CS2311 Computer Programming

LT9: Pointers

Part I

Outline

- Memory and variables
- Pointers and pointer operations
- Call by Pointers
- Call by reference

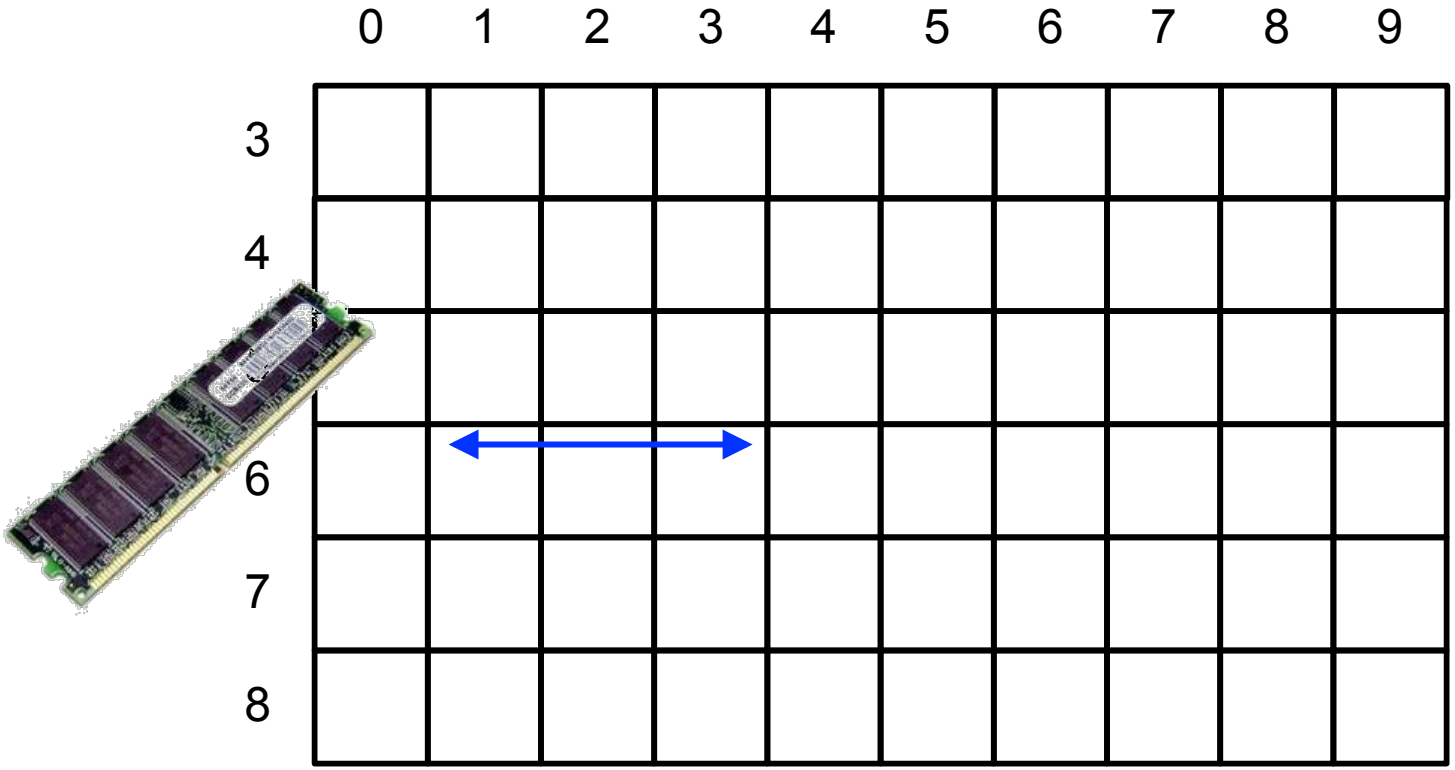
Memory and Variables

- A **variable** is used to store data that will be accessed by a program on execution.

- Normally, a **variable** is stored in the **main memory**

- A variable has **four** attributes:
  - **Value** - the content of the variable
  - **Identifier** - the name of the variable
  - **Address** - the memory location of the variable
  - **Scope** - the accessibility of the variable

Main Memory

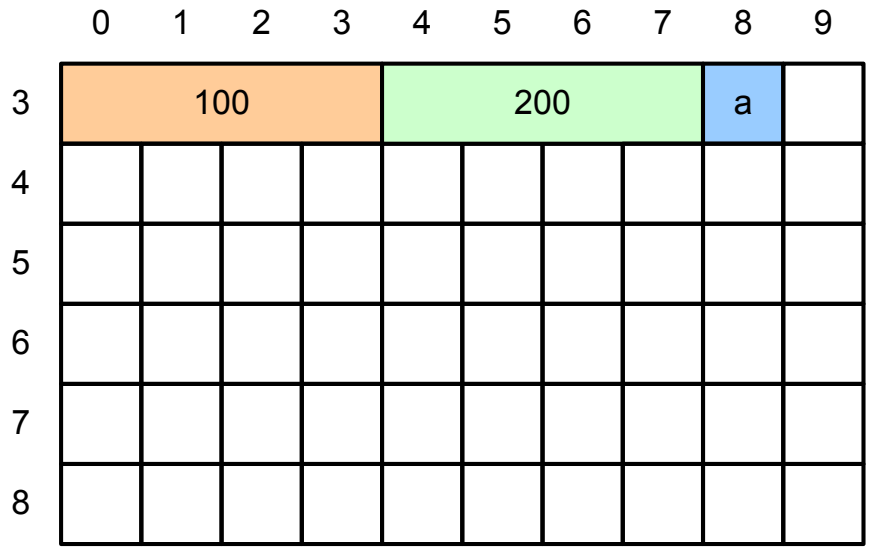


Variable and Memory

```
int main() {

   int x;
   int y;
   char c;
   x = 100;
   y = 200;
   c = 'a';

   return 0;
}
```



Identifier	Value	Address
x	100	30
y	200	34
c	'a'	38

Variable and Memory

- *Most of the time*, the computer allocates adjacent memory locations for variables declared one after the other.
- A variable's **address** is the **first byte** occupied by the variable.
- Address of a variable depends on the computer, and is usually in ***hexadecimal*** (base 16 with values 0-9 and A-F).
  - e.g. `0x`**`00023AF0`**, `00023AF0`

Pointers

- A **pointer** is a variable which stores the address of another variable, i.e. it points to the variable.

- A pointer, like a regular variable, has a type. Its type is determined by the type of the variable it points to.

Variable type	`int i`	`float f`	`double d`	`char ch`
Pointer type	`int* iptr`	`float* fptr`	`double* dptr`	`char* chptr`

* and & Operators

- To declare a pointer, use "*****" before an identifier name or after the type:
  - `char  *cptr;  // a character pointer`
  - `int   *nptr;  // an integer pointer`
  - `float *fp;    // a floating point pointer`
- To retrieve the address of a variable, use the "**&**" operator (referencing):
  - `int x;`
  - `nptr = &x;   // &x returns the address of x;`
- To access the variable a pointer pointing to, use "*****" operator (dereferencing)
  - `*nptr = 10; // x=10`
  - `int y;`
  - `y = *nptr;  // y=x`

reference vs **dereference**
& *****

Example

```
int x, y;           // x and y are integer variables
int main() {

   int *p1,*p2;     // p1 and p2 are pointers of integer type
   x = 10;
   y = 12;
   p1 = &x;         // p1 stores the address of  variable x
   p2 = &y;         // p2 stores the address of variable y
   *p1 = 5;         // p1 value unchanged but x is updated to 5
   *p2 = *p1+10;    // what are the values of *p2 and y?

   return 0;
}
```

Common Operations

```
int* p1; int* p2; int x;
```

- Set a pointer **p1** point to a variable **x**

```
p1 = &x;
```

- Set a pointer **p2** point to the variable pointed by another pointer **p1**

```
p2 = p1;
```

- Update the value of variable pointed by a pointer

```
*p2 = 10;
```

- Retrieve the value of variable pointed by a pointer

```
int y = *p2;
```

Important Note.
Before dereferencing a pointer, make sure it is pointing to a valid memory location [a static variable]

Summary

- ***** operator will give the ***value*** of pointing variable
  - so that you can *indirectly* update/modify the pointing variable
  - e.g., `int x; int *p = &x;`
    - then using "`*p`" is equal to "`x`";

- **&** operator will give the ***address*** of a variable

Exercise: Errors?

```
int x = 3;
char c = 'a';
char *ptr;
ptr = &x;
ptr = c;
ptr = &c;
```

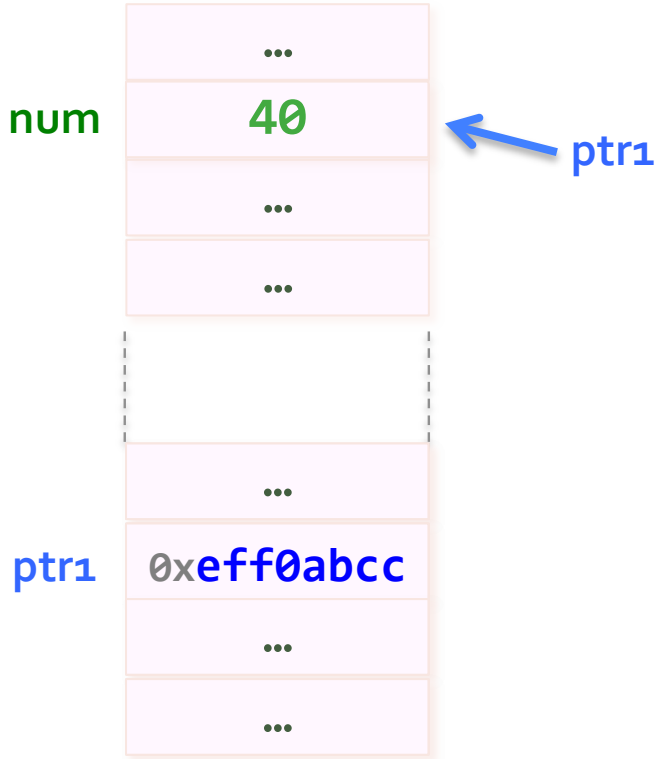
Answer

```
int x = 3;
char c ='a';
char *ptr;
ptr = &x; // error: ptr can only points to a char, but not int
ptr = c;  // error: cannot assign a char to a pointer.
          // A pointer can only store a location
ptr = &c; // correct
```

Exercise: What's the Output?

```
int num = 100;
int *ptr1;

ptr1 = &num;
*ptr1 = 40;
cout << num;
```



Answer

```
int num = 100;
int *ptr1;

ptr1 = &num;
*ptr1 = 40;
cout << num; // print 40
```

`const` Pointer and Pointer to `const`

- If we add keyword **`const`** at the right-hand side of the '*****' sign, the declared pointer is a ***constant pointer***
- A constant pointer must be initialized in declaration
- We cannot change the address that a constant pointer is pointed to
- But we can still modify the value of the variable that the constant pointer is pointed to

```
int num = 100;
int * const ptr1 = &num;    //initialization
*ptr1 = 40; // value of num changes to 40
cout << num;
```

`const` Pointer and Pointer to `const`

- If we add keyword **`const`** at the *left-hand side* of the '*****' sign, this pointer is pointed to a ***constant value***
- This pointer can point to other constant values later. However, we cannot change the value that this pointer is pointed to

```
const int num1 = 100;
const int num2 = 150;
int const * ptr1; // or const int * ptr1

ptr1 = &num;
*ptr1 = 40; // illegal: cannot change the
            // value of const int
ptr1 = &num2;
```

Array of Pointers

- We can define a "pointer array" to manage multiple pointers. For example: `int *n[5];`
- We can use the '*' sign to change the value of the variable that each pointer (in the array) is pointed to

```
int *n[5];
int a[5] = {0}; // initialize to all 0

for(int i = 0; i<5; i++) {
  n[i] = &a[i];
  *n[i] = i;
}
// value of a[] changes to 0,1,2,3,4
```

Applications of Pointers

- **Call by Reference**
- **Fast Array Access**
  - Will be covered in later class
- **Dynamic Memory Allocation**
  - Require ***additional*** memory space for storing value.
  - Similar to variable declaration but the variable is stored outside the program.

Call By **Pointer**

- Pass the ***address*** of a **variable** to a function
- ***call by value*** cannot be used to update arguments to function
- Consider the following function

```
void f (char c1_in_f) {
   c1_in_f = 'B';  //c1_in_f=66
}

int main() {

   char c1_in_main = 'A';     // c1_in_main=65
   f(c1_in_main);
   cout << char(c1_in_main); // print 'A'

   return 0;
}
```

Call By **Value**

- When calling **f()**, the value of **c1_in_main (65 or 'A')** is assigned to **c1_in_f**
- Inside **f()**, the value of **c1_in_f** is changed to **66** or **'B'**.
- **c1_in_main** remains unchanged (**65** or **'A'**).
- In call by value, there is no way to modify **c1_in_main** inside **f()**, unless we use the return statement.

```
void f(char c1_in_f) {
   c1_in_f = 'B';    //c1_in_f=66
}
int main() {
   char c1_in_main = 'A';        // c1_in_main=65
   f(c1_in_main);
   cout << char(c1_in_main);     // print 'A'
   return 0;
}
```

Call By Value

- When calling **f()**, the value of **c1_in_main (65 or 'A')** is assigned to **c1_in_f**
- Inside **f()**, the value of **c1_in_f** is changed to 66 or 'B'.
- **c1_in_main** remains unchanged (65 or 'A').
- In call by value, there is no way to modify **c1_in_main** inside **f()**, unless we use the return statement.

```
void f(char c1_in_f) {
   c1_in_f = 'B';   //c1_in_f=66
}
int main() {
   char c1_in_main = 'A';       // c1_in_main=65
   f(c1_in_main);
   cout << char(c1_in_main);    // print 'A'
   return 0;
}
```

Call by Pointer – Guidelines

- Add the '*' sign to the function parameters that store the variable call by pointer

```
void cal(int *x, int y) {
  //x is call by pointer
  //y is call by value
  //:
}
```

- Add the '&' sign to the variable when it needs to be call by pointer

```
cal(&result, factor);
```

Call By Pointer

```
void f(char *c1_ptr) {
   *c1_ptr = 'B';
}

int main() {
   char c1_in_main = 'A';  // c1_in_main =65
   f(&c1_in_main);
   cout << c1_in_main;     // print 'B'
   return 0;
}
```

When `f()` is called, the following operation is performed

```
c1_ptr = &c1_in_main;
```

Call By Pointer

```
void f (char *c1_ptr) {
   *c1_ptr = 'B';
}
int main() {
   char c1_in_main = 'A';     // c1_in_main=65
   f(&c1_in_main);
   cout << c1_in_main;  // print 'B'
   return 0;
}
```

points to

Variable	Variable type	Memory location	Content
c1_in_main	char	**3A8E**	65
c1_ptr	char pointer	4000	3A8E

Assign location 3A8E to c1_ptr

Location of **c1_in_main** (location 3A8E) is assigned to c1_ptr1

```
c1_ptr = &c1_in_main;
```

Call By Pointer

```
void f (char *c1_ptr) {
   *c1_ptr = 'B';
}
                  update
int main() {
   char c1_in_main = 'A'; // c1_in_main = 65
   f(&c1_in_main);
   cout << c1_in_main;    // print 'B'
   return 0;
}
```

c1_ptr points to location 3A8E (that is the variable c1_in_main). ***c1_ptr** refers to the variable pointed by **c1_ptr**, i.e. the variable stored at 3A8E

Variable	Variable type	Memory location	Content
c1_in_main	char	3A8E	**66**
c1_ptr	char pointer	4000	3A8E

```
c1_ptr = 'B';   // error
```

Reason: **c1_ptr** stores a **location** so it cannot store a char (or the ASCII code of a char)

Note the different meaning of *

The type of **c1_ptr** is **char*** (pointer to char)

dereference a pointer

***c1_ptr = 'B';**

```
void f(char *c1_ptr) {
   *c1_ptr = 'B';
}

int main() {
   char c1_in_main = 'A'; // c1_in_main=65
   f(&c1_in_main);
   cout << c1_in_main;   // print 'B'
   return 0;
}
```

Call by Value and Call by Pointer

- In **call by value**, only a single value can be returned using a *return statement*
- In **call by pointer**, the argument(s) can be a pointer which may point to the variable(s) in the caller function
  - *More than one* variables can be updated, achieving the effect of returning multiple values.

Call by Value vs Call by Pointer

Call-by-Value

```
int add(int m, int n) {
   int c;
   c = m + n;
   m = 10;
   return c;
}

int main() {
   int a = 3, b = 5, c;
   c = add(a,b);
   cout << a << " " << c << endl;
   return 0;
}
```

Pass value as parameter.

Call-by-Pointer

```
int add(int *m, int *n) {
   int c;
   c = *m + *n;
   *m = 10;
   return c;
}

int main() {
   int a = 3, b = 5, c;
   c = add(&a, &b);
   cout << a << " "<< c << endl;
   return 0;
}
```

Pass address as parameter.

Call by Reference vs Call by Pointer

Call-by-Reference

```
int add(int &m, int &n) {
   int c;
   c = m + n;
   m = 10;
   return c;
}

int main() {
   int a = 3, b = 5, c;
   c = add(a,b);
   cout << a << " " << c << endl;
   return 0;
}
```

Pass address as parameter.

Call-by-Pointer

```
int add(int *m, int *n) {
   int c;
   c = *m + *n;
   *m = 10;
   return c;
}

int main() {
   int a = 3, b = 5, c;
   c = add(&a, &b);
   cout << a << " "<< c << endl;
   return 0;
}
```

pointer

Pass value as parameter.

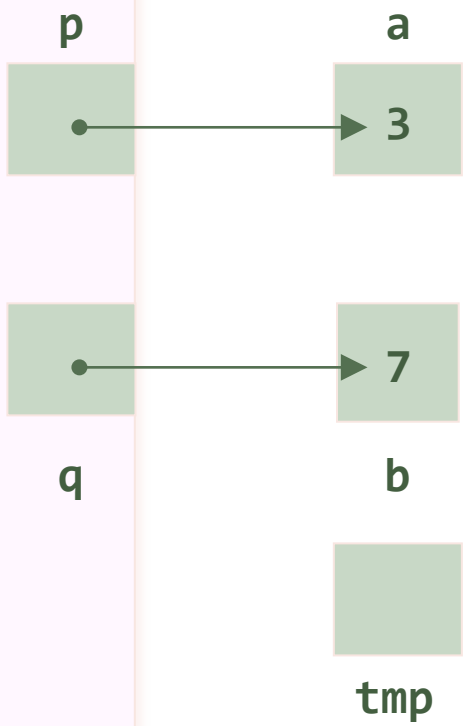
Example: swapping values

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

void swap(int *p, int *q) {
   int   tmp;

   tmp = *p;   // tmp = 3
   *p = *q;    // *p = 7
   *q = tmp;   // *q = 3
}

int main() {
   int   a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl; // 7 3 is printed
   return 0;
}
```



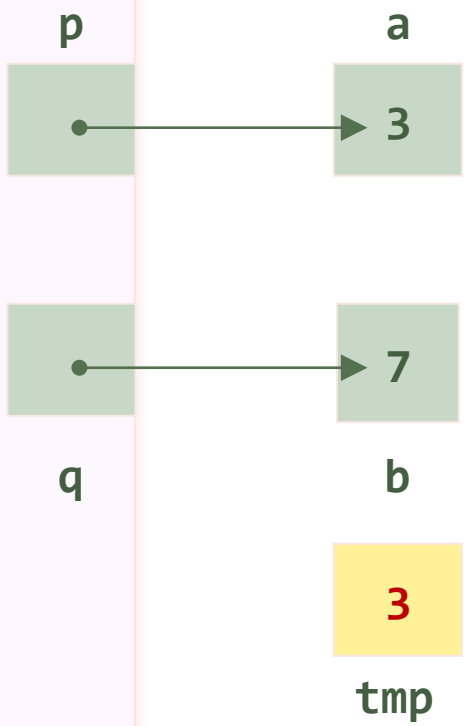
Example: swapping values

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

void swap(int *p, int *q) {
   int   tmp;

   tmp = *p;   // tmp = 3
   *p = *q;    // *p = 7
   *q = tmp;   // *q = 3
}

int main() {
   int   a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl; // 7 3 is printed
   return 0;
}
```



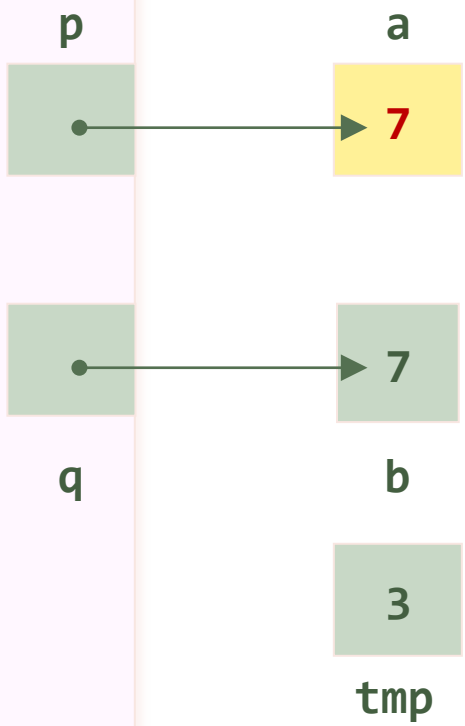
Example: swapping values

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

void swap(int *p, int *q) {
   int   tmp;

   tmp = *p;   // tmp = 3
   *p = *q;    // *p = 7
   *q = tmp;   // *q = 3
}

int main() {
   int   a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl; // 7 3 is printed
   return 0;
}
```



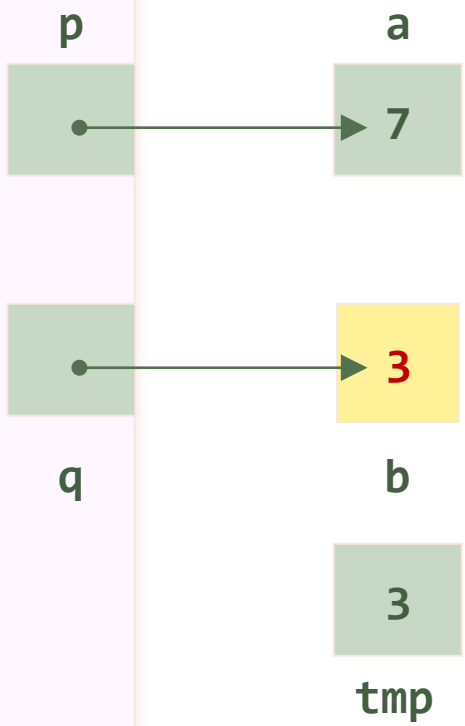
Example: swapping values

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

void swap(int *p, int *q) {
   int   tmp;

   tmp = *p;   // tmp = 3
   *p = *q;    // *p = 7
   *q = tmp;   // *q = 3
}

int main() {
   int   a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl; // 7 3 is printed
   return 0;
}
```



Call by Reference

- Reference is another "name" of a variable

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

void swap(int &a, int &b) {
   int   tmp;
   tmp = a;    // tmp = 3
   a = b;      // a = 7
   b = tmp;    // b = 3
}

int main() {
   int   a = 3, b = 7;
   swap(a, b);
   cout << a << " " << b << endl; // 7 3 is printed
   return 0;
}
```

Summary

- Pointer is a special variable used to store the memory address (location) of another variable.
- Pointer is typed, and its type is determined by the variable it pointing to.
- * operator has two meaning
  - For declaration, e.g `int *p1, char *pch;`
  - For dereference, e.g. `x=*p1, *pc='b';`
- & operator return the address of any variable
- Parameter passing
  - call by value: *a copy of value – single value*
  - call by pointer (value): *a copy of the pointer/address – multiple values*
  - call by reference: *an address – multiple values*