This document is based on http://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html

Pointers and References

Pointers, References and Dynamic Memory Allocation are the most powerful features in C/C++ language, which allows programmers to directly manipulate memory to *efficiently manage the memory* - the most critical and scarce resource in computer - *for best performance*. However, "pointer" is also the most complex and difficult feature in C/C++ language.

Pointers are extremely powerful because they allows you to access addresses and manipulate their contents. But they are also complex to handle. Using them correctly, they could greatly improve the efficiency and performance. On the other hand, using them incorrectly could lead to many problems, from un-readable and un-maintainable codes, to infamous bugs such as memory leaks and buffer overflow, which may expose your system to hacking. Many new languages (such as Java and C#) remove pointer from their syntax to avoid the pitfalls of pointers, by providing automatic memory management.

Although you can write C/C++ programs without using pointers, however, it is difficult not to mention pointer in teaching C/C++ language.

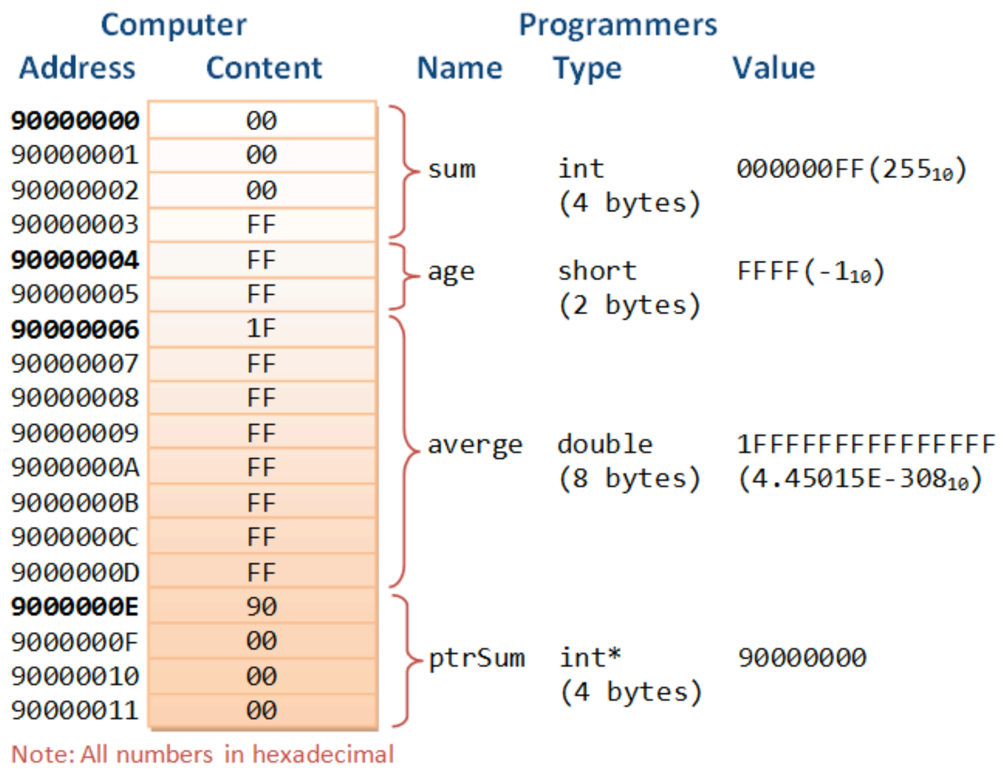
Pointer Variables

A *computer memory location* has an *address* and holds a *content*. The *address* is a numerical number (often expressed in hexadecimal), which is hard for programmers to use directly. Typically, each address location holds 8-bit (i.e., 1-byte) of data. It is entirely up to the programmer to interpret the meaning of the data, such as integer, real number, characters or strings.

To ease the burden of programming using numerical address and programmer-interpreted data, early programming languages (such as C) introduce the concept of variables. A variable is a *named* location that can store a *value* of a particular *type*. Instead of numerical addresses, names (or identifiers) are attached to certain addresses. Also, types (such as `int`, `double`, `char`) are associated with the contents for ease of interpretation of data.

Each address location typically hold 8-bit (i.e., 1-byte) of data. A 4-byte `int` value occupies 4 memory locations. A 32-bit system typically uses 32-bit addresses. To store a 32-bit address, 4 memory locations are required.

The following diagram illustrates the relationship between computers' memory *address* and content; and variable's *name*, *type* and *value* used by the programmers.



Pointer Variables (or Pointers)

A *pointer variable* (or *pointer* in short) is basically the same as the other variables, which can store a piece of data. Unlike normal variable which stores a value (such as an int, a double, a char), a *pointer stores a memory address*.

Declaring Pointers

Pointers must be declared before they can be used, just like a normal variable. The syntax of declaring a pointer is to place a * in front of the name. A pointer is associated with a type (such as int and double) too.

```
type *ptr;   // Declare a pointer variable called ptr as a pointer of type
// or
type* ptr;
// or
type * ptr;  // I shall adopt this convention
```

For example,

```
int * iPtr;     // Declare a pointer variable called iPtr pointing to an int (an int pointer)
                // It contains an address. That address holds an int value.
double * dPtr;  // Declare a double pointer
```

Take note that you need to place a * in front of each pointer variable, in other words, * applies only to the name that followed. The * in the declaration statement is not an operator, but indicates that the name followed is a pointer variable. For example,

```
int *p1, *p2, i;    // p1 and p2 are int pointers. i is an int
```

```
int* p1, p2, i;     // p1 is a int pointer, p2 and i are int
int * p1, * p2, i;  // p1 and p2 are int pointers, i is an int
```

Naming Convention of Pointers: Include a "p" or "ptr" as *prefix* or *suffix*, e.g., iPtr, numberPtr, pNumber, pStudent.

Initializing Pointers via the Address-Of Operator (&)

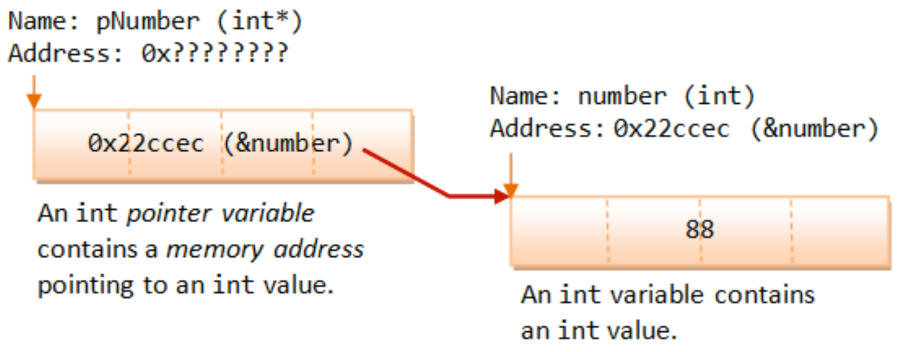
When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location. This is dangerous! You need to initialize a pointer by assigning it a valid address. This is normally done via the *address-of operator*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

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,

```
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

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


Name: pNumber (int*)
Address: 0x????????
0x22ccec (&number)
An int pointer variable contains a memory address pointing to an int value.
Name: number (int)
Address: 0x22ccec (&number)
88
An int variable contains an int value.

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.

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,

```
int number = 88;

int * pNumber = &number;   // Declare and assign the address of variable number to pointer pNumber (0x22ccec)
printf("%p\n", pNumber);   // Print the content of the pointer variable, which contain an address (0x22ccec)
printf("%d\n", *pNumber);  // 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
printf("%d\n", *pNumber);  // Print the new value "pointed to" by the pointer (99)
printf("%d\n", number);    // 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.

A 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;   // WRONG, cannot hold address of different type
dPtr = &i;   // WRONG
iPtr = i;    // WRONG, 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.c) */
#include<stdio.h>

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

  printf("%p\n", pNumber);   // Print content of pNumber (0x22ccf0)
  printf("%p\n", &number);   // Print address of number (0x22ccf0)
  printf("%d\n", *pNumber);  // Print value pointed to by pNumber (88)
  printf("%d\n", number);    // Print value of number (88)

  *pNumber = 99;            // Re-assign value pointed to by pNumber
  printf("%p\n", pNumber);  // Print content of pNumber (0x22ccf0)
  printf("%p\n", &number);  // Print address of number (0x22ccf0)
  printf("%d\n", *pNumber); // Print value pointed to by pNumber (99)
  printf("%d\n", number);   // Print value of number (99)
                            // The value of number changes via pointer

  printf("%p\n", &pNumber); // Print the address of pointer variable pNumber (0x22ccec)
}
```

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.

Uninitialized Pointers

The following code fragment has a serious logical error!

```
int * iPtr;
*iPtr = 55;
printf("%d\n", *iPtr);
```

The pointer `iPtr` was declared without initialization, i.e., it is pointing to "somewhere" which is of course an invalid memory location. The `*iPtr = 55` corrupts the value of "somewhere"! You need to initialize a pointer by assigning it a valid address. Most of the compilers does not signal an error or a warning for uninitialized pointer?!

Null Pointers

You can initialize a pointer to 0 or `NULL`, i.e., it points to nothing. It is called a *null pointer*. Dereferencing a null pointer (*p) causes a runtime error.

```
int * iPtr = 0;      // Declare an int pointer, and initialize the pointer to point to nothing
printf("%d\n", *iPtr);   // ERROR!

int * p = NULL;         // Also declare a NULL pointer points to nothing
```

Initialize a pointer to null during declaration is a good software engineering practice.