Computational Intelligence on Automation Lab @ NCTU

UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 02: *Pointers, References, and Dynamic Memory Allocation*

Outlines

You should be able to review/understand:

- Fundamentals of C++ pointer
  - –difference between C and C++ pointers
- References
  - –as a reference variables
  - –pass to functions
  - –return by functions
- Using references and pointer with constants
- Dynamic memory allocation
  - –dynamic arrays
  - –pass/return array to/from function

Introduction

- C pointers are very powerful
  - –but prone to error if not properly used
  - –including system crashes
- C++ enhances C pointers and provides increased security because of its rigidity
  - –by providing a new kind of pointer *reference*
- References have advantages over regular pointers when *passed to functions*

C/C++ Pointer

- A *pointer* is a variable that is used to store a memory address
  - –can be a location of *variable*, *pointer*, *function*
- Major benefits of using pointers in C/C++:
  - –support *dynamic memory allocation*
  - –provide the means by which functions can modify their *actual arguments*
  - –support some types of *data structures* such as linked lists and binary trees.
  - –improve the *efficiency* of some programs

Pointer

- A pointer variable is declared using

```
⟨datatype⟩ *⟨variable_name⟩;
⟨datatype⟩ * ⟨variable_name⟩;
⟨datatype⟩* ⟨variable_name⟩;
```

- Data type that the pointer points can be any valid C/C++ type including *void* types and *user-defined* types

```
int *ptr1;
double *ptr2;
void *ptr3; //can point to a variable
            //of any datatype
AirTicket *ptr4; //user-defined datatype
```


Operators for Pointers

- *Indirection* operator (*) precedes a pointer and returns the *value* of a variable
  - Address of the variable is stored in the pointer
  - *dereferencing* : access the value that the pointer points to
- *Address-of* operator (&) returns the memory *address* of its operand

```
float x = 1.23, y;
float *pt; //point to any float variable
pt = &x;   //place x’s address into pt
cout << *pt; //print x’s value 1.23
```

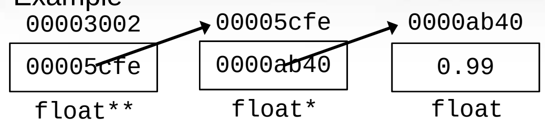

Pointer Expressions

- Pointers can be used as *operands* in assignment, arithmetic (only + and -), and comparison expressions
- Example

```
float f=13.3, *p1, *p2;
p2 = p1 = &f; //p1 and p2 point to f
p1--; //decrementing ptrl
cout << pl; //pl is 992=1000-8
p2 += 5; //add 5 and assign to p2
cout << ptr2; //p2 is 1040=1000+5*8
if (p1==p2){ //compare two addresses by ==
    cout << “Two addresses are the same”
         << endl;
}
```


Point to Another Pointer

- When declaring a pointer that points to another pointer, two *asterisks* (**) must precede the pointer name
- Example




```
float a=0.99, *b, **c;
b = &a; //pointer b points to variable
c = &b; //pointer c points to pointer
cout << **c; //dereferencing pointer c
             //two times to access var a
```

Access Array by Pointer

- An array name
  - –the *starting address* of the array
  - –the address of the first element of the array
  - –can also be used as a *pointer to the array* ⇨ *faster* than index
- Example: `int array[3] = {1, 2, 3};`

array indexing	pointer notation
`array[0]`	`*array`
`array[1]`	`*(array + 1)`
`array[2]`	`*(array + 2)`

Differences between C and C++

- C++ is much *more strict* than C
  - –when dealing with *pointers*
  - –especially apparent for *void pointers*
- A void pointer is a raw address
  - –Compiler has no idea what type (can be *any type*) of object being pointed to
- Format: `void *⟨vaiable_name⟩;`

```
void *p1;
int *p2, x = 3;
p1 = p2 = &x; //p1 and p2 both point to x
//ERROR! p1 can’t be directly dereferenced.
int y = *p1; //int y = *(int *)p1;
```


Function Pointers

- A function block also occupies memory space
  - –placed at Code Section
  - –has its own address
- *function pointer* ⇨ a pointer to the function
  - –Ex: `int (*fp)(char a, int b);`
  - –() has higher precedence than *
- Ex:

```
int fc1(char x, int y);
int (*fp2)(char a, int b);
int fn3(double x);
fp2 = fn1; //OK!! Same type
fp2 = fn3; //Error
fp2=fc1(‘x’, 5); //Error!
```


void Pointers

```
// fun.cpp
void fun1(void (*fp)(void *), void *q) {
    fp(q);
}
void fun2(void *r) {
    int a = * (int *) r;
    cout << a << endl;
}
```

```
//main() block in main.cpp
    int var = 22;
    void *p = &var;
    fun1(fun2, p);
```

- `fp` is a void pointer that points to a function

References

- C++ provides a new kind of variables called *references*
  - a reference is an *implicit* pointer that is *automatically dereferenced*
  - need not use `*` to get values
  - also act as *alternative names* for variables
- A reference can be used in three ways:
  - created as an *independent variable*
  - *passed to* a function
  - *returned by* a function
- Passing references between functions is a powerful, important use

References As Independent Variables

- Put & before the variable when declared
  - The type of reference variable should be the *same* as the type of the variable it refers
  - *reference variables have to be initialized when declared*
- Format: `⟨datatype⟩ & ⟨ref_var⟩ = ⟨old_var⟩;`
- Example:

```
double num = 6.75;
...
double & refnum = num;
```

  - `refnum` is initialized to `num` as its *alias*

Passing References To Function

- C++ supports the three methods for passing values to functions:
  - pass by *value*
  - pass by *address*
  - pass by *reference*
- Passing a reference to a function is like passing the address of the variable to such function, but with advantages:
  - code is *cleaner*
  - *no copy* of function arguments
  - not remember to *pass the address*

Example of Passing References

```
//swap.cpp
void swap(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}
```

```
//main() in main.cpp
int main() {
    int a(4), b(11); //a=4 and b=11
    cout << a << " " << b << endl;
    swap(a, b);
    cout << a << " " << b << endl;
    return 0;
}
```

Constant References

- Reference parameters preceded with `const`
  - can prevent a function from changing them inadvertently
- Example:

```
void fun(const int &cref)
{
    cout << cref/15; //no problem!
    cref++; //error! cannot modify it
}
```


Returning Reference by Function

- A function may return a reference
  - particularly important when overloading some types of operators
  - Ex: *inserter* and *extractor* (in later lecture)
  - also permits the function to be called from the *left side* of the assignment operator

```
int & fun(int * a, int i) {
    if (i>0 && i<5) return a[i];
    else exit(0);
}
```

```
//iary: an integer array
    for (int idx=0; idx<5; idx++)
        fun(iary, idx) = idx*2;
```


References/Pointers with Constants

- If the `const` modifier is used onto references and pointers, one of the following four types can be created
  - a reference to a constant
  - a pointer to a constant
  - a constant pointer
  - a constant pointer to a constant

A Reference to a Constant

- A *read-only* alias
  - cannot be used to change the value it references
- However, a variable that is referenced by this reference can be changed
- Example:

```
int x = 8;
const int & xref=x; //a ref to a const.
x = 33;
cout << xref ;
xref = 15; //ERROR! cannot modify xref
x = 50; //OK
```

A Pointer to a Constant

```
int x = 4, y = 7;
const int *pt = &x; //a pointer to a const.
cout << *pt;        //print 4 on screen
pt = &y;
cout << *pt;        //print 7 on screen
*pt = 11;           //ERROR! cannot modify
```

- The pointer `pt` to a constant used in this example can store different addresses
  - –can point to different variables, x or y
- However, cannot change the dereferenced value that `pt` points to

A Constant Pointer

- A constant pointer is a kind of pointers that its content is constant and cannot be changed
  - –cannot be changed to point to another variable
  - –but can change the value it points to
- Example

```
int var1 = 3, var2 = 5;
//a constant pointer to a declared variable
int * const cpt = &varl;
*cpt = 8; //change the value cpt points to
cout << var1; //print 8 on screen
//ERROR! a const. pointer cannot be changed
cpt = &var2;
```


A Constant Pointer to a Constant

- A constant pointer to a constant
  - –cannot be used to change the constant value to which it points.
  - –can be changed to point to another constant of the same type
- Example:

```
const int vl = 11, v2 = 22;
//a constant pointer to a constant
const int *cptc = &v1; //v1 is constant
*cptc = 33; //ERROR! cannot modify
cout << *cptc; //print 11 on screen
cptc = &v2;
cout << *cptc; //print 22 on screen
```


Memory Allocation

- *Static* memory allocation
  - –uses the explicit variable and fixed-size array declarations to allocate memory
  - –reserves an amount of memory allocated when a program is loaded into the memory
  - –a program could fail when lacking enough memory
  - –or reserve an excessive amount of memory so that other programs may not run
- What if the size can be known until the program is running?

  ⇨ *dynamic* memory allocation

Dynamic Memory Allocation

- Only allocate the amount of memory needed at run-time
- Heap (a.k.a. *freestore*)
  - reserved for dynamically-allocated variables
  - all new dynamic variables consume memory in freestore
  - if too many ⇨ could use all freestore memory
- C: `malloc()`, `calloc()`, `realloc()`, `free()`
- C++: `new` and `delete`

`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!
- Example: `int *p1 = new int;`
  - creates a new *nameless* variable, and assigns `p1` to *point to* it
  - can be dereferenced with `*p1`
  - use just like ordinary variable

`delete` Operator

- De-allocate dynamic memory
  - when no longer needed
  - return memory to heap/freestore
- Example:

```
int* p = new int(5); //allocate an int
... //some processing
delete p; //delete space that p points to
```

  - *de-allocate* dynamic memory pointed to by pointer p
  - literally *destroys* memory space

`new`/`delete` Example

```
// declare a pointer and allocate space
double *dpt = new double(0.0);
if (dpt == NULL)
{
    //no enough memory to be allocated
    cout << "Insufficient memory.\n";
    exit(1);
}
*dpt = 3.4; //use pointer to access value
cout << *dpt << endl;
//return the space to heap
delete dpt;
```

- if `new` succeeds, program continues; if not, exit the program ⇨ good to use NULL check

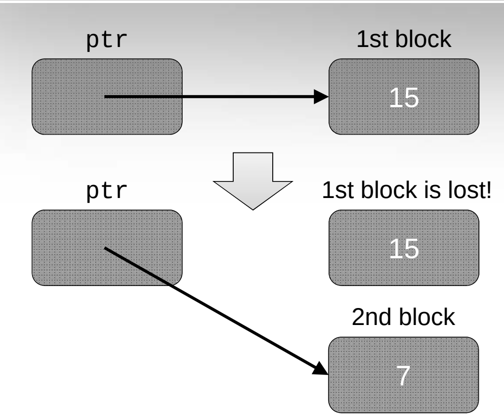
Memory Leaking

- A very common type of error
  - may result in memory resource problem
- A memory leak occurs when a pointer points to *another* block of memory without the *delete* statement that frees the *previous* block
- Example:

```
int *ptr = new int; //allocate 1st block
*ptr = 15; //access 1st block
//not delete the space of 1st block
ptr = new int; //allocate 2nd block
*ptr = 7; //access 2nd block
```

  - no way to access the space of 1st block

illustration of Memory Leaking



Standard vs. Dynamic Arrays

- Standard array limitations
  - must specify size first ⇨ *estimate* maximum
  - may not know until program runs!
  - *waste* memory
- Example:

```
const int MAX_SIZE = 100000;
int iArray[MAX_SIZE];
```

  - what if the user only need 100 integers?
- Dynamic arrays
  - can grow and shrink as needed

Create Dynamic Arrays

- Use **new** operator
  - dynamically allocate with pointer variable
  - treat like standard arrays
- Example:

```
int iSize = 0;
cin >> iSize;
typedef double* DoublePtr;
DoublePtr d;
d = new double[iSize]; //size in brackets
```

  - create a dynamical array variable `d`
  - contain `iSize` elements of type `double`

Delete Dynamic Arrays

- Allocated dynamically at run-time
  – so should be destroyed at run-time
- Continue the previous example:
  de-allocate memory for a dynamic array

```
...
d = new double[iSize]; //size in brackets
...
delete [] d; //delete array that p points
```

  – brackets `[]` indicate *array* is there
  – note that `d` still points there ⇨ dangling!
  ⇨ should add "`d = NULL;`" immediately

Dynamic Multi-dimensional Arrays

- Multi-dimensional arrays are *arrays of arrays*
  – various ways to create dynamic multi-dimensional arrays
- Example:
  declare one array `m` of 3 `IntArrayPtr` pointers

```
typedef int* IntArrayPtr;
IntArrayPtr* m = new IntArrayPtr[3];
for (int idx = 0; idx < 3; idx++)
    m[idx] = new int[4];
```

  – make each allocated array of 4 integers
  – create one 3×4 dynamic array

Two-dimensional Dynamic Arrays

- Example:

```
int *Mat1[4]; //fix row number at 4
for (int r=0; r<4; r++)
    Mat1[r]=new int[6]; //create 6 columns
```

  – 4 rows `Mat1[0]`, `Mat1[1]`, `Mat1[2]` and `Mat1[3]` are declared
  – each row has 6 columns to be created
- Example: (*most common*)

```
int **Mat2; //2-level pointer
Mat2=new int *[4]; //create 4 rows
for (int r=0; r<4; r++)
    Mat2[r]=new int [6]; //create 6 columns
```

  – both `Mat2` and `*Mat2` are pointers

Shallow vs. Deep Copies

- Shallow copy (copy-by-address)
  – two or more pointers point to the same memory address
- Deep copy (copy-by-value)
  – two or more pointers have their own data
- Example:

```
int *first, *second;
first = new int[10];
second = first; //shallow copy
second = new int[10];
for (int idx=0;idx<10;idx++) //deep copy
    second[idx] = first[idx];
```

Delete Dynamic Arrays

- After a dynamic array is of no use any more, deallocate the memory by delete operation
  - –Clean *reversely* from last allocated memory
- Example: //reallocate a dynamic 5x9 matrix

```
int** Mat = new int *[5]; //create 5 rows
for (int r=0; r<9; r++)
    Mat[r] = new int [9]; //create 9 columns
... //some processing
for (int r=0; r<9; r++) //clean columns
    delete [] Mat[r];
delete [] Mat; //clean rows
Mat = NULL;
```


Expand Dynamic Arrays

- A program can start with a small array and then expands it only if necessary
- Example: initially `MAX` is set as `10`

```
int n = 0;
int * ivec = new int [MAX];
while (cin>>ivec[n]) {
    n++;
    if (n>=MAX) {
        MAX *= 2;
        int * tmp = new int [MAX];
        for (int j=0; j<n; j++)
            tmp[j] = ivec[j];
        delete [] ivec;
        ivec = tmp;
}}
```


Pass Arrays to Function

- When array is passed to a function, only pass the *address* of the first element
- Example: in main function

```
int max = FindMax(array, size);
```

in function declaration section

```
int FindMax(int *array, int size) {
    ...
}
```

  - –parameter receives the address of array
    `array` ⇨ `val` is one pointer
  - –Another form:

```
int FindMax(int val[], int size) {}
```


Return Array from Function (1/2)

- Array type pointers are not allowed as return-type of function
- Example:

```
int [] someFun(...); //illegal
```

- Instead return pointer to array base type:

```
int * someFun(...); //legal
```

- Return a *integer pointer* after function call
  - –in main (or caller) function,

```
int * pt = someFun(...);
```

  - –only **one** array (address) can be returned!

Return Array from Function (2/2)

- One more example:

```
int *display();
...
int main() {
    cout << *display() << endl;
}
int *display() {
    int *pt = new int[2];
    pt[0] = 0; pt[1] = 0;
    int b[2] = {10,20};
    for (int i=0; i<2; i++)
        *pt = b[i];
    return pt;
}
```


Allocate C-style Strings

- To store an array of C-style strings
  - –first declare an array of C-style strings
  - –dynamically allocate space for C-style strings
- Example:

```
char wd[100]; //one word
char *wv[50]; //can save 50 words
while (cin>>wd) {
    int len = strlen(wd)+1;
    char *nw = new char [len];
    strcpy(nw, wd);
    wv[n] = nw; //wv[0], wv[1], ...
    n++;
}
```


Summary (1/2)

You have reviewed/learned:

- Fundamentals of C++ pointer
  - –operators and expressions for pointers
  - –point to another pointer/array
  - –void pointers
  - –difference between C and C++ pointers
- References
  - –as a reference variables
  - –pass to functions including constant references
  - –return by functions

Summary (2/2)

- Using references and pointer with constants
  - –a reference to a constant
  - –a pointer to a constant
  - –a constant pointer
  - –a constant pointer to a constant
- Dynamic memory allocation
  - –C/C++ memory allocation
  - –new/delete operators
  - –memory leaking
  - –multi-dimensional dynamic arrays
  - –pass/return array to/from function