# C++ Advanced Pointers

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize a pointer</td>
<td>3</td>
</tr>
<tr>
<td>Reference and dereference operators</td>
<td>4</td>
</tr>
<tr>
<td>The &amp; Operator — Reference To</td>
<td>4</td>
</tr>
<tr>
<td>Using pointers to pass values</td>
<td>5</td>
</tr>
</tbody>
</table>

## Pointers and Arrays

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator precedence</td>
<td>7</td>
</tr>
<tr>
<td>Pointer to pointer</td>
<td>7</td>
</tr>
<tr>
<td>Null pointer</td>
<td>8</td>
</tr>
<tr>
<td>Pointers to function</td>
<td>8</td>
</tr>
<tr>
<td>Pointer operators</td>
<td>11</td>
</tr>
</tbody>
</table>

## Dynamic Memory Allocation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Mistakes with Dynamic Memory Allocation</td>
<td>14</td>
</tr>
<tr>
<td>Dynamically Allocating Strings</td>
<td>15</td>
</tr>
<tr>
<td>Dynamically Allocating Arrays</td>
<td>15</td>
</tr>
</tbody>
</table>

## Linked Lists

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Data Types</td>
<td>16</td>
</tr>
<tr>
<td>Operations performed on a (linked) list data type</td>
<td>16</td>
</tr>
</tbody>
</table>

## More Complex Data Structures, e.g. Advanced Linked Lists, Trees, etc.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers to Pointers (to Pointers to ...)</td>
<td>19</td>
</tr>
<tr>
<td>Function Pointers (Pointers to Functions)</td>
<td>19</td>
</tr>
<tr>
<td>Can you assign a function pointer to functions with different number of arguments</td>
<td>20</td>
</tr>
<tr>
<td>Pointers as Function Arguments</td>
<td>21</td>
</tr>
<tr>
<td>Passing a Pointer to a Pointer</td>
<td>22</td>
</tr>
</tbody>
</table>

## Pointer to Pointer and Reference to Pointer

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>23</td>
</tr>
<tr>
<td>Why We Need Them?</td>
<td>23</td>
</tr>
<tr>
<td>Passing arguments by address</td>
<td>23</td>
</tr>
<tr>
<td>Syntax of Pointer to Pointer</td>
<td>23</td>
</tr>
<tr>
<td>Reference to Pointer</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to Pointer PTR-TO-PTR PARAMETER</td>
<td>26</td>
</tr>
<tr>
<td>Reference to Pointer REF-TO-PTR PARAMETER</td>
<td>26</td>
</tr>
<tr>
<td>Preference of one over the other?</td>
<td>27</td>
</tr>
<tr>
<td>Do not Mistake Pointer to Pointer Arguments</td>
<td>27</td>
</tr>
<tr>
<td>Reference to Pointer type (RTTI)</td>
<td>27</td>
</tr>
<tr>
<td>What are other alternatives?</td>
<td>27</td>
</tr>
<tr>
<td>Conclusion</td>
<td>27</td>
</tr>
<tr>
<td>How to Use Arrays of Function Pointers</td>
<td>28</td>
</tr>
<tr>
<td>Is the type of “pointer-to-member-function” different from “pointer-to-function”?</td>
<td>29</td>
</tr>
<tr>
<td>Function Pointers</td>
<td>30</td>
</tr>
<tr>
<td>Assigning a function to a function pointer</td>
<td>30</td>
</tr>
<tr>
<td>Calling a function using a function pointer</td>
<td>31</td>
</tr>
<tr>
<td>When to pass parameters by value, reference, and pointer</td>
<td>31</td>
</tr>
<tr>
<td>Difference between references and pointers</td>
<td>31</td>
</tr>
</tbody>
</table>

## Pointers

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address-of operator (&amp;)</td>
<td>37</td>
</tr>
<tr>
<td>Dereference operator (*)</td>
<td>38</td>
</tr>
</tbody>
</table>
SMART POINTERS

INTRODUCTION

SKILLS FOR USING C++ SMART POINTERS

SUMMARY
C++ Advanced Pointers

To better understand pointers, it sometimes helps to compare a “normal variable” with a pointer.

When a “normal variable” is declared, memory is claimed for that variable. Let’s say you declare an integer variable MYVAR. Four bytes of memory is set aside for that variable. The location in memory is known by the name MYVAR. At the machine level that location has a memory address.

A pointer differs in the way that a pointer is a variable that points to another variable. A pointer holds the memory address of that variable. That variable contains a value. Pointers are also called address variables because they contain the addresses of other variables.

**Example:** We have a piece of memory with a start address of 0x2000. That piece of memory contains a value and is named MYEXAMPLE. (This is in fact a variable). We also have a pointer MYPOINT. In this case, our pointer MYPOINT contains the address 0x2000. This is the address of MYEXAMPLE so we say MYPOINT points to MYEXAMPLE. So there is the pointer and the value pointed to. (You can use a pointer in a certain way to get the value at the address to which the pointer points).

Many novice programmers get pointers and their contents confused. So from now on every pointer starts with the extension ptr_ (for example: ptr_MYPOINT).

**Initialize a pointer**

To declare a pointer you have to put an * in front of its name. A pointer can be typed or un-typed. (A typed pointer points to a particular variable type such as an integer. An un-typed pointer points to any data type).

See the following example of a declaration of a typed pointer and an un-typed pointer:

```cpp
void main()
{
    int * ptr_A;    /* A typed pointer */
    void * ptr_B;   /* A untyped pointer */
}
```

**Note:** A void pointer cannot be directly dereferenced because it is not pointing at a specific type. (Reference and dereference is explained further down.)

Before you can use a pointer in for instance a cout statement, you have to initialize the pointer. The following example will not initialize the pointer:

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

void main()
{
    int *ptr_p;    /* A typed pointer */
    cout << *ptr_p;
}
```

**Note:** Most compilers will give a warning when you try to compile the example, others will not. Visual Studio 2005 will give the following warning: C4700: un-initialized local variable ‘ptr_p’ used. (But the example will/should compile without errors.)

In this example we print the value that ptr_p points to. However, we did not initialize the pointer. In this case the pointer contains a random address or 0. The result of this program is a segmentation fault, some other run-time error or the random address is printed. The meaning of a segmentation fault is that you have used a pointer that points to an invalid address. In most cases, a pointer that is not initialized or a wrong pointer address is the cause of
The next example demonstrates the correct usage of pointers:

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

void main()
{
    int x;
    int *ptr_p;
    x = 5;
    ptr_p = &x;
    cout << *ptr_p;
}
```

**Note:** If you forget to place * (in front of the pointer) in the cout statement, you will print the address of integer x. (Try it).

**Reference and dereference operators**

In the example above we used ampersand sign (&). This sign is called the reference operator. If the reference operator is used you will get the “address of” a variable. In the example above we said: ptr_p = &x;. In words: store the address of the variable x in the pointer ptr_p.

We also used the asterisk sign (*) in the cout statement. This sign is called the dereference operator. If the dereference operator is used you will get the “value pointed by” a pointer. So we said: cout << *ptr_p;. In words: print (or put into the stream) the value pointed by ptr_p. (It will print the contents of integer x.)

**Note:** The asterisk (*) sign in the declaration of the pointer does not mean “value pointed by”, it only means that it is a pointer (it is part of its type compound specifier). It should not be confused with the dereference operator. They are simply two different things represented with the same sign.

So we can say:

- & is the reference operator and can be read as “address of”.
- * is the dereference operator and can be read as “value pointed by”.

**The & Operator — Reference To**

There are several ways to compute a reference to a pointee suitable for storing in a pointer. The simplest way is the & operator. The & operator can go to the left of any variable, and it computes a reference to that variable. The code below uses a pointer and an & to produce the earlier num/numPtr example.
Using pointers to pass values

In the example above we used an integer ‘x’ of which we stored the “address of” into a pointer ‘ptr_p’. But it is also possible to use a pointer to pass a value to a variable.

Take a look at the next example:

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

int main ()
{
    int x;
    int * ptr_p;
    ptr_p = &x;
    *ptr_p = 5;
    cout << x;
    return 0;
}
```

**Note:** You can see that no value is stored in x as in the previous example (x = 5). But the result is the same.

First we store the address of x into the pointer ptr_p. Then we say we want the value 5 stored at the address where the pointer is pointing to (in this case x).

So you can see we can use pointers to pass values to other variables.
Pointers and arrays

When you write
btr[0] = crr;
the expression \texttt{crr} has type "4-element array of char *"); since it is not the operand of a \texttt{sizeof} or \texttt{&} operator, it is converted to type "pointer to char *"); or char **. This matches the type of \texttt{btr[0]}.

However, when you write
\texttt{char** btr[4] = &crr;}
the expression \texttt{crr} is the operand of the \texttt{&} operator; thus, the type of the expression \texttt{&crr} is "pointer to 4-element array of char *"); or char *(*)[4].

The parentheses are necessary because the postfix [] operator has higher precedence than the unary * operator; the expression \texttt{*a[i]} will always be parsed as *(a[i]). Thus,

\begin{verbatim}
T *a[N]; // a is an N-element array of pointer to T
T (*a)[N]; // a is a pointer to an N-element array of T
T *(*a)[N]; // a is a pointer to an N-element array of pointer to T
\end{verbatim}

The same is true for pointers and functions:

\begin{verbatim}
T *f(); // f is a function returning a pointer to T
T *(*f)(); // f is a pointer to a function returning T
T *(*f)(); // f is a pointer to a function returning pointer to T
\end{verbatim}

The C++ language allows pointer addition and subtraction. Let’s take a look at this example:

\begin{verbatim}
char num[10];
char *ptr_toarray = &num[0];
\end{verbatim}

In this example we declare an array with ten elements. Then we say that the pointer \texttt{ptr_toarray} must point at the first element of the array (num[0]).

Now we could do the following (note the round brackets):

\begin{verbatim}
char num[10];
char *ptr_toarray = &num[0];
*(ptr_toarray + 2);
\end{verbatim}

This is the same as num[2]. Or we can do this:

\begin{verbatim}
char num[10];
char *ptr_toarray = &num[0];
ptr_toarray++;
\end{verbatim}

So now the pointer is pointing at the second element: num[1]. (We also could write: ptr_toarray=ptr_toarray+1;)

Take a look at the different things you can do with pointers and arrays, explained by the following example:

\begin{verbatim}
#include<iostream>
using namespace std;
int main ()
{
    int num[5];
    int * ptr_p;
    ptr_p = num; //ptr_p points at first element
    *ptr_p = 1; //Store the value 1
}
\end{verbatim}
ptr_p++;
//Increase to second element
*ptr_p = 2;
//Store the value 2

ptr_p = &num[2];
//Get address of third element
*ptr_p = 3;
//Store the value 3

ptr_p = num + 3;
//Goto element 4
*ptr_p = 4;
//Store the value 4

ptr_p = num;
//Point at first element
*(ptr_p+4) = 5;
//First goto element 5 and then store 5

//Now print value of each element
for (int i=0; i<5; i++)
    cout << num[i] << '\n';
return 0;
}

**Operator precedence**

Both the increase (++) and decrease (--) operators have greater operator precedence than the dereference operator (*), but both have a special behavior when used as suffix.

Let’s take a look at an example:

```
*ptr_p++;
```

Because ++ has greater precedence than *, this expression is equivalent to *(ptr_p++). Therefore, what it does is to increase the value of ptr_p (so it now points to the next element). You might think this but because ++ is used as post-fix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with the following example:

```
(*ptr_p)++
```

Here the value pointed by ptr_p is increased by one. The value of the pointer itself (p_ptr) is not modified. So the only thing that is modified is what it is being pointed to by the pointer.

Another example:

```
*ptr_p++ = *ptr_a++;
```

The increase operator (++) has a higher precedence than *. Because we use the increase operators as post-fix (instead of prefix) first the value of *ptr_a is assigned to *ptr_p. After this is done both are increased by one.

So always remember the operator precedence. Also use parentheses () in order to avoid unexpected results and confusion when reading the code.

**Pointer to Pointer**

It is allowed in C++ to use a pointer to point at a pointer. The last pointer may even point at data or even point at another pointer. In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations. Take a look at the example:
```cpp
#include<iostream>
using namespace std;

int main ()
{
    int a;
    int * ptr_b;
    int ** ptr_c;
    a = 1;
    ptr_b = &a;
    ptr_c = &ptr_b;  //Get address of ptr_b
    //print value of a (output is 1)
    cout << a << 'n';
    //print value where pointer ptr_b points to. (output is 1)
    cout << *ptr_b << 'n';
    //print address of ptr_b, that we got during the
    //ptr_c = &ptr_b; statement. (output is for example 0021F7A4)
    cout << *ptr_c << 'n';
    //print value where ptr_c points to. Same as value of ptr_b
    //Same as value of ptr_b and integer a. (output is 1)
    cout << **ptr_c << 'n';
    return 0;
}
```

**Null pointer**

A null pointer is a regular pointer. It only indicates that it is not pointing to a valid memory address or reference. For instance:

```cpp
int * ptr_p;
ptr_p = 0;
```

Don’t confuse null pointers with void pointers. Null pointers point to “nowhere”. Void pointers are special types of pointers that can point to anything (it has no type).

**Pointers to function**

The C and C++ language are a “call by value” language, which means that the called function is given a copy of its arguments, and doesn’t know their addresses. (For example: myfunction(x) call is given, the value of x is passed, not its address).

This makes it impossible to change the value of x from the inside of the function (myfunction).

**Note:** With an array this is not a problem. If x is an array (char x[10]) then x is an address anyway.

If we have a function `int f()` then we may simply (!) write:

```cpp
pf = &f;
```

For compiler prototyping to fully work it is better to have full function prototypes for the function and the pointer to a function:

```cpp
int f(int);
```
int (*pf)(int) = &f;

Now f() returns an int and takes one int as a parameter.
You can do things like:
ans = f(5);
ans = pf(5);
which are equivalent.

Pointer variables can hold the address of any kind of data, including the address of where to find functions.
Pointers to functions are declared according to the argument type and return type of the function that they point to.
For example:

    int i;  // i is an int
    int *iptr;  // iptr is a pointer to an int
    int ifcn(double);  // ifcn is a function that returns an int (and takes a double.)
    int *iptrfcn(char);  // iptrfcn is a function that returns a pointer to an int (and takes a char)
    int *(*ifcnptr)(double);  // ifcnptr is a pointer to (a function that returns an int and takes a double)
    int *(*iptrfcnptr)(node*);  // iptrfcnptr is a pointer to (a function that returns an int * and takes a node *)

In order to get the address of a function for assignment to a function pointer variable, simply use the name of the function without parentheses:

    ifcnptr = ifcn;  // The ifcnptr pointer variable now holds the address of the ifcn function.

Note carefully that the types of the function pointer and the types of the function it points to must agree.
In use, the function pointers can be used identically to the functions they point to:

    int answer = ifcnptr(3.14159);  // same as answer = ifcn(3.14159);

In practice, function pointers are often used to allow one function to call any of a number of different functions, without specifying ahead of time which function(s) will be called, provided they all have the correct argument types.
For example, the following is a prototype for a function that will find the value of X that yields the largest value of f(X) over the range from start to end. The function to be optimized is passed in as the first argument:

    double findMax(double (*fptr)(double), double start, double end);

To call this function we can use any function that takes a double and returns a double, so given:

    double sinePlusCosine(double x);
    double sineMinusCosine(double x);

Then we can find the maximum value of (sin(x) + cosine(x)) and (sin(x) - cosine(x)) over the range from 0 to 2 pi by passing the address of the function as the function name without (parentheses):

    double xMaxPlus = findMax(sinePlusCosine, 0.0, 2.0 * 3.14159);
    double xMaxMinus = findMax(sineMinusCosine, 0.0, 2.0 * 3.14159);

And then report the results as:

    printf("The maximum value of sin( x ) + cos( x ) over the range 0 to 2 pi is \
            " %f, at an X value of %f\n", sinePlusCosine(xMaxPlus), xMaxPlus);
Take a look at the following example, which will illustrate the problem:

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

void swapping(int c, int d)
{
    int tmp;
    tmp = c;
    c = d;
    d = tmp;
    cout << "In function:\n" << c << \n << d << \n;
}

void main()
{
    int a,b;
    a=5;
    b=10;
    cout << "Before:\n" << a << \n << b << \n;
    swapping(a,b);
    cout << "After:\n" << a << \n << b << \n;
}
```

In the example the values of the parameters are swapped in the function `swapping`. But when the function returns nothing has happened. The result is that the values are not swapped. (Try it!).

Pointers can be used to get around the “call by value” restriction. In the next example we will use pointers to correct the problem:

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

void swapping(int *ptr_c, int *ptr_d)
{
    int tmp;
    tmp = *ptr_c;
    *ptr_c = *ptr_d;
    *ptr_d = tmp;
    cout << "In function:\n" << *ptr_c << \n << *ptr_d << \n;
}

void main()
{
    int a,b;
    a=5;
    b=10;
    cout << "Before:\n" << a << \n << b << \n;
    swapping(&a,&b);
    cout << "After:\n" << a << \n << b << \n;
}
```

**Note:** Don’t forget to replace “swapping(a,b);” for “swapping(&a,&b);”.

That is all for this tutorial.
**Pointer Operators**

There are two important pointer operators such as ‘*’ and ‘&’. The ‘&’ is a unary operator. The unary operator returns the address of the memory where a variable is located. For example,

```c
int x*;
int c;
x=&c;
```

variable x is the pointer of the type integer and it points to location of the variable c. When the statement

```c
x=&c;
```

is executed, ‘&’ operator returns the memory address of the variable c and as a result x will point to the memory location of variable c.

The ‘*’ operator is called the indirection operator. It returns the contents of the memory location pointed to. The indirection operator is also called deference operator. For example,

```c
int x*;
int c=100;
int p;
x=&c;
p=*x;
```

variable x is the pointer of integer type. It points to the address of the location of the variable c. The pointer x will contain the contents of the memory location of variable c. It will contain value 100. When statement

```c
p=*x;
```

is executed, ‘*’ operator returns the content of the pointer x and variable p will contain value 100 as the pointer x contain value 100 at its memory location. Here is a program which illustrates the working of pointers.

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

int main ()
{
    int *x;
```
int c=200;
int p;
x=&c;
p=*x;
cout << " The address of the memory location of x : " << x << endl;
cout << " The contents of the pointer x : " << *x << endl;
cout << " The contents of the variable p : " << p << endl;
return(0);
}

The result of the program is:

In the program variable x is the pointer of integer type. The statement

    x=&c;

points variable x to the memory location of variable c. The statement

    p=*x;

makes the contents of the variable p same as the contents of the variable c as x is pointing to the memory location of c. The statement

    cout << " The address of the memory location of x : " << x << endl;

prints the memory address of variable x which it is pointing to. It prints the hexadecimal address 0012FF78. This address will be different when the program is run on different computers. The statement

    cout << " The contents of the pointer x : " << *x << endl;

prints the contents of memory location of the variable x which it is pointing to. The contents are same as the variable c which has value 200. The statement
has the same output 200 as the statement above. The contents of variable p is same as the contents of the pointer x.
Dynamic Memory Allocation

There are a number of valuable functions for dynamically allocating memory (from the heap) as programs run. These functions all require include <stdlib.h>

- void *malloc(size_t size);
  - Allocates size bytes of memory, does not clear it, and returns a void pointer to the address where the memory is located.
  - Void pointers are generic pointers that can point to anything. They must normally be type cast to a real pointer type, either explicitly or implicitly, before they can be used.
  - malloc returns a NULL pointer if the dynamic memory request cannot be granted. The return value should always be checked before it is used.

- void *calloc(size_t nmemb, size_t size);
  - Similar to malloc, calloc allocates enough memory for an array of nmemb elements, where each element is size bytes big. calloc DOES zero out the memory before returning.

- void *realloc(void *ptr, size_t size);
  - realloc reallocates memory previously allocated with malloc or calloc. Existing data is unchanged, up to the smaller of the old size or the new size.

When the memory is no longer needed, it should be returned to the memory heap.

- void free(void *ptr);
  - Frees up (gives back) the memory pointed to by ptr, which must have previously been allocated with one of the dynamic memory allocation methods described above.
  - FREE DOES NOT CHANGE THE VALUE OF THE PTR VARIABLE.
  - ALWAYS RESET THE POINTER VARIABLE TO NULL AFTER RETURNING MEMORY!

Common Mistakes with Dynamic Memory Allocation

There are a number of errors that occur commonly when using dynamic memory:

1. Dangling Pointers:
   - If dynamic memory is freed up using free, but the pointer variable is not reset back to NULL, then the pointer still contains the address of where the dynamic memory used to be. Using (following) this pointer later can have a wide variety of consequences, depending on what if anything is done with that block of memory in the meantime.
   - The problem can be especially difficult to find if multiple pointers point to the same place. Even though one of the pointers may have been reset to NULL, the other pointers could be left dangling.

2. Memory Leaks:
   - If dynamic memory is continuously allocated without ever being freed back up again, the system will eventually run out of memory, which will result in either a stack overflow error or a failure of the dynamic memory allocation calls.
   - This type of error occurs most commonly when dynamic memory is allocated in a loop.
   - Always make sure to free up dynamic memory when it is no longer being used.
   - Always check the return value of malloc, calloc, or realloc to verify that it did not return NULL.
   - Memory leaks can be extremely hard to find, because they often don't result in program failures until the program runs for a sufficiently long time, so unless the program tests run long enough, the problems won't be detected.

3. Double deallocation
If an attempt is made to deallocate memory pointed to by a dangling pointer, it can cause all sorts of problems, particularly if that memory has since been reallocated for other purposes.

4. Use of uninitialized pointers
   - As covered earlier, any variable that is not initialized starts out with a random unknown value, which will generally be different from one run of the program to another. If the variable in question is a pointer, then the problem can be especially difficult when the uninitialized pointer is used later on in the program.
     - In a best case, the pointer will point to inaccessible memory, and a "segmentation fault" will occur, stopping the program.
     - If the bad pointer points to a valid location within your data or worse your code, then any number of problems can occur, possibly at some later time in the program.

Dynamically Allocating Strings

One common use of dynamic memory allocation is for the storage of strings, in an array that is just big enough to hold the necessary data:

```c
char line[200];
char *name = NULL;

printf("Please enter your name: ");
fgets(buffer, 200, stdin);

name = (char *) malloc(strlen(buffer) + 1);  // +1 to hold the null byte

if(!name) {
    fprintf(stderr, "Error - malloc failed!\n");
    exit(-1);
}

strcpy(name, buffer);
```

Dynamically Allocating Arrays

Another very common use of dynamic memory allocation is for allocating arrays. (Particularly in the days before C99 allowed you to allocate arrays after the program was already executing.)

```c
int nData = -1;
double *data = NULL;

do {
    printf("How many data items would you like to store? ");
    scanf("%d", &nData);
} while(nData < 1);

data = (double *) malloc(nData * sizeof(double));  // Uninitialized, random values
// data = (double *) calloc(nData, sizeof(double));  // Cleared to all zeros

if(!data) {
    fprintf(stderr, "Error - malloc failed!\n");
    exit(-1);
}
```
**Linked Lists**

**Abstract Data Types**

An Abstract Data Type, ADT, is defined in terms of the operations that can be performed on or with the data type, independent of any specific implementation of the data type. A linked list is one of the simplest and most common ADTs used in computer science, consisting of a series of nodes (links) each of which points to the next node in the list, like links in a chain.

**Operations Performed on a (Linked) List Data Type:**

**Define a list node**

A list node must contain at a minimum a pointer to the next node in the list. In order to be effective it must also contain some data, and depending on the specific kind of linked list, it may or may not contain additional pointers.

In C a list node is typically defined as a struct, e.g.:

```c
struct StudentLink {
    int nClasses;
    double gpa;
    char *name; // will point to a dynamically allocated string
    struct StudentLink * next;
};
```

At first glance it appears that the above struct contains a circular reference, with one StudentLink contained within another. That is not the case, however, because next is a pointer, not a full struct. (Note that all pointers are the same size, regardless of what they point to, so there is no question here about how much space to allocate for the pointer or for the struct.)

**Create an empty list**

A null pointer of the appropriate type is effectively an empty list:

```c
struct StudentLink *head = NULL;
```

**Create a list node.**

List nodes can be declared as ordinary auto variables, but more commonly they are allocated dynamically:

```c
// Possible but not common
// struct StudentLink newGuy = { 3, 4.0, "George P. Burdell", NULL };

// Much more common
struct StudentLink * newStudent = NULL; /
newStudent = ( struct newStudent * ) malloc( sizeof( struct newStudent ) );
if( newStudent ) {
    newStudent->nClasses = 0;
    newStudent->gpa = 0.0;
    newStudent->name = ( char * ) malloc( strlen( buffer ) + 1 );
    if( !newStudent->name ) {
        fprintf( stderr, "Error - malloc failed!\n";
        exit( -1 );
    }
```
Insert a new data item (node)

Adding a link into an existing list requires setting the next pointer in the new link to point into the list, and then setting a pointer from the list to point to the new link. There are two basic ways to do this, depending on circumstances:

1. **Beginning of the list** - Inserting a new link at the beginning of a list is easiest, because you just have to work with the new link and the list head:

   ```c
   newStudent->next = head;
   
   // Remember newStudent is a pointer, not a struct
   head = newStudent; // Pointer assigned to pointer
   
   // head now points to newStudent, which then points to the rest of the list.
   ```

2. **Maintaining an ordered list** - Inserting a new link into the middle of an existing list is harder, because you generally need it to go to a specific spot, which means you first need to find the node in the list just before the spot where the new node needs to go. Generally we write a function to do this:

   ```c
   struct StudentLink *previous = findPrevious( head, newStudent->name );
   
   if( !previous ) {
      // There is a link that belongs before the new student
      newStudent->next = previous->next;
      previous->next = newStudent;
   } else {
      // This student belongs at the beginning of the list
      newStudent->next = head; // Same code as above
      head = newStudent;
   }
   ```

Remove a data item (node)

Find a data item (node)

Determine the length of a list

Destroy a list
More Complex Data Structures, e.g. Advanced Linked Lists, Trees, etc.

- Circularly Linked Lists
  - Normally the last node on a linked list has the "next" field set to NULL.
  - If it points instead to the first node on the list, then the list is circular.
  - Circular linked lists have some advantages, but are also more complicated to maintain and use.
- Doubly Linked Lists
  - Suppose we add a second pointer to each node of a linked list, pointing back to the previous node in the list.
  - Now we can traverse the list back and forth in either direction, but adding and removing nodes is more complicated.
  - Doubly linked lists can also be circular.
- Cross Linked Lists
  - Suppose we have two (or more) links in each node, which connect the list based on different criteria:
    - For example, one set of links may connect the nodes in the order by size, and the other by age.
    - Alternatively one set of links could connect all the students in a class, while another set links all the classes a student is taking.
- Binary (Search) Trees
  - Each node has two pointers, pointing to two "children".
  - Typically the first points to nodes having a "smaller" value than the current node, and the other to nodes having "larger" values.
  - Greatly speeds up finding data, at the cost of increasing complexity.
    - (Requires rebalancing the tree occasionally for best performance.)
- N-ary Trees
  - Extend the tree concept to nodes with more than 2 children.
  - For example, a node in a dictionary tree could have 26 children, one for each legal value for the next character in the word.
- Graphs (Networks)
  - Nodes connected by arbitrary arcs
  - Consider a bus map, with routes connecting different bus stops.
  - Arcs may be directed or undirected.
  - Cycles may or may not be allowed.
- Complex combinations of different struct types.
  - Structs can contain pointers to different kinds of structs
  - The possibilities are endless!
**Pointers to Pointers (to Pointers to . . .)**

- Pointer variables can hold the address of any kind of data, including the address of where to find other pointer variables.
- With multiple levels of indirection, pointer variables are declared according to the basic type one eventually gets to if you follow the daisy-chain of pointers long enough. For example:

  ```c
  int i, *iptr = &i, **iptrptr = &iptr, ***iptr3 = &iptrptr;
  ```

- In this example:
  - `i` is an int.
  - `iptr` is a pointer to an int, holding the address of an int.
  - `iptrptr` is a pointer to a pointer to an int, holding the address of an int pointer.
  - `iptr3` is a pointer to (a pointer to a pointer to an int), holding the address of an int **

- When using indirection with these types of pointers, just keep asking "what kind of data is this" until you get to a basic type. So for example:
  - `i` is an int.
  - `iptr` is an int *, so *iptr is an int.
  - `iptrptr` is an int **, so *iptrptr is an int *, and **iptrptr is an int.
  - `iptr3` is an int ***, so *iptr3 is an int **, **iptr3 is an int *, and ***iptr3 is an int.

- Pointers to pointers allows us to pass pointer variables to functions via the pointer/address passing mechanism, so the function can change the pointer located back in main:

  ```c
  void movePointerToNextNode( struct node ** pointer ) {
  
  node *p = *pointer;
  p = p->next;
  *pointer = p;
  return;
  }
  ```

  And then in main:

  ```c
  node *current = NULL;
  
  // code omitted that makes current point into a list
  movePointerToNextNode( &current ); // Pointer variable passed by address
  ```

- Note that because of the interchangability of pointers and arrays, arrays of pointers can often be considered equivalent to double indirection.
  - For example, the following two function prototypes are equivalent:

    ```c
    int main( int argc,char * argv[ ] );
    int main( int argc, char **argv );
    ```

**Function Pointers (Pointers to Functions)**

- Pointer variables can hold the address of any kind of data, including the address of where to find functions.
- Pointers to functions are declared according to the argument type and return type of the function that they point to.
- For example:

  ```c
  int i; // i is an int
  int *iptr; // iptr is a pointer to an int
  ```
• int ifcn( double ); // ifcn is a function that returns an int (and takes a double.)
• int *iptrfcn( char ); // iptrfcn is a function that returns a pointer to an int (and takes a char)
• int (*ifcnptr) ( double ); // ifcnptr is a pointer to (a function that returns an int and takes a double)
• int *(iptrfcnptr) ( node * ); // iptrfcnptr is a pointer to (a function that returns an int * and takes a node *)

- In order to get the address of a function for assignment to a function pointer variable, simply use the name of the function without (parentheses):
  - ifcnptr = ifcn; // The ifcnptr pointer variable now holds the address of the ifcn function.
  - Note carefully that the types of the function pointer and the types of the function it points to must agree.

- In use, the function pointers can be used identically to the functions they point to:
  - int answer = ifcnptr( 3.14159 ); // same as answer = ifcn( 3.14159 );

- In practice, function pointers are often used to allow one function to call any of a number of different functions, without specifying ahead of time which function(s) will be called, provided they all have the correct argument types.
  - For example, the following is a prototype for a function that will find the value of X that yields the largest value of f( X ) over the range from start to end. The function to be optimized is passed in as the first argument:
    
    ```
    double findMax( double ( *fptr ) ( double ), double start, double end );
    ```
  
  - To call this function we can use any function that takes a double and returns a double, so given:
    ```
    double sinePlusCosine( double x );
    double sineMinusCosine( double x );
    ```
  - Then we can find the maximum value of ( sine( x ) + cosine( x ) ) and ( sine( x ) - cosine( x ) ) over the range from 0 to 2 pi by passing the address of the function as the function name without (parentheses).

    ```
    double xMaxPlus = findMax( sinePlusCosine, 0.0, 2.0 * 3.14159 );
    double xMaxMinus = findMax( sineMinusCosine, 0.0, 2.0 * 3.14159 );
    ```

And then report the results as:

    ```
    printf( "The maximum value of sin( x ) + cos( x ) over the range 0 to 2 pi is " 
    " %f, at an X value of %f\n", sinePlusCosine( xMaxPlus ), xMaxPlus );
    ```

**Can you assign a function pointer to functions with different number of arguments**

*NO.*

Function pointers are specific to *one* function signature. This is entirely logical: how would you invoke such a function? (Yes, C allows invoking functions without specifying in their declaration how many parameters the function has, but this doesn’t work in C++ since it subverts the type system.)

Unless

1. C+11 OR
2. `std::function`

    ```
    boost::function and boost::bind are equivalent, although slightly more restrictive.
    ```
Pointers as Function Arguments

One of the best things about pointers is that they allow functions to alter variables outside of their own scope. By passing a pointer to a function you can allow that function to read and write to the data stored in that variable. Say you want to write a function that swaps the values of two variables. Without pointers this would be practically impossible, here’s how you do it with pointers:

Example 5-2. swap_ints.c
#include <stdio.h>

int swap_ints(int *first_number, int *second_number);

int main()
{
    int a = 4, b = 7;
    printf("pre-swap values are: a == %d, b == %d\n", a, b);
    swap_ints(&a, &b);
    printf("post-swap values are: a == %d, b == %d\n", a, b);
    return 0;
}

int swap_ints(int *first_number, int *second_number)
{
    int temp;
    /* temp = "what is pointed to by" first_number; etc... */
    temp = *first_number;
    *first_number = *second_number;
    *second_number = temp;
    return 0;
}

As you can see, the function declaration of swap_ints() tells GCC to expect two pointers (address of variables). Also, the address-of operator (&) is used to pass the address of the two variables rather than their values. swap_ints() then reads
Passing a Pointer to a Pointer

A regular use of the double pointer is to pass it as argument. This can easily be done as follows:

```c
void ShowValue(int **value)
{
}
```

A double-pointer is primarily a pointer. It is just a pointer that points to another pointer. If it's passed as argument, you can display its value using `Write()` or `WriteLine()` as done above. Here is an example:

```c
void ShowValue(int **value)
{
    Console::WriteLine("Value = {0}", **value);
}
```

When calling a function that takes a double-pointer, if the variable passed was declared as a (single) pointer, type an ampersand to the left of the argument. Here is an example:

```c
using namespace System;

void ShowValue(int **value)
{
    Console::WriteLine("Value = {0}", **value);
}

int main()
{
    int *vl = new int(428);
    ShowValue(&vl);
    return 0;
}
```

This would produce:

```
Value = 428
Press any key to continue...
```
 Pointer to Pointer and Reference to Pointer

Introduction

This article explains the reason behind using pointer-to-pointer and reference-to-pointer to modify a pointer passed to a function, so as to understand their usage better. For brevity, I use the terms, ptr-to-ptr and ref-to-ptr to represent them respectively. In this article, I'm not going to discuss how to use ptr-to-ptr as a 2 dimensional array or array of pointers. Please note we can use ptr-to-ptr in both C and C++ but we can use ref-to-ptr only in C++.

Why We Need Them?

When we use "pass by pointer" to pass a pointer to a function, only a copy of the pointer is passed to the function. We can say "pass by pointer" is passing a pointer by value. In most cases, this does not present a problem. But problem comes when you modify the pointer inside the function. Instead of modifying the variable, you are only modifying a copy of the pointer and the original pointer remains unmodified, that is, it still points to the old variable. The code below demonstrates this behavior.

Passing arguments by address

```c
//global variable
int g_One=1;

//function prototype
void func(int* pInt);

int main()
{
    int nvar=2;
    int* pvar = &nvar;

    func(pvar);  //call function with pointer
    std::cout<< *pvar  //Will still show 2
    return 0;
}

void func(int* pInt)  //receive pointer
{
    pInt = &g_One;  //address of int into pointer of int
}
```

Syntax of Pointer to Pointer

This is how you called the function with ptr-to-ptr parameter.
void func(int** ppInt)
{
    //Modify the pointer, ppInt points to
    *ppInt = &g_One;

    //You can also allocate memory, depending on your requirements
    *ppInt = new int;

    //Modify the variable, *ppInt points to
    **ppInt = 3;
}

int main()
{
    int nvar=2;
    int* pvar = &nvar;
    func(&pvar);
    ....
    return 0;
}

Let me summarise what all those dereferencing are,

- `ppInt` is the ptr-to-ptr. **We will never modify this because if we do, we'll lose our grip on the address of the pointer it is pointing to.**
- `*ppInt` is the pointed pointer. If we modify this, we are modifying the contents of the pointed pointer, which is an address and in the above example, `pvar`. In other words, we are effectively modifying what `pvar` points to.
- `**ppInt` is the dereferenced twice variable which is what `pvar` points to.
Reference to Pointer

Passing an Argument as a Reference to a Pointer

`ref-to.ptr` parameter

```cpp
void func(int*& rpInt)
{
    //Modify what rpInt and pvar is pointing to, to g_One
    rpInt = &g_One;

    //You can also allocate memory, depending on your requirements
    rpInt = new int;

    //Modify the variable rpInt points to
    *rpInt = 3;
}

int main()
{
    int nvar = 2;
    int* pvar = &nvar;
    func(pvar);
    ....
    return 0;
}
```

You may wonder whether, in the above `func()`, the parameter `rpInt` is pointer to reference. Just take my word for it that it is called `ref-to.ptr` and it is `ref-to.ptr`.

Let me once again summarize what all those dereferencing are,

- `rpInt` is the reference for the pointer, `pvar` in the above example.
- `*rpInt` dereferences what `pvar` point to, so you get the variable the pointer, `pvar` is pointing to.
**Pointer to Pointer**  
ptr-to-ptr parameter

```c
void func(int **ppInt)
{
    //Modify the pointer, ppInt points to
    *ppInt = &g_One;

    //You can also allocate memory, depending on your requirements
    *ppInt = new int;

    //Modify the variable, *ppInt points to
    **ppInt = 3;
}

int main()
{
    int nvar = 2;
    int* pvar = &nvar;
    func(&pvar);
    ....
    return 0;
}
```

**Reference to Pointer**  
ref-to-ptr parameter

```c
void func(int *&rpInt)
{
    //Modify what rpInt and pvar is pointing to, to g_One
    rpInt = &g_One;

    //You can also allocate memory, depending on your requirements
    rpInt = new int;

    //Modify the variable rpInt points to
    *rpInt = 3;
}

int main()
{
    int nvar = 2;
    int* pvar = &nvar;
    func(pvar);
    ....
    return 0;
}
```
Preference of one over the other?

Now we have seen the syntax of ptr-to-ptr and ref-to-ptr. Are there any advantages of one over the other? I am afraid, no. The usage of one of both, for some programmers are just personal preferences. Some who use ref-to-ptr say the syntax is "cleaner" while some who use ptr-to-ptr, say ptr-to-ptr syntax makes it clearer to those reading what you are doing.

Do not Mistake Pointer to Pointer Arguments

Do not mistake every ptr-to-ptr arguments as purely ptr-to-ptr. An example would be when some write `int main(int argc, char *argv[])` as `int main(int argc, char **argv)` where `**argv` is actually an array of pointers. Be sure to check out the library documentation first!

Reference to Pointer type (RTTI)

You cannot use RTTI to find out the type of ref-to-ptr. As `typeid()` does not support reference types.

```cpp
void test(int*& rpInt)
{
    std::cout << "type of *\&rpInt: " << typeid(rpInt).name()
    << std::endl; // will show int *
}
```

What are other alternatives?

If you find that the ptr-to-ptr and ref-to-ptr syntax are rather hard to understand, you can just use the "return the pointer" method.

```cpp
ClassA* func()
{
    ClassA* p = new ClassA();
    // do my things with p
    // ...
    return p;
}
```

Conclusion

You may ask if you would ever use ptr-to-ptr and ref-to-ptr in your projects and if it is necessary to know about them. Well, as developers, we use libraries and technologies developed by others. One example would be COM uses ptr-to-ptr to return an interface pointer using `CoCreateInstance()` and `IUnknown::QueryInterface()`. Up to some point in your developer career, you are definitely going to come across them. It is good to know them.
How to Use Arrays of Function Pointers?

Operating with arrays of function pointers is very interesting. This offers the possibility to select a function using an index. The syntax appears difficult, which frequently leads to confusion. Below you find two ways of how to define and use an array of function pointers in C and C++. The first way uses a typedef, the second way directly defines the array. It's up to you which way you prefer.

// type-definition: 'pt2Member' now can be used as type
typedef int (TMyClass::*pt2Member)(float, char, char);

// illustrate how to work with an array of member function pointers
void Array_Of_Member_Function_Pointers()
{
    cout << endl << "Executing 'Array_Of_Member_Function_Pointers'" << endl;

    // define arrays and ini each element to NULL, <funcArr1> and <funcArr2> are
    // arrays with 10 pointers to member functions which return an int and take
    // a float and two char

    // first way using the typedef
    pt2Member funcArr1[10] = {NULL};

    // 2nd way of directly defining the array
    int (TMyClass::*funcArr2[10])(float, char, char) = {NULL};

    // assign the function's address - 'DoIt' and 'DoMore' are suitable member
    // functions of class TMyClass like defined above in 2.1-4
    funcArr1[0] = funcArr2[0] = &TMyClass::DoIt;
    funcArr1[1] = funcArr2[0] = &TMyClass::DoMore;
    /* more assignments */

    // calling a function using an index to address the member function pointer
    // note: an instance of TMyClass is needed to call the member functions
    TMyClass instance;
    cout << (instance.*funcArr1[1])(12, 'a', 'b') << endl;
    cout << (instance.*funcArr1[0])(12, 'a', 'b') << endl;
    cout << (instance.*funcArr2[0])(34, 'a', 'b') << endl;
    cout << (instance.*funcArr2[0])(89, 'a', 'b') << endl;
}
Is the type of “pointer-to-member-function” different from “pointer-to-function”?

Yep.

Consider the following function:

```c++
int f(char a, float b);
```

The type of this function is different depending on whether it is an ordinary function or a non-static member function of some class:

- Its type is “int (*)(char, float)” if an ordinary function
- Its type is “int (Fred::*)(char, float)” if a non-static member function of class Fred

Note: if it’s a static member function of class Fred, its type is the same as if it were an ordinary function: “int (*)(char, float)”.

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

int main()
{
    int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    cout << "\n Number    :  " << Number;
    cout << "\n&Number    :  " << &Number;
    cout << "\n&number[0] :  " << &number[0] << endl;

    return 0;
}
```

This would produce:

```
Number    :  1245020
&Number    :  1245020
&number[0] :  1245020
```

This demonstrates that Number, &Number, and &number[0] have the same value. As we learned with pointers, the use of the ampersand "&" allows us to get the address of a variable. Therefore, &Number gives us the address of the array variable. Furthermore, since &Number and &number[0] have the same value, and seeing that all three (Number, &Number, and &number[0]) have the same value, this demonstrates that the name of the variable in fact carries, or holds, or represents, the address of the first value of the array.
Function Pointers
By Alex, on August 8th, 2007

Function pointers are an advanced topic, and this section can be safely skipped or skimmed by those only looking for C++ basics.

In the lesson on pointers, you learned that a pointer is a variable that holds the address of another variable. Function pointers are similar, except that instead of pointing to variables, they point to functions!

Consider the case of an array:

```cpp
int nArray[10];
```

As you now know, nArray is actually a constant pointer to a 10 element array. When we dereference the pointer (either by `*nArray` or `nArray[nIndex]`), the appropriate array element is returned.

Now consider the following function:

```cpp
int foo();
```

If you guessed that foo is actually a constant pointer to a function, you are correct. When a function is called (via the () operator), the function pointer is dereferenced, and execution branches to the function.

Just like it is possible to declare a non-constant pointer to a variable, it’s also possible to declare a non-constant pointer to a function. The syntax for doing so is one of the ugliest things you will ever see:

```cpp
// pFoo is a pointer to a function that takes no arguments and returns an integer
int (*pFoo)();
```

The parenthesis around `*pFoo` are necessary for precedence reasons, as `int *pFoo()` would be interpreted as a function named pFoo that takes no parameters and returns a pointer to an integer.

In the above snippet, pFoo is a pointer to a function that has no parameters and returns an integer. pFoo can “point” to any function that matches this signature.

Assigning a function to a function pointer

There are two primary things that you can do with a pointer to a function.

1) assign a function to it:

```cpp
int foo(){}
int goo(){}

int main()
{
    int (*pFoo)() = foo; // pFoo points to function foo()
pFoo = goo;          // pFoo now points to function goo()
    return 0;
}
```

One common mistake is to do this:

```cpp
pFoo = goo();
```

This would actually assign the return value from a call to function goo() to pFoo, which isn’t what we want. We want pFoo to be assigned to function goo, not the return value from goo(). So no parenthesis are needed.

Note that the signature (parameters and return value) of the function pointer must match the signature of the
function. Here is an example of this:

```c
// function prototypes
int foo();
double goo();
int hoo(int nX);

// function pointer assignments
int (*pFcn1)() = foo; // okay
int (*pFcn2)() = goo; // wrong -- return types don't match!
double (*pFcn3)() = goo; // okay

pFcn1 = hoo; // wrong -- pFcn1 has no parameters, but hoo() does
int (*pFcn3)(int) = hoo; // okay
```

**Calling a function using a function pointer**

2) call the function.

A) via explicit dereference:

```c
int foo(int nX)
{
}

int (*pFoo)(int) = foo; // assign pFoo to foo()

(*pFoo)(nValue); // call function foo(nValue) through pFoo.
```

B) via implicit dereference:

```c
int foo(int nX)
{
}

int (*pFoo)(int) = foo; // assign pFoo to foo()

pFoo(nValue); // call function foo(nValue) through pFoo.
```

As you can see, the implicit dereference method looks just like a normal function call — which is what you’d expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

**When to pass parameters by value, reference, and pointer**

**Difference between references and pointers**

A pointer variable is a variable which holds the memory address of some other variable. That "target" variable may be named or unnamed.

For example:
int i;
int* pInt = &i;  // pInt "points to" i
int* pInt2 = new int;  // pInt2 "points to" an unnamed int.

A reference variable is a variable which "refers to" another named or unnamed variable. For example:

```cpp
void foo( const std::string& str ) {}
std::string s1;
std::string& s1ref = s1;   // s1ref "refers" to s1
// Here, we construct an unnamed, temporary string object to call foo.
// foo's "str" parameter now "refers to" this unnamed object.
foo( std::string( "Hello World" ) );
```

There are three critical attributes of pointers that differentiate them from references.

1. You use pointer syntax to access the values "pointed to" by the pointer.
2. You can redirect the pointer to point it to a different "target" variable.
3. You can make a pointer point to nothing (ie, NULL pointer).

Examples:

```cpp
int i, j;
int* pInt = &i;  // pInt "points to" i
*pInt = 42;  // This assigns the variable pointed to by pInt to 42
// So in other words, since pInt points to i, i now has
// the value 42.
pInt = &j;   // This makes pInt now point to j instead of i.
pInt = NULL; // This makes pInt point to nothing.
```

Notice how use of an asterisk prior to the pointer variable accesses the value being pointed to. This is called the dereference operator, a somewhat unfortunate name given that the language also supports references and the dereference operator has nothing to do with references.

Now to references. References have a couple of key characteristics that differentiate them from pointers:

1. References must be initialized at the point of instantiation.
2. References must ALWAYS "refer to" a named or unnamed variable (that is, you cannot have a reference variable that refers to nothing, the equivalent of a NULL pointer).
3. Once a reference is set to refer to a particular variable, you cannot, during its lifetime, "rebind" it to refer to a different variable.
4. You use normal "value" syntax to access the value being referred to.

Examples:

```cpp
int i = 20, j = 10;
int& iref = i;    // Instantiate iref and make it refer to i
iref = 42;        // Changes the value of i to 42
iref = j;         // Changes the value of i to 10 (the value of j)
```
iref = NULL; // Changes the value of i to 0.

So it seems like references are more limiting than pointers in that two of the three characteristics of pointers are not available with references. But in fact, these limitations tend to make programming easier.

First of all, when writing generic template code, you can't easily write a single template function that operates on values, references, and pointers, because to access the value "pointed to" by a pointer requires an asterisk whereas accessing a normal value does not require an asterisk. Accessing the value of a reference works the same way as accessing a normal value -- no asterisk needed. So writing templates that can handle values and references is easy. Here's a real-world example:

```cpp
template< typename T >
void my_swap( T& t1, T& t2 )
{
    T tmp( t1 );
    t1 = t2;
    t2 = tmp;
}
```

The above function works great in these cases:

```cpp
int i = 42, j = 10;
int& iref = i, jref = j;
my_swap( i, j ); // Sets i = 10 and j = 42
my_swap( iref, jref ); // Swaps i and j right back
```

But, if you are expecting with the following code that i will be set to 10 and j to 42, then this doesn't do what you want:

```cpp
int i = 42, j = 10;
int* pi = &i, *pj = &j;
my_swap( pi, pj ); // sets pi = &j and pj = &i
```

Why? Because you need to dereference the pointers to get to the values being pointed to, and the template function above does not have a single asterisk in it.

If you wanted this to work correctly, you'd have to write:

```cpp
template< typename T >
void my_ptr_swap( T* t1, T* t2 ) // There are other ways to declare this
{
    T tmp( *t1 );
    *t1 = *t2;
    *t2 = tmp;
}
```

And now in the above example, you'd use my_ptr_swap( pi, pj ); to swap the values pointed to by pi and pj.

Personally, I think this solution stinks for three reasons. First, I have to remember two function names instead of one: my_ptr_swap and my_swap. Second, my_ptr_swap is harder to understand than my_swap because although they have the same number of lines of code and effectively do the same thing, there are extra deferences involved. (And I almost implemented the function wrong when I wrote it). Thirdly, NULL pointers! What happens if one or
both of the
pointers you pass to my_ptr_swap are NULL? Nothing good. In reality, if I wanted to make my_ptr_swap robust, to
avoid the crash, I'd have to write:

template< typename T >
void my_ptr_swap( T* t1, T* t2 )  // There are other ways to declare this
{
    if( t1 != NULL && t2 != NULL )
    {
        T tmp( *t1 );
        *t1 = *t2;
        *t2 = tmp;
    }
}

But this isn't exactly a great solution either, I suppose, because now the caller of my_ptr_swap cannot be 100% sure the function did anything unless they duplicate the if() check:

if( pi != NULL && pj != NULL )
    my_ptr_swap( pi, pj );
else
    std::cout << "Uh oh, my_ptr_swap won't do anything!" << std::endl;

But duplicating the check makes the check inside of my_ptr_swap kinda pointless. But on the other hand, a function should always validate its arguments. A conundrum. Perhaps a return value is in order:

template< typename T >
bool my_ptr_swap( T* t1, T* t2 )  // There are other ways to declare this
{
    if( t1 != NULL && t2 != NULL )
    {
        T tmp( *t1 );
        *t1 = *t2;
        *t2 = tmp;
        return true;
    }
    return false;  // false means function didn't swap anything
}

And now I can write:

if( my_ptr_swap( pi, pj ) == false )
    std::cout << "Uh oh, my_ptr_swap won't do anything!" << std::endl;

Which is definitely better. But this simply introduces an extra error leg in your program that you need to think about and deal with. What if my_ptr_swap fails? What should I do? In most trivial applications, handling errors, if even done at all, are easy. But in larger applications where you need to perform a sequence of 5 steps, each of which may fail
means you have to think about all of the following error legs:

1. What if operation #1 fails?
2. What if operation #2 fails? Do I roll back operation #1? What if
   rolling back operation #1 fails?
3. What if operation #3 fails? Do I roll back #1 and #2? What if
   rolling back operation #2 succeeds but #1 fails? What if rolling
   back operation #2 fails?
4. What if operation #4 fails? .... etc ...
5. What if operation #5 fails? .... etc ...

There are a mind-boggling number of failure cases to think about and test. Because it quickly gets complicated, most programmers handle only one failure; double faults are often not handled very gracefully (the program aborts in some way).

It seems to me that since error handling can easily dominate the design effort and implementation effort, programmers should strive NOT to artificially introduce error legs where they can easily be avoided.

One of the most prevalent error cases to deal with is that of a NULL pointer. Enter references.

```cpp
std::vector<int> v;
// assume v is filled out with values
// This is REALLY suboptimal, but I'm writing this without testing it, and I
// want to ensure I get it right:
for( size_t i = 0; i < v.size() - 1; ++i )
  for( size_t j = 0; j < v.size() - 1; ++ )
    if( v[ i ] < v[ j ] )
      my_ptr_swap( &v[ i ], &v[ j ] );
```

Students are taught that when a function needs to modify its parameters, you should pass by pointer. That's great for languages like Pascal which don't support references, but in C++, you have another option: pass by reference. In fact, if I replace the my_ptr_swap call with my_swap( v[ i ], v[ j ] ); it still works. AND, I've eliminated the use of pointers.

### Summary of when to pass parameters by value, by reference, and by pointer

1. Pass by value when the function does not want to modify the parameter and the value is easy to copy (ints, doubles, char, bool, etc... simple types. std::string, std::vector, and all other STL containers are NOT simple types.)

2. Pass by const pointer when the value is expensive to copy AND the function does not want to modify the value pointed to AND NULL is a valid, expected value that the function handles.

3. Pass by non-const pointer when the value is expensive to copy AND the function wants to modify the value pointed to AND NULL is a valid, expected value that the function handles.

4. Pass by const reference when the value is expensive to copy AND the function does not want to modify the value referred to AND NULL would not be a valid value if a pointer was used instead.

5. Pass by non-cont reference when the value is expensive to copy AND the function wants to modify the value referred to AND NULL would not be a valid value if a pointer was used instead.

6. When writing template functions, there isn't a clear-cut answer because there are a few
tradeoffs to consider that are beyond the scope of this discussion, but suffice it to say that most template functions take their parameters by value or (const) reference, however because iterator syntax is similar to that of pointers (asterisk to "dereference"), any template function that expects iterators as arguments will also by default accept pointers as well (and not check for NULL since the NULL iterator concept has a different syntax).
Pointers
Cplusplus.com

This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime. However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

Address-of operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator. For example:

```cpp
foo = &myvar;
```

This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the address-of operator (&), we are no longer assigning the content of the variable itself to foo, but its address.

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.

In this case, consider the following code fragment:

```cpp
myvar = 25;
foo = &myvar;
bar = myvar;
```

The values contained in each variable after the execution of this are shown in the following diagram:

<table>
<thead>
<tr>
<th></th>
<th>myvar</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1775</td>
<td>25</td>
<td>1777</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>foo</th>
<th></th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1776</td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>

First, we have assigned the value 25 to myvar (a variable whose address in memory we assumed to be 1776).

The second statement assigns foo the address of myvar, which we have assumed to be 1776.
Finally, the third statement, assigns the value contained in myvar to bar. This is a standard assignment operation, as already done many times in earlier chapters.

The main difference between the second and third statements is the appearance of the `address-of operator` (&).

The variable that stores the address of another variable (like foo in the previous example) is what in C++ is called a `pointer`. Pointers are a very powerful feature of the language that has many uses in lower level programming. A bit later, we will see how to declare and use pointers.

**Dereference operator (*)**

As just seen, a variable which stores the address of another variable is called a `pointer`. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the `dereference operator` (*). The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, the following statement:

```
baz = *foo;
```

This could be read as: "baz equal to value pointed to by foo", and the statement would actually assign the value 25 to baz, since foo is 1776, and the value pointed to by 1776 (following the example above) would be 25.

```
| foo         |
| 1776        |
|             |
| 1775 1776 1777 | {memory} |
|             |
| 25           |
```

It is important to clearly differentiate that foo refers to the value 1776, while *foo (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have added an explanatory comment of how each of these two expressions could be read):

```
1  baz = foo;   // baz equal to foo (1776)
2  baz = *foo;  // baz equal to value pointed to by foo (25)
```

The reference and dereference operators are thus complementary:

- & is the `address-of operator`, and can be read simply as "address of"
- * is the `dereference operator`, and can be read as "value pointed to by"

Thus, they have sort of opposite meanings: An address obtained with & can be dereferenced with *

Earlier, we performed the following two assignment operations:
Right after these two statements, all of the following expressions would give true as result:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

The first expression is quite clear, considering that the assignment operation performed on myvar was myvar=25. The second one uses the address-of operator (&), which returns the address of myvar, which we assumed it to have a value of 1776. The third one is somewhat obvious, since the second expression was true and the assignment operation performed on foo was foo=&myvar. The fourth expression uses the dereference operator (*) that can be read as "value pointed to by", and the value pointed to by foo is indeed 25.

So, after all that, you may also infer that for as long as the address pointed by foo remains unchanged, the following expression will also be true:

```
*foo == myvar
```

### Declaring pointers

Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

The declaration of pointers follows this syntax:

```
type * name;
```

where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int * number;
char * character;
double * decimals;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely going to occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the program runs). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char, and the last one to a double. Therefore, although these three example variables are all of them pointers, they actually have different types: int*, char*, and double* respectively, depending on the type they point to.

Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator seen a bit earlier, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.

Let's see an example on pointers:

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
```
int firstvalue, secondvalue;
int * mypointer;

mypointer = &firstvalue;
*mypointer = 10;
mypointer = &secondvalue;
*mypointer = 20;
cout << "firstvalue is " << firstvalue << '
';
cout << "secondvalue is " << secondvalue << '
';
return 0;
}

OUTPUT
firstvalue is 10
secondvalue is 20

Notice that even though neither firstvalue nor secondvalue are directly set any value in the program, both end up with a value set indirectly through the use of mypointer. This is how it happens:

First, mypointer is assigned the address of firstvalue using the address-of operator (&). Then, the value pointed to by mypointer is assigned a value of 10. Because, at this moment, mypointer is pointing to the memory location of firstvalue, this in fact modifies the value of firstvalue.

In order to demonstrate that a pointer may point to different variables during its lifetime in a program, the example repeats the process with secondvalue and that same pointer, mypointer.

Here is an example a little bit more elaborated:

// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;
  p1 = &firstvalue; // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10; // value pointed to by p1 = 10
  *p2 = *p1; // value pointed to by p2 = value pointed by p1
  p1 = p2; // p1 = p2 (value of pointer is copied)
  *p1 = 20; // value pointed by p1 = 20
  cout << "firstvalue is " << firstvalue << '
';
cout << "secondvalue is " << secondvalue << '
';
return 0;
}

OUTPUT
firstvalue is 10
secondvalue is 20

Each assignment operation includes a comment on how each line could be read: i.e., replacing ampersands (&) by "address of", and asterisks (*) by "value pointed to by".

Notice that there are expressions with pointers p1 and p2, both with and without the dereference operator (*). The meaning of an expression using the dereference operator (*) is very different from one that does not. When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name
appears without this operator, it refers to the value of the pointer itself (i.e., the address of what the pointer is pointing to).

Another thing that may call your attention is the line:

```c
int * p1, * p2;
```

This declares the two pointers used in the previous example. But notice that there is an asterisk (*) for each pointer, in order for both to have type int* (pointer to int). This is required due to the precedence rules. Note that if, instead, the code was:

```c
int * p1, p2;
```

*p1 would indeed be of type int*, but *p2 would be of type int. Spaces do not matter at all for this purpose. But anyway, simply remembering to put one asterisk per pointer is enough for most pointer users interested in declaring multiple pointers per statement. Or even better: use a different statement for each variable.

```c
int x;
int y = 10;
const int * p = &y;
x = *p;         // ok: reading p
*p = x;        // error: modifying p, which is const-qualified
```

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., *myptr). Therefore, the code above shall not be confused with:

```c
int myvar;
int * myptr;
*myptr = &myvar;
```

<table>
<thead>
<tr>
<th>Pointer declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *x</td>
<td>x is a pointer to int data type.</td>
</tr>
<tr>
<td>int *x[10]</td>
<td>x is an array[10] of pointer to int data type.</td>
</tr>
<tr>
<td>int *(x[10])</td>
<td>x is an array[10] of pointer to int data type.</td>
</tr>
<tr>
<td>int **x</td>
<td>x is a pointer to a pointer to an int data type – double pointers.</td>
</tr>
<tr>
<td>int (*x)[10]</td>
<td>x is a pointer to an array[10] of int data type.</td>
</tr>
<tr>
<td>int *funct()</td>
<td>funct is a function returning an integer pointer.</td>
</tr>
</tbody>
</table>
**C/C++ Pointers vs References**

Consider the following code:

**Pointers**

```
int i;
int *pi = &i;
```

**References**

```
int i;
int &ri = i;
```

In both cases the situation is as follows:

Since `x` is a reference to `i`, all operations on `x` are transformed by the compiler into operations on `i`. So doing `&x`, gives you the memory address `i` is in.

(Note that unlike pointers, references have the property of not being 're-referenced' after declared - they always reference the same thing - so there is no way to write operations that operate 'on the reference', not 'on what is referenced')

**Reference parameters**

If a function has reference parameter, it accesses the actual argument instead of a copy. This is similar to call by pointer, but there is no extra pointer involved here. Let us look at an example.
Now in this example, b is a reference parameter. So b++ will change our actual argument n.

Output will be

```
22
m and n are 10 13
```

The parameter can even be const & which indicates that the reference parameter can not be modified by the function.

It is faster to send a large object as a reference parameter.

**Pointers vs. References**

A pointer variable is declared with an asterisk between the type name and the variable name (the asterisk binds with the variable name). For instance,

```cpp
int *pValue;
```

declares `pValue` to be a pointer to an integer. The pointer can be initialized with (or assigned to, using the assignment operator = ) an address of some other variable. The address of operator is denoted by the ampersand (there is no conflict between this ampersand and the reference ampersand--they appear in different contexts). For instance,

```cpp
int TheValue = 10;
pValue = &TheValue;
```

assigns the address of the variable `TheValue` to the pointer `pValue`.

If we want to access the value to which the pointer points, we use the dereference operator, the asterisk (again, its
double meaning doesn’t lead to confusion). For instance,

```c
int i = *pValue;
```

assigns the value that `pValue` points to, to the integer `i`. In our example `i` will change its value to 10. Conversely,

```c
*pValue = 20;
```

### Pointer initialization

#### Initializing Pointers via the Address-Of Operator (`&`)

When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location. This is dangerous! You need to initialize a pointer by assigning it a valid address. This is normally done via the `address-of operator` (`&`).

The `address-of operator` (`&`) operates on a variable, and returns the address of the variable. For example, if `number` is an `int` variable, `&number` returns the address of the variable `number`.

You can use the address-of operator to get the address of a variable, and assign the address to a pointer variable. For example,

```c
int number = 88;    // An int variable with a value
int * pNumber;      // Declare a pointer variable called pNumber pointing to an int (or int pointer)
pNumber = &number;  // Assign the address of the variable number to pointer pNumber
```

```c
int * pAnother = &number;   // Declare another int pointer and init to address of the variable number
```

As illustrated, the `int` variable `number`, starting at address `0x22ccec`, contains an `int` value `88`. The expression `&number` returns the address of the variable `number`, which is `0x22ccec`. This address is then assigned to the pointer variable `pNumber`, as its initial value.

The `address-of operator` (`&`) can only be used on the RHS.

#### 1.4 Indirection or Dereferencing Operator (`*`)

The `indirection operator` (or `dereferencing operator`) (`*`) operates on a pointer, and returns the value stored in the address kept in the pointer variable. For example, if `pNumber` is an `int` pointer, `*pNumber` returns the `int` value "pointed to" by `pNumber`.

For example,

```c
int number = 88;
```
int * pNumber = &number;  // Declare and assign the address of variable number to pointer pNumber (0x22ccf0)
cout << pNumber << endl;  // Print the content of the pointer variable, which contain an address (0x22ccf0)
cout << *pNumber << endl;  // Print the value "pointed to" by the pointer, which is an int (88)
*pNumber = 99;  // Assign a value to where the pointer is pointed to, NOT to the pointer variable
cout << *pNumber << endl;  // Print the new value "pointed to" by the pointer (99)
cout << number << endl;  // The value of variable number changes as well (99)

Take note that pNumber stores a memory address location, whereas *pNumber refers to the value stored in the address kept in the pointer variable, or the value pointed to by the pointer.

As illustrated, a variable (such as number) directly references a value, whereas a pointer indirectly references a value through the memory address it stores. Referencing a value indirectly via a pointer is called indirection or dereferencing.

The indirection operator (*) can be used in both the RHS (temp = *pNumber) and the LHS (*pNumber = 99) of an assignment statement.

Take note that the symbol * has different meaning in a declaration statement and in an expression. When it is used in a declaration (e.g., int * pNumber), it denotes that the name followed is a pointer variable. Whereas when it is used in an expression (e.g., *pNumber = 99; temp << *pNumber;), it refers to the value pointed to by the pointer variable.

1.5 Pointer has a Type Too

A pointer is associated with a type (of the value it points to), which is specified during declaration. A pointer can only hold an address of the declared type; it cannot hold an address of a different type.

int i = 88;  
double d = 55.66;  
int * iPtr = &i;  // int pointer pointing to an int value  
double * dPtr = &d;  // double pointer pointing to a double value  

iPtr = &d;  // ERROR, cannot hold address of different type  
dPtr = &i;  // ERROR  
iPtr = i;  // ERROR, pointer holds address of an int, NOT int value

int j = 99;  
iPtr = &j;  // You can change the address stored in a pointer

Example

/* Test pointer declaration and initialization (TestPointerInit.cpp) */
#include <iostream>
using namespace std;

int main() {
    int number = 88;  // Declare an int variable and assign an initial value  
    int * pNumber;  // Declare a pointer variable pointing to an int (or int pointer)  
    pNumber = &number;  // assign the address of the variable number to pointer pNumber

    cout << pNumber << endl;  // Print content of pNumber (0x22ccf0)  
cout << &number << endl;  // Print address of number (0x22ccf0)  
cout << *pNumber << endl;  // Print value pointed to by pNumber (88)  
cout << number << endl;  // Print value of number (88)

    *pNumber = 99;  // Re-assign value pointed to by pNumber
    cout << pNumber << endl;  // Print content of pNumber (0x22ccf0)
cout << &number << endl;  // Print address of number (0x22ccf0)
cout << *pNumber << endl;  // Print value pointed to by pNumber (99)
cout << number << endl;    // Print value of number (99)
                          // The value of number changes via pointer

cout << &pNumber << endl;  // Print the address of pointer variable pNumber
                          // (0x22ecce)
}

Notes: The address values that you get are unlikely to be the same as mine. The OS loads the program in available
free memory locations, instead of fixed memory locations.

Pointers can be initialized to point to specific locations at the very moment they are defined:

    int myvar;
    int * myptr = &myvar;

The resulting state of variables after this code is the same as after:

    int myvar;
    int * myptr;
    myptr = &myvar;

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being
pointed (i.e., *myptr). Therefore, the code above shall not be confused with:

    int myvar;
    int * myptr;
    *myptr = &myvar;

Which anyway would not make much sense (and is not valid code).

The asterisk (*) in the pointer declaration (line 2) only indicates that it is a pointer, it is not the dereference operator
(as in line 3). Both things just happen to use the same sign: *. As always, spaces are not relevant, and never change
the meaning of an expression.

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another
pointer (or array):

    int myvar;
    int *foo = &myvar;
    int *bar = foo;

*p++  // same as *(p++): increment pointer, and dereference unincremented address
*++p  // same as *(++p): increment pointer, and dereference incremented address
+++p  // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to

A typical -but not so simple- statement involving these operators is:

    *p++ = *q++;

Because ++ has a higher precedence than *, both p and q are incremented, but because both increment operators
(++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are incremented. And then
both are incremented. It would be roughly equivalent to:

```
1  *p = *q;
2  ++p;
3  ++q;
```

Like always, parentheses reduce confusion by adding legibility to expressions.

/* References vs. Pointers (TestReferenceVsPointer.cpp) */

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

int main() {
    int number1 = 88, number2 = 22;

    // Create a pointer pointing to number1
    int * pNumber1 = &number1; // Explicit referencing
    *pNumber1 = 99; // Explicit dereferencing
    cout << *pNumber1 << endl; // 99
    cout << &number1 << endl; // 0x22ff18
    cout << pNumber1 << endl; // 0x22ff18 (content of the pointer var, same as above)
    cout << &pNumber1 << endl; // 0x22ff10 (address of the pointer variable)
    pNumber1 = &number2; // Pointer can be reassigned to store another address

    // Create a reference (alias) to number1
    int & refNumber1 = number1; // Implicit referencing (NOT &number1)
    refNumber1 = 11; // Implicit dereferencing (NOT *refNumber1)
    cout << refNumber1 << endl; // 11
    cout << &number1 << endl; // 0x22ff18
    cout << &refNumber1 << endl; // 0x22ff18
    //refNumber1 = &number2; // Error! Reference cannot be re-assigned
    // error: invalid conversion from 'int*' to 'int'
    refNumber1 = number2; // refNumber1 is still an alias to number1.
    // Assign value of number2 (22) to refNumber1 (and number1).
    number2++;
    cout << refNumber1 << endl; // 22
    cout << number1 << endl; // 22
    cout << number2 << endl; // 23
}
```

**new and delete Operators**

Instead of define an `int` variable (`int number`), and assign the address of the variable to the `int` pointer (`int *pNumber = &number`), the storage can be dynamically allocated at runtime, via a `new` operator. In C++, whenever you allocate a piece of memory dynamically via `new`, you need to use `delete` to remove the storage (i.e., to return the storage to the heap).

The `new` operation returns a pointer to the memory allocated. The `delete` operator takes a pointer (pointing to the memory allocated via `new`) as its sole argument.

For example,

```cpp
// Static allocation
int number = 88;
int * pl = &number; // Assign a "valid" address into pointer

// Dynamic Allocation
int * p2; // Not initialize, points to somewhere which is invalid
```
cout << p2 << endl; // Print address before allocation
p2 = new int;       // Dynamically allocate an int and assign its address to pointer
             // The pointer gets a valid address with memory allocated
*p2 = 99;
cout << p2 << endl; // Print address after allocation
cout << *p2 << endl; // Print value point-to
delete p2;        // Remove the dynamically allocated storage

Function Pointer

In C/C++, functions, like all data items, have an address. The name of a function is the starting address where the
function resides in the memory, and therefore, can be treated as a pointer. We can pass a function pointer into
function as well. The syntax for declaring a function pointer is:

    // Function pointer declaration
    return-type (* function-ptr-name) (parameter-list)

Example

    /* Test Function Pointers (TestFunctionPointer.cpp) */
    #include <iostream>
    using namespace std;

    int arithmetic(int, int, int (*)(int, int));
        // Take 3 arguments, 2 int's and a function
        //   int (*)(int, int), which takes two int's
        // and return an int
    int add(int, int);
    int sub(int, int);

    int add(int n1, int n2) { return n1 + n2; }
    int sub(int n1, int n2) { return n1 - n2; }

    int arithmetic(int n1, int n2, int (*)(int, int)) (int, int)) { return (*operation)(n1, n2);
int main() {
    int number1 = 5, number2 = 6;

    // add
    cout << arithmetic(number1, number2, add) << endl;
    // subtract
    cout << arithmetic(number1, number2, sub) << endl;
}

Pointers and const

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed by the pointer as const. For example:

    int x;
    int y = 10;
    const int * p = &y;
    x = *p;          // ok: reading p
    *p = x;          // error: modifying p, which is const-qualified

Here p points to a variable, but points to it in a const-qualified manner, meaning that it can read the value pointed, but it cannot modify it. Note also, that the expression &y is of type int*, but this is assigned to a pointer of type const int*. This is allowed: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around! As a safety feature, pointers to const are not implicitly convertible to pointers to non-const.

One of the use cases of pointers to const elements is as function parameters: a function that takes a pointer to non-const as parameter can modify the value passed as argument, while a function that takes a pointer to const as parameter cannot.

    // pointers as arguments:
    #include <iostream>
    using namespace std;

    void increment_all (int* start, int* stop)
    {
        int * current = start;
        while (current != stop) {
            ++(*current);  // increment value pointed
            ++current;     // increment pointer
        }
    }

    void print_all (const int* start, const int* stop)
    {
        const int * current = start;

while (current != stop) {
    cout << *current << '
';
    ++current;        // increment pointer
}

int main ()
{
    int numbers[] = {10,20,30};
    increment_all (numbers,numbers+3);
    print_all (numbers,numbers+3);
    return 0;
}

Note that print_all uses pointers that point to constant elements. These pointers point to constant content they cannot modify, but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.

And this is where a second dimension to constness is added to pointers: Pointers can also be themselves const. And this is specified by appending const to the pointed type (after the asterisk):

    int x;
    int * p1 = &x;  // non-const pointer to non-const int
    const int * p2 = &x;  // non-const pointer to const int
    int * const p3 = &x;  // const pointer to non-const int
    const int * const p4 = &x;  // const pointer to const int

The syntax with const and pointers is definitely tricky, and recognizing the cases that best suit each use tends to require some experience. In any case, it is important to get constness with pointers (and references) right sooner rather than later, but you should not worry too much about grasping everything if this is the first time you are exposed to the mix of const and pointers. More use cases will show up in coming chapters.

To add a little bit more confusion to the syntax of const with pointers, the const qualifier can either precede or follow the pointed type, with the exact same meaning:

    const int * p2a = &x;  // non-const pointer to const int
    int const * p2b = &x;  // also non-const pointer to const int

As with the spaces surrounding the asterisk, the order of const in this case is simply a matter of style. This chapter uses a prefix const, as for historical reasons this seems to be more extended, but both are exactly equivalent. The merits of each style are still intensely debated on the internet.

**Pointers and string literals**

As pointed earlier, *string literals* are arrays containing null-terminated character sequences. In earlier sections, string literals have been used to be directly inserted into cout, to initialize strings and to initialize arrays of characters.

But they can also be accessed directly. String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type const char (as literals, they can never be modified). For example:

    const char * foo = "hello";

This declares an array with the literal representation for "hello", and then a pointer to its first element is assigned to foo. If we imagine that "hello" is stored at the memory locations that start at address 1702, we can represent the previous declaration as:
Note that here foo is a pointer and contains the value 1702, and not 'h', nor "hello", although 1702 indeed is the address of both of these.

The pointer foo points to a sequence of characters. And because pointers and arrays behave essentially in the same way in expressions, foo can be used to access the characters in the same way arrays of null-terminated character sequences are. For example:

```
*(foo+4)
foo[4]
```

Both expressions have a value of 'o' (the fifth element of the array).

**Literals**

Literals are the most obvious kind of constants. They are used to express particular values within the source code of a program. We have already used some in previous chapters to give specific values to variables or to express messages we wanted our programs to print out.

**Pointers to pointers**

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). The syntax simply requires an asterisk (*) for each level of indirection in the declaration of the pointer:

```c
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, assuming the randomly chosen memory locations for each variable of 7230, 8092, and 10502, could be represented as:

```
a
\textcolor{red}{\texttt{z}}
\downarrow
7230
```

```
b
\downarrow
8092
```

```
c
\downarrow
10502
```

With the value of each variable represented inside its corresponding cell, and their respective addresses in memory represented by the value under them.

The new thing in this example is variable c, which is a pointer to a pointer, and can be used in three different levels of indirection, each one of them would correspond to a different value:

- c is of type char** and a value of 8092
- *c is of type char* and a value of 7230
- **c is of type char and a value of 'z'
**void pointers**

The void type of pointer is a special type of pointer. In C++, void represents the absence of type. Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters. In exchange, they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

One of its possible uses may be to pass generic parameters to a function. For example:

```cpp
// increaser
#include <iostream>
using namespace std;

void increase (void* data, int psize)
{
    if ( psize == sizeof(char) )
    {
        char* pchar; pchar=(char*)data; ++(*pchar);
    }
    else if (psize == sizeof(int) )
    {
        int* pint; pint=(int*)data; ++(*pint);
    }
}

int main ()
{
    char a = 'x';
    int b = 1602;
    increase (&a,sizeof(a));
    increase (&b,sizeof(b));
    cout << a << ', ' << b << '
';
    return 0;
}
```

sizeof is an operator integrated in the C++ language that returns the size in bytes of its argument. For non-dynamic data types, this value is a constant. Therefore, for example, sizeof(char) is 1, because char is has always a size of one byte.

**Invalid pointers and null pointers**

In principle, pointers are meant to point to valid addresses, such as the address of a variable or the address of an element in an array. But pointers can actually point to any address, including addresses that do not refer to any valid element. Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

```cpp
int * p; // uninitialized pointer (local variable)
int myarray[10];
int * q = myarray+20; // element out of bounds
```

Neither p nor q point to addresses known to contain a value, but none of the above statements causes an error. In
C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not. What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to). Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address. For such cases, there exists a special value that any pointer type can take: the null pointer value. This value can be expressed in C++ in two ways: either with an integer value of zero, or with the nullptr keyword:

```cpp
int * p = 0;
int * q = nullptr;
```

Here, both p and q are null pointers, meaning that they explicitly point to nowhere, and they both actually compare equal: all null pointers compare equal to other null pointers. It is also quite usual to see the defined constant NULL be used in older code to refer to the null pointer value:

```cpp
int * r = NULL;
```

NULL is defined in several headers of the standard library, and is defined as an alias of some null pointer constant value (such as 0 or nullptr).

Do not confuse null pointers with void pointers! A null pointer is a value that any pointer can take to represent that it is pointing to "nowhere", while a void pointer is a type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer, and the other to the type of data it points to.

**Pointers to functions**

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```cpp
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}

int main ()
{
    int m,n;
    int (*minus)(int,int) = subtraction;
```
m = operation (7, 5, addition);
n = operation (20, m, minus);

cout << n;
return 0;
}

OUTPUT
8

In the example above, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function subtraction:

int (* minus)(int, int) = subtraction;

C++ program example compiled using g++.  
// illustrates that function receives addresses of variables and then alters their contents
#include <iostream>
using namespace std;

int main()
{
    int x = 4, y = 7;
    // function prototype...
    void addcon(int*, int*);
    cout << "nInitial value of x = " << x;
    cout << "nInitial value of y = " << y;
    cout << "nThen calls function addcon()n";
    cout << "nBringing along the &x = " << &x << "endl;
    cout << "and &y = " << &y << "n";
    cout;
    // function call, address of x any y are passed to addcon()
    addcon(&x, &y);
    cout << "nAdd 10...n";
    cout << "nNew value of x = " << x;
    cout << "nminus 10...n";
    cout << "nNew value of y = " << y << endl;
    return 0;
}

// a function definition, parameters are pointers...
void addcon(int *px, int *py)
{
    // adds 10 to the data stored in memory pointed to by px
    *px = *px + 10;
    // minus 10 to the data stored in memory pointed to by py
    *py = *py - 10;
}

[bodo@bakawali ~]$ g++ funcref.cpp -o funcref
[bodo@bakawali ~]$ ./funcref

Initial value of x = 4
Initial value of y = 7
Then calls function addcon()

Bringing along the &x = 0xbfed7944
and &y = 0xbfed7940

Add 10...
New value of x = 14
minus 10...
New value of $y = -3$

Pointers

Pointers are variables whose values are memory locations – addresses of some data.

C++ pointers

C++ is different. In C++ you have a choice. In C++ you have access to actual memory locations. C++ is dangerous. It's like the second amendment, which gives you the power to bear arms, but does not prevent you from shooting your own leg.

The null value in C++ is the number 0 (memory address 0).

Declaring a pointer:

1. `int *p_i;`
2. `float *p_f;`

Accessing the values of a pointer:

1. `int j = 5;`
2. `int *p_j = &j; // p_j value is the address of the variable j`
3. `cout << p_j << endl; // prints the address of variable j`
4. `cout << *p_j << endl; // prints the value of variable j (5)`

Example of bad but possible assigning to pointers

1. `int *p_i = (int *) 923451; // p_i points to the int at memory location 923451`
2. `cout << p_i << endl; // prints the number (address) 923451`
3. `cout << *p_i << endl; // prints the value of the integer at address 923451`
4. `// (might be garbage, might crash the program).`

The operators * and &

When dealing with pointers, the operator * has 2 different meanings, depending on the context in which it appears.

- In variable definition, it means "this variable is a pointer".
- Once the variable is defined, it means "the value at the address pointed to by the variable".

The operator & is "the address of the given variable".

[Note that in different contexts, these operators can represent completely different operations. For example, * is also the multiplication operator, and & is the bitwise and operator (as well as the reference operator, which we'll learn later). In addition, C++ supports operator overloading, which allow you to redefine these and other operators for the classes you create].

Comparing pointers:

1. `int j = 5;`
2. `int i = 5;`
3. `int *p_j = &j; // p_j value is the address of the variable j`
4. `int *p_j2 = &j; // p_j2 value is the address of the variable j also`
5. `int *p_i = &i; // p_i value is the address of variable i`
6. if (p_j == p_j2) { cout << "1" << endl; }
7. if (p_j == p_i) { cout << "2" << endl; }
8. if (p_j != p_i) { cout << "3" << endl; }
9. if (*p_j == *p_i) { cout << "4" << endl; }

(what will be printed??)

You can also have a pointer to a pointer:

1. int num = 5;
2. int *x = &num;
3. int **y = &x;

4. double num2 = 2.5;
5. double* z = &num2;

6. cout << y << endl; // this is the memory location of x
7. cout << *y << endl; // this is the memory location of num
8. cout << **y << endl; // 5
Pointers to pointers

A pointer is just a 32bit (or 64bit) number. So why do we need different types of pointers (a.k.a: pointer to int, pointer to char, etc.)?

**Type safety**: telling the compiler to check that we meant what we wrote is always a good idea (recall the use of generics in java).

As a side effect, we need to discuss pointer arithmetic.

The operators+, +=, ++, --, -=, -- are all defined for pointers. But what does it mean to add a number to a pointer? Let's see:

1. int j = 5;
2. int *i = &j;
3. cout << "i = " << i << endl;
4. i+=1;
5. cout << "i+1 = " << i << endl;

   i = 0x22ccc4
   i+1 = 0x22ccc8
The difference is 4. Why?

Adding 1 to a pointer means "make it point to the next element of the same type", so adding 1 to a pointer to int actually adds $1 \times \text{sizeof(int)}$ (which happens to equal 4 on my computer). Similarly, adding 5 to a pointer of type float actually adds $5 \times \text{sizeof(float)}$. To add exactly 1 (byte) to a pointer use the type char since 
\[ \text{sizeof(char)} == 1. \]

**Arrays**

Arrays in C++ are just special pointers.

The special thing about them is that they point to the start of a memory block, and this memory block is on the stack. You can not make an array pointer point to anywhere else. (However, you can treat "non-special" pointers as arrays as well, and we'll see that below. We also strongly recommend you not to use arrays. Use vector instead of arrays)

Here is how you declare an array of 10 integers in C++:
\[ \text{int A[10];} \]

Let's understand what this line means:

- A memory block big enough for holding 10 integers ($10 \times \text{sizeof(int)}$) is allocated on the stack.
- \( A \) holds the value of the start of this block.

Here is how you access an array element:
\[
\begin{align*}
\text{int A[10];} \\
A[3] &= 9; \\
\text{int j = A[3];}
\end{align*}
\]

The meaning of these lines is:

- The first line declares an array of 10 integers. \( A \) is a pointer holds the address of the first element.
- The second line puts the value 9 in the 4th place in the array. Note that the location of the 4th place in the array is "the value of \( A \) plus 3*sizeof(int)", or using pointers arithmetic: \( A + 3 \). So this line could be written as \( *(A+3) = 9; \).
- The third line reads a value from the 4th place in the array. Note that this line could be written as \( \text{int j = *(A+3)}. \)

Notice that unlike Java, C++ arrays don't have a "length" properties: C++ arrays are just special pointers, and they don't know the size of the memory block they are pointing to. You need to track this information yourself if need it.

For this reason, C++ arrays will not give you an "array out of bound" exception if you try to access elements beyond the allocated memory. This is very dangerous: the operation will probably work, but write or read things you didn't intend to.

For example, what does the following lines do?
\[
\begin{align*}
\text{int A[10];} \\
A[100] &= 9; \\
A[-2] &= 500;
\end{align*}
\]

Regular pointers can also be used like arrays. You just have to make sure you actually try and access a reasonable memory location (Assume we have a 32 bit address length):
\[
\begin{align*}
\text{double A[10];} \\
\text{double s=0; } //\text{just to clear size of a box} \\
\text{double *pi;} \\
\text{double j;}
\end{align*}
\]

1. /* The stack at this point is depicted in the image below.*/
pi = &j;
pi[3] = 5;

pi = A;
pi[3] = 5; // OK. Accessing the 4th element of A.
pi++;    // now pi points to the second element of A
pi[0] = 1; // assign 1 to the second element of A.
/* Question: What will happen, if s is an int and not double */

Illustration of the stack as a result of the above code

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>1908</td>
</tr>
<tr>
<td>pi</td>
<td>1916</td>
</tr>
<tr>
<td>s</td>
<td>1920</td>
</tr>
<tr>
<td>A[0]</td>
<td>1928</td>
</tr>
<tr>
<td>A[1]</td>
<td></td>
</tr>
</tbody>
</table>

The file Pointers.cpp has some examples of using pointers and arrays (It also introduces "structures", which we didn't learn yet. Structures are almost classes, and we'll get to them later in the course).

Dynamically allocating arrays

Declaring arrays dynamically allows us to choose their size while the program is running. To allocate an array dynamically, we use the array form of new and delete (often called new[] and delete[]):

```cpp
int nSize = 12;
int *pnArray = new int[nSize]; // note: nSize does not need to be constant!
pnArray[4] = 7;
delete[] pnArray;
```

Strings vs. char*

In the previous practical session we've seen the std::string object.

This is similar to a Java string: it's an instance of a class representing a string.

C++ has another way of representing string: an array of characters, where the last character in the string has the value 0.

1. char *c = "abcdefg";
2. std::cout << c[3] << endl; // what will be printed?
3. std::cout << &(c[3]) << endl; // what will be printed? (defg)

Next, we see how to convert std::strings() to char* strings and back:
1. `std::string s("abcdefg");` // what goes on in this line?

2. // `c_str()` returns a char* representation of the string
3. // NOTE: it returns a pointer to an internal representation.
4. // this is GOOD: because you don't have to manage this memory yourself
5. // (more on memory management below)
6. // this is also BAD: there's no guarantee you will keep pointing
7. // to a valid c-string after additional calls
8. // `c_str()` returns a char* representation of the string.
9. const char *cs = s.c_str();

10. // this is actually very similar to the first line.
11. `std::string s2 = std::string(cs);`

The commandline arguments to a C++ programs are passed as c-strings (char* string) to the main function, as in the following code sample: `commandline.cpp`.

```cpp
pstr = new char[20];    // Allocate a string of twenty characters
int *pnArray = new int[nSize];   // Dynamically allocating arrays
                                     // nSize does not need to be constant!
flist = new Fraction[20];        // dynamic array of 20 Fraction objects
                                     // default constructor used on each
```

**Memory Handling**

Up till now we allocated memory on the stack. Now we learn how to allocate memory on the heap. This is done with the `new` operator.

`new` allocates memory on the heap, initializes it and returns a pointer to it.

Allocating primitives:

1. `int *i = new int;` // i points to a heap location containing 0
2. `int *i2 = new int(2);` // i2 points to a heap location containing 2
3. // if this was a complete code, it would have been terrible! we don't free the memory.

Allocating objects:

1. `string *s = new string("I'm on the heap");`
2. `cout << "s" << s << endl;`
3. `cout << "s" << *s << endl;`
4. `delete s;` // memory on the heap must be freed!

All the memory allocated by `new` must be freed, or you will get a memory leak. The `delete` operator takes a pointer to an allocated space on the heap, and frees it: the memory is returned to the OS. However, you need to be very careful: don't try to `delete` a pointer not allocated by `new`!
Safe delete

```c
if (0 != p){
    delete p;
    p=0;
}
```

Safe delete is recommended when a pointer is shared in many functions.

Memory allocation may fail. There might not be enough space on the heap, in which case `new` will throw the `std::bad_alloc` exception. You should only catch it if the program is able to recover from such a state.

**Arrays on the heap**

We've seen before that arrays are just pointers to a block of memory. You can use the `new[]` operator to allocate a block of memory on the heap. Use `delete[]` to free the allocated memory.

1. `int *A = new int[10]; // allocate 10 integers on the heap`
2. `int *B = A + 1;`
3. `A[0] = 1;`
5. `B[1] = 3;`
6. `delete[] A; // free the allocated memory`

Illustration of the stack and the heap as a result of the above code

**Pointers Dangers**

When you use pointers, beware of the following situations:

- **Uninitialized pointers**
  ```c
  int *p; // p is an uninitialized pointer
  *p = 3; // bad!
  ```

- **Dereferencing a null pointer**
  ```c
  int *p = NULL; // p is a NULL pointer
  *p = 3; // bad!
  ```

- **Dereferencing a deleted pointer**
  ```c
  int *p = new int;
  delete p;
  *p = 5; // bad!
  ```
• Dereferencing a dangling pointer
  ```c
  int *p, *q;
p = new int;
q = p;       // p and q point to the same location
delete q;   // now p is a dangling pointer
*p = 3;     // bad!
  ```

• If p is the only pointer that points to dynamically allocated memory, and you reassign p without first deleting it, that memory will be lost (your code will have a storage leak)
  ```c
  int *p = new int;
p = NULL;       // reassignment to p without freeing the storage
            // it pointed to -- storage leak!
  ```

• use `delete` for freeing memory allocated with `new` and `delete[]` for memory allocated with `new[]`

Memory leak

Memory leak occurs when the program fails to release memory that is no longer needed. A memory leak can diminish the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or devices stops working correctly, the application fails, or the system slows down.
Skills for using C++ smart pointers

Introduction

C++ is well-known for efficiency. On the other hand, there are some traps which annoy developers. As you know, wild pointers and memory leaks are problems which happen to us most. Those problems make some developers desperate sometimes and makes using C++ difficult. Actually there are some methods to avoid those problems. Here, I'd like to share my experience dealing with those problems.

Smart Pointers

Most C++ developers know about smart pointers. We can use them with boost or C++ 11. There are lots of articles that introduce smart pointers but less of those clarify how to use them right. Unfortunately there are some opinions to be ignored when using smart pointers. To some extent, that's right. Using smart pointers improperly may bring bad results, even memory leaks. So let's talk about the rules of using smart pointers.

Generally there are three kinds of smart pointers:

Scope Pointer: Scope pointer will auto release the object when the scope pointer exits a code statement. This is a so easy, one that we nearly will not make mistakes. So we will talk more about it.

Shared pointer. Object will not be released until a non-shared pointer has the reference. This will be very useful to avoid wild pointers.

Weak pointer. This must be used by combining with a shared pointer. Weak Pointer has a reference to a shared pointer. Weak pointer can be converted to a shared pointer. But if a shared pointer has released the object, the weak pointer will auto release the reference.

Mgr class to use shared pointer

We like to use a class like xx_mgr to manage objects. There are some codes using the game server model as an example:

```cpp
class player_t
{
public:
    int id();
};
typedef shared_ptr<player_t> player_ptr_t;

class player_mgr_t
{
public:
    int add(player_ptr_t player_)
    {
        m_players[player_->id()] = player_;
        return 0;
    }
    player_ptr_t get(int id_)
    {
        map<int, player_ptr_t>::iterator it = m_players.find(id_);
        if (it != m_players.end())
        {
            return it->second;
        }
    }
};
```
```cpp
int del(int id_)
{
    m_players.erase(id_);
    return 0;
}
```

player_mgr_t controls the life cycle of player_t. We must make sure only player_mgr_t owns the player_t shared pointer. We can use a temporary variable of the player_t shared pointer which will be auto released when the function exits. So when the player gets offline, we delete it from player_mgr_t so that this player_t object will be auto released. That's what we expect.

**Recourse object to use shared pointer**

We are all familiar with the situation that we new an object or buffer to put contents read from a file or database. I recently solved a memory leak in my project that was caused by a MySQL query. The MySQL API of `mysql_fetch_result` returns an object that needs to free even though a zero row is included. For example:

```cpp
struct db_data_t{
    string data;
};
typedef shared_ptr<db_data_t> db_data_ptr_t;
db_data_ptr_t load_data(const string& sql){
    return new db_data_t();
}
int process_1(db_data_ptr_t data_);
int process_2(db_data_ptr_t data_);
int process_3(db_data_ptr_t data_);
int process(db_data_ptr_t data_){
    db_data_ptr_t data = load_data(sql);
    if (process_1(data)){
        return -1;
    }
    if (process_2(data)){
        return -1;
    }
    if (process_3(data)){
        return -1;
    }
}
```
As you can see, whenever we quit the process, the object of `db_data` will be deconstructed. That makes sure `db_data_t` will not cause a memory leak.

**Property object to use shared pointer**

This situation happens when two objects have the owner relationship. E.g., `player_t` may have a weapon, or may not. Once we allocate a weapon object to `player_t`, the weapon will not be destroyed until `player_t` is destroyed. Let’s see an example code:

```cpp
class weapon_t;
typedef shared_ptr<weapon_t> weapon_ptr_t;
class player_t
{
public:
    int id();
    weapon_ptr_t get_weapon() { return m_weapon; }
    void set_weapon(weapon_ptr_t weapon_) { m_weapon = weapon_; }
protected:
    weapon_ptr_t m_weapon;
};
```

**Reference relationship to use weak pointer**

I use weak pointers besides the situations mentioned above. Using a game server model is an example. Monster will auto attack players trying to approach it. So class `monster_t` has an interface `lock_target()` which will be auto invoked by the AI system. We should set the target player of the monster to `null` if the target player gets offline. That is annoying. That is where we regularly make mistakes. And weak pointer fits here very much.

```cpp
class monster_t{
public:
    void set_target(shared_ptr<player_t> player_){
        m_target_player = player_; 
    }
    shared_ptr<player_t> get_target() { return m_target_player.lock(); }
protected:
    weak_ptr<player_t> m_target_player;
};
```

`m_target_player` will be auto set to `null` if the real player gets offline. Do you remember the example of the `mgr` object? If we combine them together, it's easy to manage memory allocation.

**Object Counter**

Though we use smart pointers, we likely make mistakes that make program memory increase all the way. For example, we forget to delete `player_t` from `player_mgr_t` when the player gets offline in some situations; that will not generate a memory leak. Another example, our program
will sometimes increase memory faster than at other times. But which object causes these memory allocations? That usually can't be tested in the developing environment. So we have a need to know object numbers anytime. Generally we need to dump the data of object numbers to a file as time goes on. Here is a simple implementation:

```cpp
#include <stdio.h>
#include "base/fftype.h"
using namespace ff;

class foo_t: public ffobject_count_t<foo_t>
{
};
class dumy_t: public foo_t, ffobject_count_t<dumy_t>
{
};

int main(int argc, char* argv[]) {
    foo_t foo;
    dumy_t dumy;
    map<string, long> data = singleton_t<obj_summary_t>::instance().get_all_obj_num();
    printf("foo_t=%ld, dumy_t=%ld\n", data["foo_t"], data["dumy_t"]);
    return 0;
}
```

output:

```
foo_t=2, dumy_t=1
```

I will upload the files of the implementation code. I just posted some key codes. First, `ffobject_count_t` will auto increase the number of objects during construction, and will auto decrease the number during destruction.
singleton_t<obj_summary_t>::instance().reg(this);
}
virtual const string& get_name() { return TYPE_NAME(T); }
};
template<typename T>
class ffobject_count_t
{
public:
    ffobject_count_t()
    {
        singleton_t<obj_counter_t<T> >::instance().inc(1);
    }
    virtual ~ ffobject_count_t()
    {
        singleton_t<obj_counter_t<T> >::instance().dec(1);
    }
};

If we dump the data of object number to a file in a timely fashion, we will get such a file:

```
obj,num,20120606-17:01:41
dumy,1111
foo,222
obj,num,20120606-18:01:41
dumy,11311
foo,2422
obj,num,20120606-19:01:41
dumy,41111
foo,24442
```

It's easy to write a tool to analyze the data. For example, we can use highchart (js lib) to draw lines. I have uploaded my implementation of the tool. The picture generated is like this:
Summary

- We should be careful with the C++ memory allocation.
- Smart pointers help us deal with memory allocations more easily.
- In my opinion, we should use Smart pointers always but properly.
- We should know when to use a shared pointer and when to use a weak pointer. If you make mistakes, that makes the situation worse than the original pointer.
- An object counter should be part of the infrastructure of our C++ program. That helps us analyze memory allocation as time goes.