Introduction to pointers

The pointer lets us directly access the address of the physical memory. To obtain the address, there is the address operator given as an ampersand, &, and the pointer is indicated by operating an asterisk, *, which is called the indirection operator. The advantages to use pointers are:

1. Simplifying the program code;
2. Reducing the execution time;
3. Helping construct complicated data structures.

- **The basics**

Here is a basic code using a pointer:

```
#include<stdio.h>
main()
{
	int a = 50;
	int *it;	//Declare a pointer variable
	it = &a;	//Input the address of "a"
	printf("it = %d", *it);
}
```

The output will be:

it = 50

In the above code, when "it" points the object, "a", "*it" is "a" itself. In a word, "*it" is the shortcut of "a." and "*it" is called the alias of "a" technically. The pointer variable returns the value of "a" located at the specified address, &a. The following code is almost identical as above except inside the print command:

```
#include<stdio.h>
main()
{
	int a = 50;
	int *it;
	it = &a;
	printf("it = %d", it); (Using "it" instead of "*it")
}
```

The output becomes

it = 2686616

This is the address number. Thus, “it + 1” gives you the next location of the address.

- **More detailed explanations**

Pointers and normal variables denote the values and addresses differently. The following table indicates the correspondent expressions:

	Value	**Address**
Declare as, double var;	var	&var
Declare as, double *var;	*var	var

This means that a variable declared as a pointer, say, “int *it;”, then “it” (without asterisk) gives the address. Therefore, “it = &a;” can be held grammatically. Here are several examples. The next couple of examples indicate the confusion of the variables to input.

Wrong		Correct

`main()`
`{`
`  int a = 50;`
`  int *it;`
 	
`  *it = &a;`
`}`
		`main()`
		`{`
		`  int a = 50;`
		`  int *it;`
		
		`  it = &a;`
		`}`

Wrong		Correct

`main()`
`{`
`  int a = 50;`
`  int *it;`
 	
`  it = a;`
`}`
		`main()`
		`{`
		`  int a = 50;`
		`  int *it;`
		
		`  *it = a;`
		`}`

Wrong		Correct

`main()`
`{`
`  int a, *it, *that;`
`  a = 50;`
`  it = &a;`
		`main()`
		`{`
		`  int a, *it, *that;`
		`  a = 50;`
		`  it = &a;`

`*it = that;` `}`		`it = that;` `}`

Wrong		Correct

`main()`
`{`
`int a, *it, *that;`
`a = 50;`
`it = &a;`
`it = *that;`
`}`
		`main()`
		`{`
		`int a, *it, *that;`
		`a = 50;`
		`it = &a;`
		`*it = *that;`
		`}`

An object declared by the storage-class specifier cannot use the address operator. The following code will give you the compiler error. (You can do the following in C++.)

```
#include<stdio.h>
main()
{
        register int a;     (“register” is the storage-class specifier.)
        printf("&a = %d", &a); (“&a” is prohibited in C.)
}
```

IMPORTANT!!

The following code would harm the operating system. **Do not run this program.**

```
#include<stdio.h>
main()
{
        int *pp;
        *pp = 2;
}
```

The above code inputs the number into the pointer variable; however, the address is not certain of where exactly it is located in the memory. Therefore, it may go out of control if an important address is replaced by the above action. Note that you must specify an address (initialize the pointer) before you put the number in the pointer address. The following is a correct code:

```
#include<stdio.h>
main()
```

```
{
	int a,*pp;
	pp = &a;
	*pp = 2;
}
```

The other related example is:

```
#include<stdio.h>
main()
{
	int t1,p0,*p1;
	p0 = 123;
	p1 = &p0;
	t1 = *p1;
	printf("%d", t1);
}
```

The output is

123

Another example shown below explains the line-by-line processes:

```
main()
{
	int a = 3;
	int *it,*that;
	that = &a; (The address of “a” is put into the pointer “that.”)
	it = that; (“it” is referenced by “that”.)
  (Until here, “*it” and “*that” have the value of “a”, which is 3.)
	a = a + 3;
  (The current “a” becomes 6 and “it” and “that” are referenced to this too.)
	*it = *it + 3;
   (Now, “*it” becomes 6 + 3; namely, 9.)
}
```

- **Arrays and pointers**

Here is an example code:

```
main()
```

```
{
	int b[15];
	int *pt;

	pt = &b[0];
}
```

The “pt” is a pointer to register the first label of the array, “b[ ].” In fact, “b[0]” is equivalent with “*pt” and you can express b[1] as *(pt+1). In C, you can find identical expressions for arrays. Each raw of the following table are equal:

b[0]	*b	*pt	pt[0]
b[1]	*(b+1)	*(pt+1)	pt[1]
b[i]	*(b+i)	*(pt+i)	pt[i]

Likewise, there are 4 equivalent address expressions:

&b[0]	b	pt	&pt[0]
&b[1]	b+1	pt+1	&pt[1]
&b[i]	b+i	pt+i	&pt[i]

The following code explains that one can make a pointer for multiple elements of an array.

```
#include<stdio.h>
main()
{
	int array[5];
	int *pini;
	int (*parray)[5];

	pini = array;   (Only for the first value of the array)
	parray = &array;  (For all of the elements of the array)

	printf("pini = %d, ",pini);
	printf("parray = %d¥n",parray);

	pini++;
	parray++;

	printf("pini = %d, ",pini);
```

```
        printf("parray = %d¥n",parray);
}
```

pini = 2686596, parray = 2686596

pini = 2686500, parray = 2686616

As you can see in the above results, after the increment, "pini" is only changed by one element that is 4 bytes. "parray" is increased by 5 elements that is $5 \times 4 = 20$ bytes.

- **Applications** (A reason why you need pointers)

There is a typical example, which exchanges two values. The following code does not work properly although it looks logical:

```
#include <stdio.h>
void Swap(int a, int b);

void main(void)
{
   int  a = 10, b = 20;
   int pa, pb;

   pa = a;
   pb = b;

   printf("  Before swapping: a = %d   b = %d¥n", a, b);

   Swap(a, b);
   printf("  After swapping: a = %d   b = %d¥n", a, b);

   Swap(pa, pb);
   printf(" After re-swapping: a = %d   b = %d¥n", a, b);
}

void Swap(int a, int b)
{
   int  job;

   work = a;
   a = b;
```

```
    b = job;
}
```

The results are:

Before swapping: a = 10 b = 20

After swapping: a = 10 b = 20

After re-swapping: a = 10 b = 20

The algorithm seems correct, but the outputs do not give the right results. This is because only values are passed among functions in C. Let us modify the above code with pointers.

```
#include <stdio.h>
void Swap(int *a, int *b);

void main(void)
{
    int  a = 10, b = 20;
    int *pa, *pb;

    pa = &a;
    pb = &b;

    printf("  Before swapping: a = %d   b = %d¥n", a, b);

    Swap(&a, &b);
    printf("  After swapping: a = %d   b = %d¥n", a, b);

    Swap(pa, pb);
    printf(" After re-swapping: a = %d   b = %d¥n", a, b);
}

void Swap(int *a, int *b)
{
    int  job;
    job = *a;
    *a = *b;
    *b = job;
}
```

The outputs are:

Before swapping: a = 10 b = 20

After swapping: a = 20 b = 20

After re-swapping: a = 10 b = 20

In the second swapping, the arguments of Swap(pa, pb); are pointers and stored their addresses, so it is equivalent with Swap(&a, &b);. The above code can be simplified as follows:

```
#include <stdio.h>
void Swap(int *a, int *b);

void main(void)
{
   int  a = 10, b = 20;

   Swap(&a,&b);
   printf("Swapped: a = %d b = %d¥n", a, b);
}

void Swap(int *a, int *b)
{
   int job;
   job = *a;
   *a = *b;
   *b = job;
}
```

- **Multiple indirection**

You can create pointer's pointer.

```
#include<stdio.h>
main()
{
      int a = 5;
      int *b, **c, ***d;

      b = &a;
      c = &b;
```

```
        d = &c;

        printf("%d %d %d",*b,**c,***d);
}
```

The output becomes

5 5 5

The above code expresses the following relationships:

***d = **c = *b = a = 5

**d = *c = b (the address of a)

*d = c (the address of b)

Another example involves an array:

```
#include<stdio.h>
main()
{
        int set[4] = {0, 1, 2, 3};
        int *px, **py;

        px = &set[0];   ("px = set" is also possible for the expression.)
        py = &px;
        printf("%d¥n", **py);

        px = px + 2;
        printf("%d¥n", **py);
}
```

You will get the output:

0

2

The pointer, px, corresponds to the element of the array, set. Then, the pointer's pointer, py, is changed by the manipulation of px.

- **Function pointers**

Functions can be addressed by pointers. For example,

```
#include<stdio.h>
double func1(double v)
{
```

```
	return 5.0*v + 2.0;
}
main()
{
/* Set the address of the function to the pointer. */
	double (*pf)(double) = func1;
/* Put a value in the pointer indicating the function. */
	double outcome = pf(3.0);
	printf("The output is %lf.", outcome);
}
```

The output is 17.000000. (Because $5.0 \times 3.0 + 2.0 = 17.0$)

The function pointer can be altered in the main function:

```
#include<stdio.h>
#include<math.h> /* Don't forget the compile option, -lm. */
double func1(double v)
{
	return 5.0*v + 2.0;
}
main()
{
	double (*pf)(double) = func1;
	double out1 = pf(3.0);
/* Set a different operation (sine) for the pointer. */
	pf = sin;
	double out2 = pf(3.0);
	printf("out1 = %lf and out2 = %lf.", out1,out2);
}
```

```
out1 = 17.000000 and out2 = 0.141120.
```

The following code is equivalent with the above:

```
#include<stdio.h>
#include<math.h> /* Don't forget the compile option, -lm. */
double func1(double v)
{
	return 5.0*v + 2.0;
```

```
}
double func2(double v)
{
	return sin(v);
}
main()
{
	double (*pf)(double) = func1;
	double out1 = pf(3.0);
/* Set a different function for the pointer. */
	pf = func2;
	double out2 = pf(3.0);
	printf("out1 = %lf and out2 = %lf.", out1,out2);
}
```

Multiple functions can be bundled as an array by using pointers.

```
#include<stdio.h>
#include<math.h>
double func1(double v, double w)
{
	return log(v) - tan(w);
}
double func2(double v, double w)
{
	return cos(v)*sin(w);
}
double func3(double v, double w)
{
	return exp(v)/sqrt(w);
}
main()
{
	int i;
	double (*pf[3])(double, double) = {func1,func2,func3};

	for(i=0;i<3;i++)
```

```
	{
		double out = pf[i](3.0, 5.0);
		printf("pf[%d] = %lf¥n",i,out);
	}
}
```

pf[0] = 4.479127
pf[1] = 0.949328
pf[2] = 8.982525

The function pointers let us make a versatile program especially when we have a function depending on the multiple sub-functions. For example, there is an algorithm to solve non-linear equations known as Newton's method. This method is one of the functions in the following code; and the equations to be solved are also included with their derivatives. Without function pointers, it may be tedious to show all of the solutions without pointers. If you use function pointers, the code looks tidy and easy to modify.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define max 1000              //Max repetition
#define eps 1.0e-6
(The equation to be solved)
double fa(double x)
{
	return sin(x)*cos(x);
}
(Newton's method needs the derivative of the above equation.)
double dfa(double x)
{
	return cos(x)*cos(x)-sin(x)*sin(x);
}
(Here is the second set of equations.)
double fb(double x)
{
	return sin(x);
```

```
}
double dfb(double x)
{
       return cos(x);
}
(This is the function of Newton's method which has an argument of
equations to be solved.)
void Newton(double anf, double (*func1)(double x), double
(*func2)(double x))
{
       int coun=0;
       double newanf;
(The following algorithm uses the pointer functions.)
       for(;;) {
              coun++;
              newanf=anf-(*func1)(anf)/(*func2)(anf);
              if(fabs(newanf-anf)<eps) break;
              anf=newanf;

              if(coun==max) {
                     printf("Not converged.¥n");
                     exit(1);}
       }
        printf("The solution: %f.¥n The # of repetition: %d.¥n",
          newanf,coun);
}

int main()
{
       double a=6.0; /*initial value*/
       double b=4.0; /*initial value*/

    Newton(a,fa,dfa);
    Newton(b,fb,dfb);
}
```

As you can see, there are two equations to be solved, but the function of Newton's

method can deal with only one of them. In the main function, you can call Newton's method twice with different equations. (The result is omitted.)

There are two more examples with function pointers. The following code indicates the data depending on grade points. Instead of using the "switch" command, function pointers make the program simpler. (The results are omitted.)

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int Agrade(int x)
{
	return x >= 80 && x <= 100;
}
int Bgrade(int x)
{
	return x >= 70 && x <= 79;
}
int Cgrade(int x)
{
	return x >= 60 && x <= 69;
}
void make_list(const int data[], int n, int (*class)(int))
{
	int i;
	for(i=0;i<n;i++){
		if((*class)(data[i]))
			printf("*");
		else
			printf(" ");
		printf("data[%d] = %d points¥n",i,data[i]);
	}
}

int main()
{
	int i;
```

```
	int data[10];
	int n=sizeof(data)/sizeof(data[0]); (The number of elements)

	srand(time(NULL));
	for(i=0;i<n;i++)
		data[i] = rand() % 101;

	puts("Excellent ------- ");
	make_list(data,n,Agrade);

	puts("¥nGood ------- ");
	make_list(data,n,Bgrade);

	puts("¥nOK ------- ");
	make_list(data,n,Cgrade);

	return 0;
}
```

This example depicts a case that the functions do not return values. Selecting an item from the menu points out each function to perform.

```
#include<stdio.h>

typedef enum {Bohr, Einstein, Feynman, Invalid} Physicist;

void bohr(void)
{
	puts("The investigation of the structure of atoms and of the
radiation emanating from them");
}

void einstein(void)
{
	puts("Theoretical Physics, and especially for his discovery
of the law of the photoelectric effect");
}
```

```
void feynman(void)
{
	puts("Fundamental work in quantum electrodynamics, with
deep-ploughing consequences for the physics of elementary
particles");
}

Physicist choose(void)
{
	int number;

	do{
	  printf("Select one of the following:¥n");
	  printf("0)--Bohr 1)--Einstein 2)--Feynman 3)--End: ");
	  scanf("%d",&number);
	}while(number<Bohr || number>Invalid);
	return number;
}

int main(void)
{
	Physicist chosen;
(The following declares the function pointer, and each function is executed
by the label of the element.)
	void (*pphys[])(void) = {bohr,einstein,feynman};

	do{
	  chosen = choose();
	  if(chosen >= Bohr && chosen < Invalid)
	       pphys[chosen](); (This is equivalent with (*pphys[chosen])();)
	}while(chosen != Invalid);
	return 0;
}
```

- **Summary**

Using pointers give us following advantages:

1. You can pass the values between functions from another scope without using global variables.
2. You can have more variations to handle arrays with pointers.
3. You can deal with multiple functions as parameters to use I other functions more conveniently.