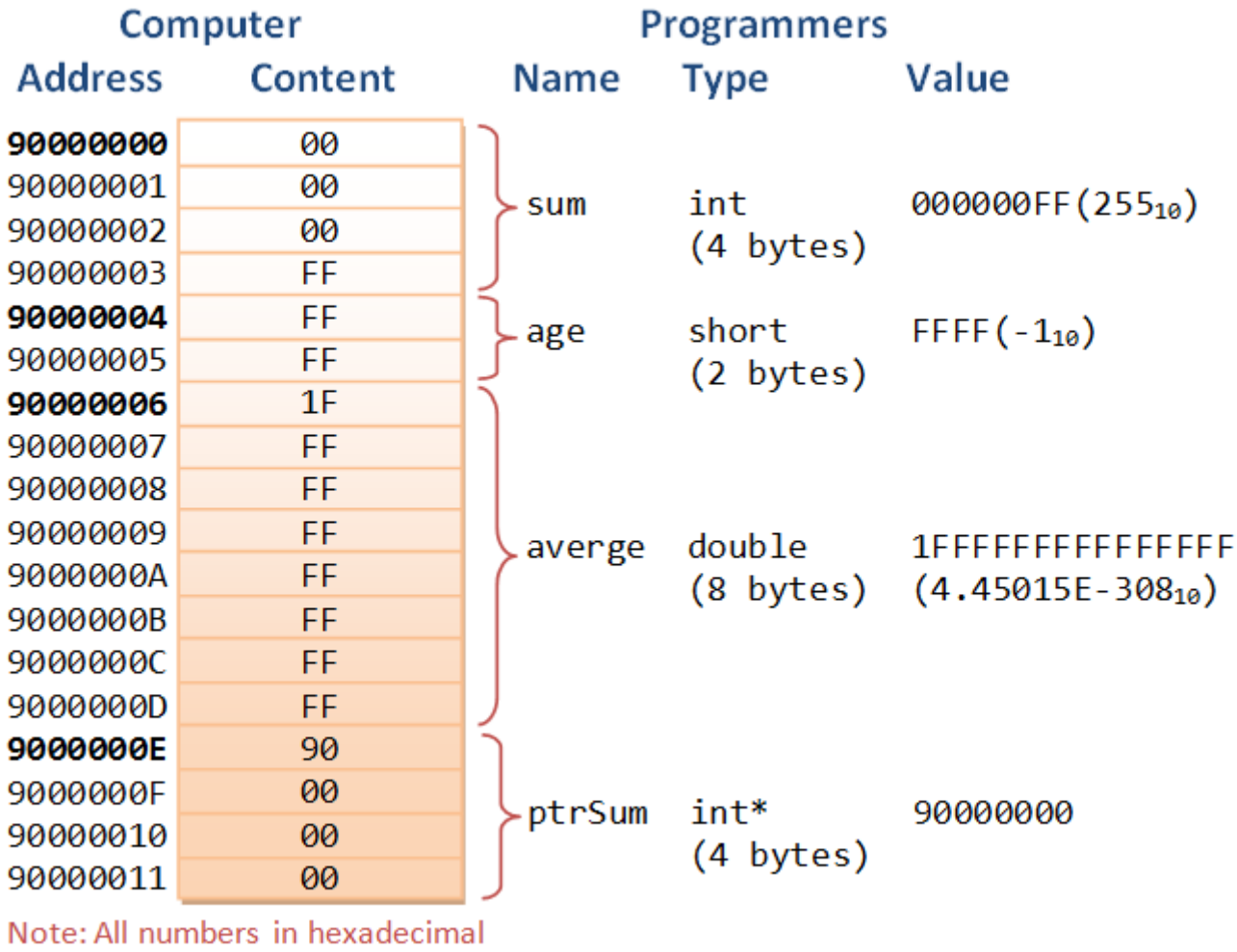
C Notes

Object reference vs pointer

Pointer variable

Let's start from the beginning, computer memory location are layered out in addresses, each particular location have an address and holds some kind of content. The address is a numerical number usually in hex for easier expression.

To help programmer, instead of using the hex address remembering them, variable are created as a named location. So instead of using the numerical address, you use a name that is attached to that particular location.



As you can see the variable `sum, age, average, ptrSum` are all named location in replacement of using the direct address.

Integer type variable holds integer value.

Double type variable holds float value.

Pointer type variable holds memory address value. It is just like any other variable that holds a value.

Pointer

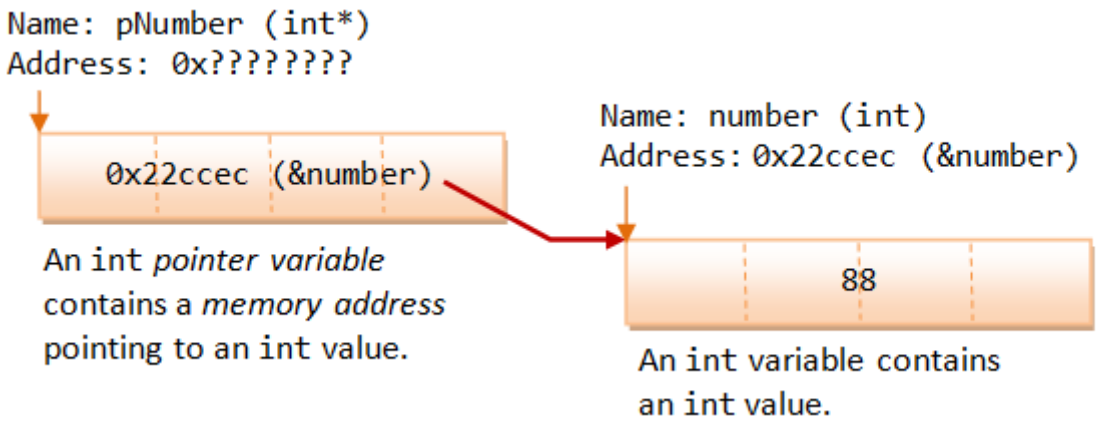
You declare them by using the `*` symbol like so:

```
type * ptr;
```

When they are declared they usually have garbage value initially, thatis why you need to initialize them with a value before they can be used. You initialize them using the address-of operator (&).

The address-of operator operates on a variable, and returns the address of the variable. Say `number` is an integer variable, `&number` returns the address of the variable `number`.

```
int number = 88;
int * pNumber = &number;
```



Dereferencing

To follow the pointer and actually retrieve the value from the pointer you will use the dereferencing operator (*).

```
int number = 99;
int *pNumber = &number;


int anotherNumber = *pNumber; // puts 99 into anotherNumber
```

Reference variables

C++ added reference variables or reference for short. A reference is an alias, or another name to an existing variable. They are mainly used for pass-by-reference, if the reference variable is passed

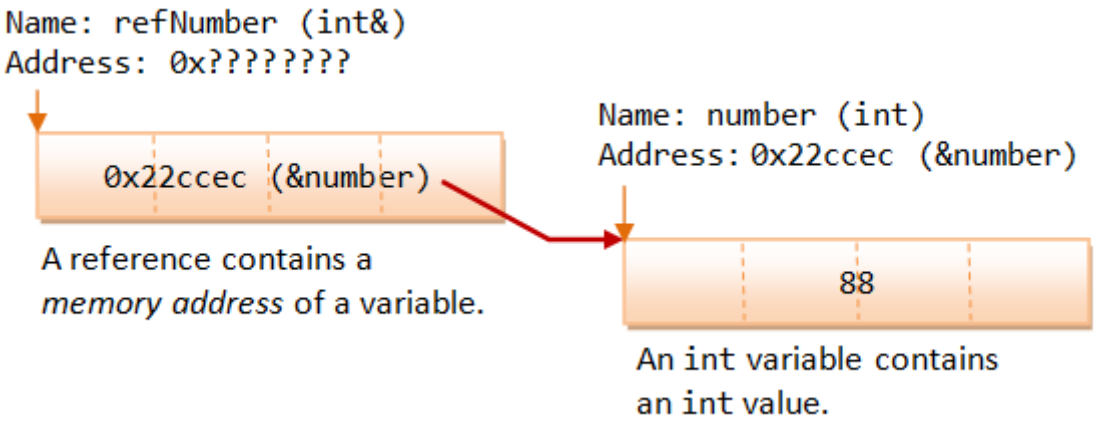
into the function, the function works on the original copy instead of a copy of the parameter. Changes to the parameter inside the function are reflected outside of the function.

To create a reference in C++:

```
type &newName = existingName;
```

How does references work?

References are implemented as a pointer!



The reference variables stores the address of the variable, much like a pointer, however there are key differences!

When you use pointer vs when you use reference: For pointer you have to explicitly dereference the pointer in order to get the value back. On the other hand, referencing and deferencing are done implicitly for you, so you don't need to dereference a reference variable for assignment, just treat it as a variable and assign to it!

In addition, references cannot be NULL and they must be initialized when they are declared, no separate declaration and assignment like pointer.

There is pointer arithmetic but there is no reference arithmetic.

Short summary

Pointer

- You think of pointer as just another type like int, char, float, it takes up a constant amount of spaces in memory, and you can assign values to them.
- The value you assign to them is a memory address. You can interpret the memory address pointed by the pointer using the dereferencing operator (*).
- A pointer can be assigned to point a `NULL` value
- A pointer can be changed to point to any variable of the same type.

- You can do pointer arithmetic, but you cannot with references.

References

- A reference must be initialized when it is declared
- It cannot be NULL
- You can use it by simply using the name. Reference is implemented via pointer, a constant pointer with automatic indirection, the compiler will apply the dereferencing for you automatically.
- Think of references as an alias to existing object in memory.

Little and Big Endian

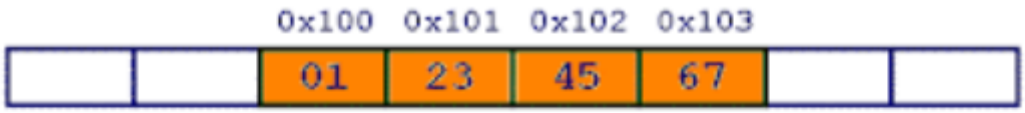
Endianness

Little and big endian are two ways of storing multibyte data-types into memory. For single byte data-types like a char, it doesn't matter what the endian because it is only one byte. Duh.

Let's assume that an integer is 4 bytes in the 32 bit system that we are working with in C.

Let's have the hexadecimal value **0x01234567** this is 32 bit remember every 2 hexadecimal digit is one byte, there is 4 pair of the hexadecimal digits hence 32 bits.

Now we will be storing it into memory and these are the two ways.



Big Endian

Little Endian

Big Endian

So we have our hexadecimal **0x01234567** when we are just writing the numbers the most significant bit is always on the left hand side, i.e. 01 in hex or 0000 0001 is the most significant byte.

If the machine is big endian then it will store each byte into memory as the way it is written:

- 01 into memory address 0x100
- 23 into memory address 0x101
- 45 into memory address 0x102
- 67 into memory address 0x103

In this example, the memory address increases.

Little Endian

With little endian we will be storing the least significant byte into the address and going backward:

- 67 into memory address 0x100
- 45 into memory address 0x101
- 23 into memory address 0x102
- 01 into memory address 0x103

Example program to show endian

```
void show_mem_rep(char *start, int n)
{
    int i;
    for (i = 0; i < n; i++)
         printf(" %.2x", start[i]);
    printf("\n");
}

/*Main function to call above function for 0x01234567*/
int main()
{
   int i = 0x01234567;
   show_mem_rep((char *)&i, sizeof(i));
   getchar();
   return 0;
}
```

It uses a char pointer to go through the integer and show the endian of the machine.

If it is big endian it will print out 01 23 45 67

If it is little endian it will print out 67 45 23 01

Do I need to care about endian?

Most of the time no, the compiler or the interpreter will usually take care of the endianness.

But when you work with network programming, i.e. sending data over a network socket, you will have to be cognisance of the endian by say converting the integer that you are sending over the wire into network bytes order (big endian), then converted back to host byte order (whether it is big endian or little endian).