2 initialisers
int a5[2] = {0, 1, 2}; // too many initializers for 'int [2]'|...
```

- The unitialized parts are value-initialized,
  - `int` to 0
  - `std::string` to empty string

Character Arrays

Character arrays are special

Character arrays can be initialised using a string literal, and they end with a null character \0

- These are referred to as C-strings.
- This can be explicit in element by element declarations

```
char a1[] = {'C', '+', '+'};
char a2[] = {'C', '+', '+', '\0'};
char a3[] = "C++";
const char a4[6]= "123456"; // error: assignment of read-only
                            // location; no space for null char \o
```

Arrays

Arrays and Pointer are friends

- Remember we indicated that array name represents a memory address
- When passed as argument to function, arrays are, by default, passed by pointer.
- Therefore arrays passed to functions can be changed by the function unless the keyword `const` is used.

Arrays

Example of passing array to functions

```
void AddArray ( int Size,          // size of the arrays
                const int A[ ],    // array passed as input
                const int B[ ],    // array passed as input
                int C[ ] )         // array passed for output
{
        for (int i=0; i<Size; i++)
        C[i] = A[i] + B[i];
}


int main(){

    const int ArySize = 5;
    int Ary1[ArySize]={1,2,3,4,5};
    int Ary2[ArySize]={6,7,8,9,10};
    int Ary3[ArySize];
    AddArray(ArySize, Ary1, Ary2, Ary3); // Ary3 will contain
                                          // {7, 9, 11, 13, 15}
    return 0;
}
```

Arrays

Example of multi-dimension array passed to function

```
void print3DMatrix (const float A[][3][3]);

int main() {
     float Matrix[3][3][3] ={{{1,2,3},{4,5,6},{7,8,9}},
                              {{1,1,1},{2,2,2},{3,3,3}},
                              {{4,4,4},{5,5,5},{6,6,6}}};

    print3DMatrix(Matrix);
...
    return 0;
}

void print3DMatrix (const float A[][3][3]) {
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
            for(int k=0;k<3;k++)
                std::cout << i  << j  << k << " = "
                << A[i][j][k] << std::endl;
}
```

Arrays

Consider the function:

```
int SumArray(int arr[], int n){
    int i, sum=0;
    for (i=0;i<n;i++)
        sum += arr[i];
    return sum;
}
```

Let an array be declared as:

```
int A[10] = {1,2,3,4,5,6,7,8,9,10};
```

- We can pass the array to the function as

  SumArray(A,10);

  or

  SumArray(&A[0],10);

- The array name A and &A[0] are indicating the address of the array

  If we declare

  ```
  int *B = A;
  ```

  we can pass B to function because B has the address of array A.

Some pointer arithmetic

- When a C++ program references array elements, the compiler has to do some pointer arithmetic.
- For example, A[1] refers to the memory location one after the address A.
- In pointer arithmetic this is *(A+1).
- What does one mean here? One memory stride?
- Depends on type of array A.
- Operator sizeof provides a clue

sizeof a pointer

- What do you get if you apply sizeof to a pointer?

```
double value;
cout << sizeof(value) << sizeof(double)
     << sizeof(double*);
```

Output will be 8 8 8 on a 64 bit computer

Practice 4

Void pointer - void*

- A void pointer
  - Can point to any type
  - But does not know what type it points to We cannot access content through the `void` pointer, unless it is cast to a given type
  - Useful when we want to deal with memory as memory, without accessing the content
  - `sizeof(void*)` is 8 on 64 bit computer.

Void pointer - void*

Consider the following code snippet:

```
int i = 5;
int *ip;
void *vp;
ip = &i;
vp = ip;
cout << *vp << endl;
cout << *((int*)vp )<< endl;
```

C++ cannot print the void but can print the int.

Pointer vs Arrays

Quick summary

- Pointers store address of (something of some type),
- Arrays store collection of elements of some type.
- Can set up a pointer to an object of unspecified type (void),
- Arrays cannot be of objects of unspecified type.
- When you generate an array, the array name will be the pointer to the first element.

Dynamic memory allocation & memory leaking

- In C++ you may want to dynamically manipulate memory.
- It is tricky and nuanced but can improve performance.
- `new` and `delete` are the keywords associated with memory allocation and release.

Program memory layout

- Dynamic memory allocation is carried out by using a special type of operator that directly communicate with the Memory Manager.
- A programmer has to specify how much memory is required.
- The memory manager will find a location currently available.

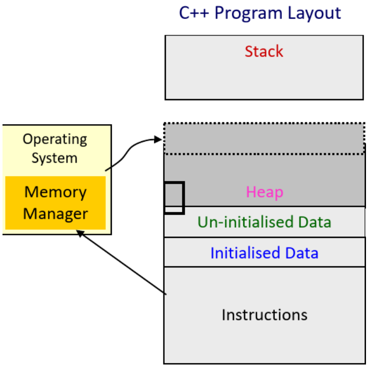

Figure 1: Memory layout in C++

Dynamic memory

Advantages of using dynamic memory (the heap)

- We may not know how many elements we need.
- The precise type we need may be unknown.
- Sharing data/state.
  - Some data is associated with or needs to be known by part of our program, but is owned somewhere else.
  - Improved storage efficiency.

Program memory layout

Mechanics - using new and delete

- Set up a pointer

  ```
  int *intPtr;
  ```

- Dynamically allocate memory with new.

  ```
  intPtr = new int;
  ```

- Operator new returns a pointer to the given type
- Allocated memory using new MUST be released using the operator delete

  ```
  delete intPtr;
  ```

- If you do not follow this procedure you get a memory leak.

Note these points

- Variables can be default initialised, with the default value type and sometimes location dependent.
- Built-in types defined outside function bodies are initialised to zero.

```
...
int *sales = new int; // initialised to zero

double func_double();

int main(){
 double retvalue = func_double();
 ...
 return 0;
}
double func_double(){
    ....
    double returnValue = 0.0;

    return returnvalue;
}
```

Note these points

- Variable can be initialised while being declared

  ```
  int *parks = new int(5); // *parks will contain value 5
  ```

- The type specifier auto can come in useful:

  ```
  auto *p1 = new auto(10);      //compiler deduces type int
  auto *p2 = new auto(5.6); //compiler deduces type double
  ```

Creating a dynamic array

Using new[]

- To create a dynamic array we can use the new[] operator.

```
int *intVar;
intVar = new int[100]; // dynamic array
for(int i = 0; i < 100; ++i)
    intVar[i] = 25-i; // initialize the array

delete [] intVar; // frees the allocated array
```

caution about delete[]

- Extra careful when dealing with pointer to an array of pointers:

```
Person **p = new Person* [2];
p[0] = new Person("Peter");
p[1] = new Person("Alex");
```

- Using delete[] p; just causes the p pointer to be released, not the actual objects themselves.
- You must step through the different index values and use delete p[index] on each.

Using delete[] on pointer to pointers

```
float **fVar;
fVar = new float* [10]; // allocate pointer
                        // array, 10 float pointers
for(int i = 0; i < 10; ++i)
    fVar[i] = new float[10]; // allocate memory to each
. . . .
. . . .
for(int i = 0; i < 10; ++i) // release each ``pointed to''
    delete [] fVar[i];
delete [] fVar; // release the pointer
```

Three common problem with dynamic memory management

1. Forgetting to delete memory.
2. Using an object after it was deleted.
3. Deleting the same memory twice.

Memory leak - example

```
double* calc(int res_size, int max){
    double* p = new double[max];
    double* res = new double[res_size];
// use p to calculate results to be put in res
 ...
    return res;
}

// use res outside the function in main for example
double* r = calc(100,1000);
```

Each call of `calc()` “leaks” the doubles allocated for p.

For example, the call `calc(100,1000)` will render the space needed for 1000 doubles unusable for the rest of the program.

Memory leak fixed

```
double* calc(int res_size, int max){
// the caller is responsible for the memory allocated for res
    double* p = new double[max];
    double* res = new double[res_size];
    // use p to calculate results to be put in res
     ...

    delete[] p; // we don't need that memory anymore: free it

    return res;
}

double* r = calc(100,1000);
// use r outside the function in main for example
delete[] r; // we don't need that memory anymore: free it
```

Segmentation fault

Segmentation fault

- Segmentation faults occur when you try to use memory which does not belong to you, typically:
    1. Getting or setting value from an illegal address
    2. Out of bounds array references
    3. Reference through un-initialized or dangling pointers
- Not allowed to access the memory specified.