IM 1003: Computer Programming
Pointers

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **The concept of pointers**
- Syntax
- Call by reference/pointer
- Pointers and arrays
- Dynamic memory allocation (DMA)

Pointers

- Pointers form one of the most difficult, dangerous, but powerful part in C and C++.
- At the beginning, they do not look useful.
- They will be required for some advanced techniques in C and C++.
- Even for beginners like us, we need them.

Pointers

- A **pointer** is a variable which stores a **memory address**.
  - In a 32-bit computer, a pointer is allocated 4 bytes.
  - In a 64-bit computer, a pointer is allocated 8 bytes.
  - Example "**07_01_pointerSize**".
- When we declare a variable, the compiler will allocate one space in memory for it. It will be accessed through a **memory address**.
- The compiler will also build a table to record each pair of variable name and it address.
- Then the system can also access these variables by their addresses.

Variables in memory

- `int a = 5;`
  `double b = 10.5;`

address	variable name	variable value
…		
0x22ff78	**a**	5
…		
0x22aa70	**b**	10.5
0x22aa74		
…		

Indicating an address

- A pointer variable stores one memory address. In other words, it **points** to a place in memory.
- The address it stores marks the **beginning** of one space.
- `int a = 5;` // give me a pointer to **a**

address	variable name	variable value
0x22ff78	**a**	5
…		
0x21aa74	pointer	**0x22ff78**
0x21aa78		

Indicating an address

- To allow the program to know how large a space does it point to, we should also declare its **type**: The type of the variable it is pointing to.
- `double b = 10.5;` // give me a pointer to a **double** (8 bytes)

address	variable name	variable value
0x22aa78	**b**	10.5
0x22aa7c		
…		
0x21aa74	pointer	**0x22aa78**
0x21aa78		

Outline

- The concept of pointers
- **Syntax**
- Call by reference/pointer
- Pointers and arrays
- Dynamic memory allocation (DMA)

Pointer declaration

- To declare a pointer:
  - *type pointed** *pointer name*;
  - *type pointed* **pointer name*;
- Example:
  - `int *ptrInt;`
  - `ptrInt` does not point to anywhere but will point to an integer variable.
  - `double* ptrDou;`
  - `ptrDou` does not point to anywhere but will point to a double variable.
- You can point to **any** type.
  - Integer pointers, double pointers, etc.
  - The type indicates the size of the variable that is pointed.

Pointer assignment

- We use the **address-of operator** to obtain a variable's address:

pointer name = &*variable name*

- The address-of operator `&`:
  - It is unary. Its associativity is right-to-left.
  - It returns the **address** of a variable.
- Example:
  - `int a = 5;`
    `int* ptr = &a;` // `ptr` points to `a`
- When assigning an address, the two types must **match**.

Pointer assignment

- `int a = 5;`
  `int* ptrA = &a;` // declaration + assignment

address	variable name	variable value
…		
0x22ff78	`a`	5
…		
0x21aa74	`ptrA`	0x22ff78
0x21aa78		

  - Example "**07_02_pointerValue**".

Address operators

- There are two address operators.
  - `&`: The **address-of operator**. It returns the variable's address.
  - `*`: The **dereference operator**. It returns the pointed variable (not the value!).
- For `int a = 5`:
  - `a` equals 5.
  - `&a` returns an address (e.g., 0x22ff78).
- For `int* ptrA = &a`:
  - `ptrA` stores an address (e.g., 0x22ff78).
  - `&ptrA` returns the pointer's address (e.g., 0x21aa74). This has nothing to do with the pointer's value!
  - `*ptrA` returns `a`, the variable pointed by the pointer.

Address operators: &

- **&**: returns the variable's address.
- You can not use **&100**, **&(a++)** (which returns the value of a), since you can only perform **&** on a **variable**.
- You can not assign value to **&x**, since **&x** is a value.
- You can get a usual variable's address. You can get a pointer variable's address, too.

Address operators: *

- *****: returns the pointed variable, **not** its value.
- You can perform ***** on a pointer variable.
- You cannot perform ***** on a usual variable.
- You cannot change a variable's address. No operation can do this.

Some examples

```
int a = 10;
int* ptr = NULL; // to be explained later
ptr = &a; // the value ptr stores (which should be an
          // address) becomes the value returned by
          // & (the address of variable a)
*ptr = 5; // the variable ptr points to (which is a)
          // becomes 5
```

```
int a = 10;
int* ptr = NULL;
ptr = &a;
cout << *ptr; // 10
*ptr = 5;     // the variable ptr points to (which
              // is a) becomes 5
cout << a;    // 5
```


Some examples

```
int a = 10;
int* ptr = NULL;
ptr = &a;
cout << *ptr; // 10
a = 5;        // the variable a becomes 5
cout << *ptr; // 5
```

```
int a = 10;
int* ptr1 = NULL;
int* ptr2 = NULL;
ptr1 = ptr2 = &a;
cout << *ptr1; // 10
*ptr2 = 5;     // the variable a becomes 5
cout << *ptr1; // 5
```

Example

- Write a program to:
  - Declare one double.
  - Declare a pointer pointing to the double.
  - Modify the double's value through the pointer.

```
int main()
{
  double a = 10.5;
  double* ptrA = &a;
  cout << a << endl; // 10.5
  *ptrA = 5.3;
  cout << a << endl;
  // *ptrA++; // this is *(ptrA++)
  // cout << a << endl;
  (*ptrA)++;
  cout << a << endl; // 6.3
}
```


Address operators

- What is **`*&x`** if **`x`** is a variable?
  - **`&x`** is the address of **`x`**.
  - **`*(&x)`** is the variable stored in that address.
  - So **`*(&x)`** is **`x`**.
- What is **`&*x`** if **`x`** is a pointer?
  - If **`x`** is a pointer, **`*x`** is the variable pointed by **`x`** (**`x`** stores an address!).
  - **`&*x`** is the address of **`*x`**, which is exactly **`x`**.
- What is **`&*x`** if **`x`** is not a pointer?

Good programming style

- Initialize a pointer variable as **`0`** or **`NULL`** if no initial value is available.
  - **`0`** is the standard in C++, while **`NULL`** is the standard in C. But they are the same when representing "null pointer".
  - By using **`NULL`**, everyone knows the variable must be a pointer, and you are not talking about a number or character.
- Without an initialization, a pointer points to **somewhere**... And we do not know where it is!
  - Accessing an unknown address results in unpredictable results.
  - Dereferencing a null pointer shutdowns the program.
- Example "**07_03_null**".

Good programming style

- It should be noted that when we use **`*`** in **declaring** a pointer variable, that **`*`** is not the dereference operator.
  - We are not getting any variable from any address.
  - It is just a special way of declaring a pointer variable.
- Thus, think **`int*`** as a type, which represents an "integer pointer".
  - **`int a;`** // **`a`** is an integer variable
  - **`int* p = &a;`** // **`p`** is pointing to an integer
- Thus, use "**`int* p`**" instead of "**`int *p`**".

Good Habits

- However:
  - `int* p, q;` // **p** is pointer, **q** is integer
  - `int *p, *q;` // two pointers
  - `int* p, *q;` // two pointers
  - `int* p, * q;` // two pointers
- Use several statements instead.

Outline

- The concept of pointers
- Syntax
- **Call by reference/pointer**
- Pointers and arrays
- Dynamic memory allocation (DMA)

References and pointers

- Recall this example:

```
void swap(int x, int y);
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << endl; // 10 20
  swap(a, b);
  cout << a << " " << b << endl; // still 10 20
  return 0;
}
void swap(int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}
```


References and pointers

- When invoking a function and passing parameters, the default scheme is "**call by value**" (or "pass by value").
  - The function declares its own local variables, using a copy of the arguments' values as initial values.
  - Thus we swapped the two local variables declared in the function, not the original two we want to swap.
- To solve this, we can use "**call by reference**" or "call by pointer."
  - They are somewhat different, but the principle is the same.
  - It is enough to know and use only one of them.

Call by reference

- A **reference** is a variable's alias.
- The reference is another variable that refers to the variable.
- Thus, using the reference is the same as using the variable.

```
int c = 10;
int& d = c; // declare d as c's reference
d = 20;
cout << c << endl; // 20
```

- `int& d = c;` // declare `d` as `c`'s reference
  - This `&` is different from the `&` operator which returns a variable's address.
- `int& d = 10` is an error.
  - A literal cannot have an alias!

Call by reference

- Now we know how to change a parameter's value:
  - instead of declaring a usual local variable as a parameter in the function, declare a **reference** varaible.
- This is "call by reference".

```
void swap(int& x, int& y);
  // declare reference variables
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << endl; // 10 20
  swap(a, b);
  cout << a << " " << b << endl; // 20 10
  return 0;
}
void swap(int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}
```


Call by reference

- The function

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

works as:

```
void swap()
{
  int& x = a; // a is the 1st argument
  int& y = b; // b is the 2nd argument
  int temp = x;
  x = y; // a will be modified
  y = temp; // b will be modified
}
```


Good programming style

- Thus we can call by reference and modify our parameters' value.
- When calling by reference, the only thing you have to do is to add a `&` in the parameter declaration in the function header.
- Mostly people use references only in call by reference.
- View the `&` in declaration as a part of type.
- Use
  - `int& a = b;`

  instead of
  - `int &a = b;`

Returning multiple values

- To return several values simultaneously, you can call by reference.
- Example: Write a program containing a function, which receives two doubles and returns their summation, difference, product and quotient.

```
void calc(double a, double b,
  double& sum, double& diff,
  double& prod, double& quo)
{
  sum = a + b;
  diff = a - b;
  prod = a * b;
  quo = a / b;
}
```


Call by pointers

- The principle behind calling by reference and calling by pointer is the same.
- You can view calling by reference as a special instrument made by using pointers.
- You can use calling by reference in most situations, and it is clearer and more convenient than call by pointer.
- You may need calling by pointer in some cases.

Call by pointers

- To call by pointers:
  - Declare a pointer variable as a parameter.
  - Pass a pointer variable or an address returned by **&** when invoking.
- For the **swap()** example:

```
void swap(int* ptrA, int* ptrB)
{
  int temp = *ptrA;
  *ptrA = *ptrB;
  *ptrB = temp;
}
```

- Invocation becomes **swap(&a, &b);**

Call by pointers

- How about the following implementation?

```
void swap(int* ptrA, int* ptrB)
{
  int* temp = ptrA;
  ptrA = ptrB;
  ptrB = temp;
}
```

  - Invocation is still **swap(&a, &b);**
- Will the two arguments be swapped? What really happens?

Good programming style

- Do not mix references and pointers!
  - E.g., we cannot pass a pointer variable or an address to a reference!
- When you just want to modify arguments or return several values, call by reference.
- When you really have to do something by pointers, call by pointer.

Outline

- The concept of pointers
- Syntax
- Call by reference/pointer
- **Pointers and arrays**
- Dynamic memory allocation (DMA)

Passing an array

- We may pass an array as a function argument.
- Simply add `[]` after the array name in the function header.
- Don't include `[]` in invocation.
- The array length need not be specified in the function header.
- But we need to know the array length by ourselves.

```
void func(int a[]);
  // an array as a argument
int main()
{
  int a[5] = {1, 2, 3, 4, 5};
  print(a); // 1 2 3 4 5
  return 0;
}
void print(int a[])
{
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
}
```


Passing an array

- Consider this example:
- Call by value or reference/pointer?

```
void func(int a[]);
int main()
{
  int a[5];
  func(a);
  for(int b = 0; b < 5; b++)
    cout << a[b] << " "; // 10 10 10 10 10
  return 0;
}
void func(int a[])
{
  for(int b = 0; b < 5; b++)
    a[b] = 10;
}
```

Pointers and arrays

- In fact, an array variable **is** a pointer!
  - It points to the **first** element of the array.
- When passing an array, we pass the pointer.
  - That is why we do not include `[]` in invocation.
- Example "**07_04_arrayPointer**".
- To further understand this issue, let's study **pointer arithmetic**.
  - Since we are dealing with address, we should be very careful!

Pointer arithmetic

- Usually, one arbitrary address returned by performing arithmetic on a pointer variable is useless to us.

```
int a = 10;
int* ptr = &a;
cout << ptr++;
  // just an address
  // we don't know what's here
cout << *ptr;
  // dangerous!
```


Pointer arithmetic

- The arithmetic is useful (and should be used) only when you can predict a variable's address.
- For example, where there are variables stored continuously:

```
int a[3] = {10, 20, 30};
int* ptr = &a[0];
ptr++;
cout << ptr; // may it be 20's address?
cout << *ptr; // may it be 20?
```


Pointer Arithmetic: ++ and −−

- The type a pointer pointing to is used as the unit of measurement.
- **++**: Increment the pointer variable's value by the number of byte a variable in this type occupies (i.e., point to the next variable).
  - e.g., for integer pointers, the value (an address) increases by 4 (bytes).
- **−−**: Decrement the pointer variable's value by the number of byte a variable in this type occupies (i.e., point to the previous variable).

```
double a[3] = {10.5, 11.5, 12.5};
double* b = &a[0];
cout << *b << endl;  // 10.5
b++;  // you may also write b = b + 1
cout << *b << endl;  // 11.5
b++;
cout << *b << endl;  // 12.5
```

Pointer Arithmetic: –

- We cannot add two address.
- However, we can find the difference of two addresses.

```
double a[3] = {10.5, 11.5, 12.5};
double* b = &a[0];
double* c = &a[2];
cout << c - b << endl; // 2, not 16!
```


Pointers and arrays

- Changing the value stored in a pointer can be dangerous:

```
int x[3] = {1, 2, 3};
int* ptr = x;
for(int y = 0; y < 3; y++)
  cout << *(ptr + y) << " "; // 1 2 3
for(int y = 0; y < 3; y++)
  cout << *(ptr++) << " "; // 1 2 3
for(int y = 0; y < 3; y++)
  cout << *(ptr + y) << " "; // unpredictable
```


Outline

- The concept of pointers
- Syntax
- Call by reference/pointer
- Pointers and arrays
- **Dynamic memory allocation (DMA)**

DMA

- In C/C++, if you declare an array by specifying it's length as a constant variable or a literal, the memory space will be allocated to it during the compiling time.
  - `int a[100];` // 400 bytes are allocated
- This is called "**static memory allocation**".

DMA

- You may decide the length of an array "**dynamically**", i.e., during the **running** time.
- To do so, you must use a different syntax to ask C/C++ not to allocate memory during compilation.
- Not only arrays but also other types of variables may also be declared in this way.

DMA: `new`

- The operator **`new`** will allocate a memory space **and** return the address.
  - In C there is a different keyword **`melloc`**.
- **`new int;`** // allocate 4 bytes without recording the address
- **`int* a = new int;`** // **`a`** points to the space
- **`int* a = new int(5);`** // the space contains 5 as the value
- **`int* a = new int[5];`**

  // allocate 20 bytes (5 integers). **`a`** points to the first integer. S

DMA: `new`

- All of these spaces are allocated when the program is running.
- So you can write

```
int len = 0;
cin >> len;
int* a = new int[len];
```

  This allocates a space according to the input from users.

DMA: `new`

- Note that a space allocated during the running time has **no name** for it!
  - On the other hand, every space allocated during compilation has a name.
  - So the system knows how to access every statically-allocated spaces.
- To access a dynamically-allocated space, there must be a pointer storing its address.

DMA: memory leak

- For spaces allocated during **compilation**, the system will **release these spaces** automatically when the corresponding variables no longer exist.

```
void func(int a)
{
  double b;
} // these 4 + 8 bytes are released
```


DMA: memory leak

- For spaces allocated during the **running** time, the system will **NOT** release these spaces unless it is asked to do so.
  - Because the space has no name!

```
void func()
{
  double* b = new double;
}
// 4 bytes for b are released
// 8 bytes for new double are not
```


DMA: memory leak

- The programmer must keep a record for all spaced allocated dynamically.

```
double* b = new double;
*b = 5.2;
double c = 10.6;
b = &c; // now no one can access
        // the space containing 5.2
```

- This problem is called **memory leak**. You lose the control of allocated spaces. Also, these spaces are **wasted**.
  - They will be released until the program ends.

DMA: `delete`

- The **`delete`** operator will release the space so that the space can be allocated to other variables in the future.

```
int* a = new int;
delete a; // release these 4 bytes
int* b = new int[5];
// ...
delete b;
  // release only 4 bytes!
  // Unpredictable results may happen
delete [] b; // release all 20 bytes
```

DMA: `delete`

- The **`delete`** operator will do nothing to the pointer. To avoid reuse the released space, set the pointer to **`NULL`**.

```
int* a = new int;
delete a;  // a is still pointing to the address
a = NULL;  // now a points to nothing
int* b = new int[5];
delete [] b;  // b is still pointing to the address
b = NULL;  // now b points to nothing
```


Good programming style

- Use DMA for arrays with no **predetermined** length.
- Whenever you write a **`new`** statement, add a **`delete`** statement below immediately (unless you know you really do not need it).
- Whenever you want to change the value of a pointer, check whether memory leak occurs.
- Whenever you write a **`delete`** statement, set of pointer to **`NULL`**.