Brown University ENGN2912B Fall 2017
Scientific Computing in C++
Lecture 5 | C++ Memory allocation: pointers and references

C++ representation of data in memory

In computer memory values are stored as a sequence of binary words of length defined by the architecture of the computer central processing unit (CPU). Today, a typical word length is 32 bits. Regardless of the CPU word size, from the programmer's point of view, memory can be considered to be a sequence of 8 bit *bytes* each accessed by an index variable called a *pointer*. For a 32 bit CPU architecture, the location of a byte in memory is indicated by a 32 bit pointer type. The value of a pointer type is called the *address*.

For an **`unsigned short`** value the number of bits allocated to each value is 16 bits, or two bytes. Consider the binary representation of an **`unsigned short`** data type for the value **`1000`**, as shown in Figure 1.

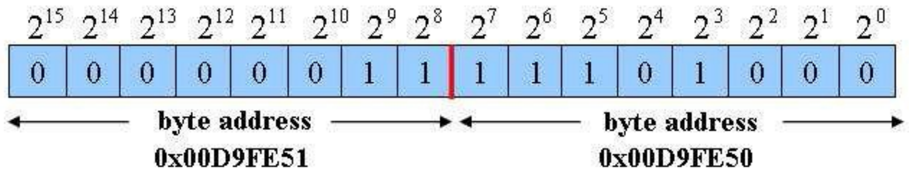

Figure 1 Binary representation of the decimal value 1000.

In hexadecimal notation, this 2 byte word has the value 0x03e8. In Visual Studio it is possible to display the memory contents as table. For the following program fragment,

```
unsigned short us = 1000;
```

The C++ compiler assigned the address location in memory for this variable to **`0x00D9FE50`**. Note that the pointer type is 32 bits long (eight hex digits) since the CPU is 32 bits. Program debuggers typically provide a tool to view the contents of memory. The contents in this case are shown below, and in Figure 1.

```
0x00D9FE50  e8
0x00D9FE51  03
```

It can be seen immediately that the more significant bits (representing larger values) are at a higher memory address than the least significant bits. This arrangement is called *little-endian*. If the most significant bits are stored first, the arrangement is called *big-endian*.

The words, "big-endian" and "little-endian" introduced by Jonathan Swift in his book, *Gulliver's Travels*, published in 1727. The "end" in the words referred to the two different political groups. The Big Endians place their boiled eggs in an egg cup, big end up. The Little Endians place the egg with the small end up. This apparently trivial difference magnified into violent political struggles. This aspect of human nature can be easily observed in today's international politics.

The "endian-ness" property is related to the hardware architecture of the CPU. Unfortunately, computer chip manufacturers have produced machines with both variants. The CPU used in this demonstration was an Intel Centrino and has little endian byte order. The Motorola 680x0 and Sun SuperSPARC^{TM} are Big Endian. Perhaps CPU manufacturers will converge to a single representation in the future.

This variation complicates the transfer of "binary" files between machines with different architectures, since a binary file is an exact copy of the memory contents corresponding to set of data, and preserves the byte order. The binary file contents will interpreted incorrectly on a CPU with the opposite byte order. In the example, with Big Endian encoding, the two bytes would be interpreted as **`0xe803`**, or the value 59395 instead of 1000.

As another example, consider the memory content for the following variable allocation.

```
float f = 0.1;
```

The memory contents for the storage of **`f`**, starting at address, **`0x00D9FE30`**, and incrementing by one byte each time, is

```
0x00D9FE30  cd
0x00D9FE31  cc
0x00D9FE32  cc
0x00D9FE33  3d
```

As a little-endian 32 bit value, the storage content is **`3dcccccd`**. This encoding is the representation of the 32 bit floating point number, with value 0.1, as described in Lecture 2.

<u>The pointer data type</u>

The C++ pointer data type is designed to allow the programmer to directly access and manipulate the contents of memory. A pointer variable is indicated by the symbol, **`*`**. The starting address of a given variable can be found with unary operator **`&`**, which returns a value of type **`pointer`**. For example, consider the following code.

```
float x = 0.1;
float* xp = &x;
```

The layout of the float variable **`x`** was just described, where the starting address is **`0x00D9FE30`**, which is the value assigned to **`xp`** by the statement, **`&x`**. That is, **`xp==0x00D9FE30.`**

Arithmetic operations also apply to variables of pointer type. Consider the memory contents shown in Figure 2.

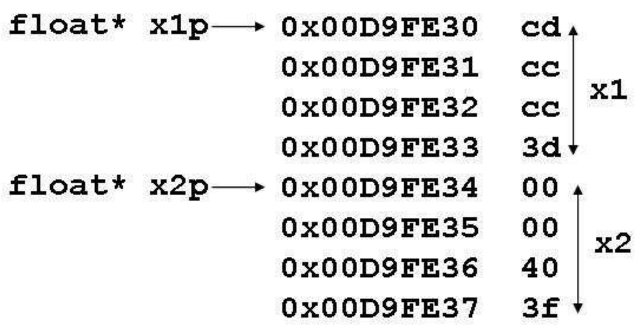

Figure 2 The memory layout for two float variables.

Suppose a new pointer variable is declared as follows.

```
float* x1p1 = x1p + 1;
```

The pointer **`x1p1`** will now point at the starting address of variable **`x2`**. That is, **`x2p==x1p1.`** The increment **`1`** is considered to be equivalent to **`sizeof(type)`**, or one storage unit for the variable of that type.

The behavior of the pointer data type is affected by the type of variable is being pointed to. The size of the **`char`** data type is one byte, so the memory address index increases by 1 for each subsequent **`char`** value as shown in shown in Figure 3.

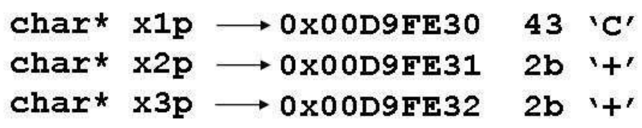

Figure 3 The layout of char variables in memory.

This time the calculation,

```
char* x1p1 = x1p + 1;
```

will increment the address by one byte and so **`x1p1==x2p`**. Similarly,

```
char* x1p2 = x1p + 2;
```

will yield **`x1p2==x3p`**.

Note that unary arithmetic operations apply as well. For example,

```
char* x1p1 = ++x1p;
```

produces the same result for the value of **`x1p1`** as before.

These examples reveal that C++ defines a matching set of real and integer pointer types that correspond, one to one, with the numerical data types, and provides a means for accessing and traversing their arrangement in memory. As will be seen in later chapters, the pointer type concept is even more general and can be assigned for any new data type created by the programmer.

The array

C++ gives the programmer explicit control over the contents of memory. A block of memory can be designated to hold a specified number of values of a given data type. Consider allocating a block of three unsigned **`char`** types. The syntax for defining an *array* is,

```
char a[5];
//give values to the array elements
a[0]='B'; a[1]='r'; a[2]='o'; a[3]= 'w'; a[4]='n';
```

The key syntactic operation is the square bracket syntax `[]`, which indicates the number of elements in the array. The specification, **`char`**, defines the type of the array elements. In this case, the size of the array is 5 elements, the characters of the word, “Brown”. A snapshot of the memory allocation for the array is shown below.

```
0x00D9FF58   42   B
0x00D9FF59   72   r
0x00D9FF5A   6f   o
0x00D9FF5B   77   w
0x00D9FF5C   6e   n
```

The `[]`declaration specifies that **`a`** is the staring address of the block of memory. Thus, the type of **`a`** must be a pointer. In this case, the data type of **`a`** is **`char*`**. The declaration must be of the form

```
type symbol [ unsigned-integer-constant-expression ];
```

The constant-expression defining the array size must be known at compile time, thus the programmer has to determine the size and it can’t be changed during operation.

It is possible to initialize the array contents as part of the declaration. For the previous example,

```
char a[5]={'B', 'r', 'o', 'w', 'n'};
```

will produce the same initial values to the elements of **`a`**.

Note that the `[ ]` operator also is used to access the elements of the array. It is important to note that the index for the first element of the array is **`0`**. In some programming languages, like FORTRAN, the array index of the first element is **`1`**. The syntax for indexing is the same as for the declaration. The general `[ ]` access syntax is,

```
type value = array_symbol[unsigned-integer-type];
```

The value of the array index, say **`i`**, can be varied at run time over the range **`0<=i<array_size`**. For example, the following program,

```
for(unsigned i = 0; i<5; ++i)
  cout << a[i];
cout << endl;
```

will print the following,

```
Brown
```

Note that the address operator applies to array elements as well. For the "Brown" example,

```
&a[3]==0x00D9FF5B
&a[2]+1==&a[3]
```

Reinterpret cast

It is sometimes necessary in C++ to interpret a value of one data type as another, when such an interpretation makes sense. A typical reinterpretation is to interpret an integer as a pointer. This reinterpretation only makes sense when the integer has the range of 0 to the maximum unsigned integer value of the address word used in the CPU architecture. The most common address word size is 32 bits in today's CPUs. Thus, a pointer and an unsigned int data type can be interpreted as each other properly. As an example,

```
short v = 10;
unsigned adr_v = reinterpret_cast<unsigned>(&v);
short* vp = reinterpret_cast<short*>(adr_v);
```

In this case, **`adr_v==0x00d9ff34`**, which is the integer, **`14286644`**. This integer value is converted back to the pointer data type by the second reinterpret_cast operation.

It is easy to misuse the **`reinterpret_cast`** operation. For example, consider the following character array.

```
unsigned char c[4]={'n','o','t','s'};
unsigned short* s = reinterpret_cast<unsigned short*>(c);
cout << s[0] << ' ' << s[1] << endl;
```

The printed result of this program is as follows.

```
28526 29556
```

Note that 28526 == 0x6f6e, and the ASCII characters 'n'==6e and 'o'=6f. Since this machine is Little Endian the first character in the array is matched to the least significant byte of the unsigned short. However, this result is basically nonsense. There is no obvious utility in interpreting an array of **`char`** as an array of **`unsigned short`**.

That having been said, it can be the case that since memory is segmented into bytes, one program function can produce a pointer of type **`unsigned char*`**, which represents a block of memory that can be interpreted differently depending on other program variables. For example the picture elements (pixels) of an image can sometimes be bytes and other times unsigned short values to convey a higher dynamic range. If the beginning of the image array is presented as a unsigned char pointer, then code to process the image can be more general if the **`reinterpret_cast`** is delayed until later stages of computation.

The dereference operator

The dereference operator is also indicated by a `'*'`. The effect of the dereference operator is to produce the *value* of the memory contents addressed by a pointer. For example,

```
int j = -10;
int* jp = &j;
int k = *jp;
```

The result is that **`k==-10`**.

Its use has also already been employed in defining a variable as a pointer type. Its syntax does make sense in that context. For example consider the two statements,

```
int* jp = &j;
int  *jp = &j;
```

These statements are identical but the second one places the dereference operator next to the variable being declared, **`jp`**, emphasizing that it must be dereferenced to obtain an **`int`** type value, thus **`jp`** can only be an **`int*`** pointer.

C-style string constant

Consider the standard C++ representation for a string constant.

```
char* st = "Brown";
for(unsigned i = 0; st[i]!=0; ++i)
 cout << st[i];
cout <<  endl;
```

The program output is,

```
Brown
```

The memory contents after **`st`** is declared is,

```
0x00407070  42  B
0x00407071  72  r
0x00407072  6f  o
```

```
0x00407073  77  w
0x00407074  6e  n
0x00407075  00
```

Note that the string constant "Brown" is encoded in memory as an array of **`char*`** values with a zero byte at the end of the string. This representation of a string was originally defined by the C language and is called a "c-style" string. The assignment has created an array of **`char`** values of size six, with the last element null, i.e. 0. This form is also called a null-terminated string.

Compact expressions with `*` and pointer.

It is possible to create very compact expressions in C++ using pointer arithmetic and dereferencing. As an illustration consider the following program.

```
char* st = "Brown";
char  copy[6];
char* copy_p=copy;
while(*copy_p++=*st++);
```

After the program executes the **`char`** array **`copy`** will have the same contents as the implicit **`char`** array, **`st`**. At the beginning of execution the pointer **`st`** has the value of the starting address of the string constant, "Brown". The pointer **`copy_p`** indicates the beginning of the array, copy. The while statement tests the value being assigned by the **`'='`** sign and if it is zero the loop is terminated. Both pointers are incremented just after the assignment statement executes. The loop termination happens when the null, **`0`**, character is encountered at the end of the string. The final null character is copied, since the assignment is done before testing the result.

Some programmers really love this compact syntax, but it does lead to code that is difficult to read. Probably there is not a great difference in execution throughput if a more readable form is used, such as follows.

```
char* st = "Brown";
char copy[6];
for(unsigned i = 0; i<6; ++i)
 copy[i]=st[i];
```

One thing to note is that after the first program, the pointer **`st`** does not point to the beginning of the string in memory. If another copy is to be made, say into a **`char`** array **`copy2`**, it is necessary to reset the pointer back to the beginning of the string constant, by executing the statement, **`st-=6;`**

The void pointer type

Consider the declaration,

```
void* a;
```

In C++ the **`void`** key word means either anything or nothing. In this case the **`void*`** pointer type means that **`a`** can point to the location of any variable type. However it is necessary to cast the pointer to a known type before any value can be accessed. If we know that the variable is of type **`unsigned`** then the following code will work sucessfully. If it is known that the pointer is of type **`char*`** then **`void*`** can be cast appropriately.

```
char* s = "I will be void soon";
void* v = reinterpret_cast<void*>(s);
//some other processing …
//later cast back to char*
char* c = reinterpret_cast<char*>(v);
```

This use of **`void*`** is when a number of different data types are to be exchanged between program modules, but the type is not known at compile time.

The **`new`** and **`delete`** operators

So far, the variables used in the programs have been all declared within the scope of a main program or within narrower scopes inside conditional expressions or loops. As soon as the program execution leaves a scope the variables declared inside the scope no longer exist. There are many cases where the programmer wants variables to persist throughout the time the application is running. Global persistence could be achieved by declaring all such variables at the top of the main program. However, this strategy makes it difficult to write structured modular code.

C++ provides an alternative memory for cases where persistence throughout execution is needed. This memory is called the *heap* or *free store* or *dynamic memory*. The memory used to hold values only within a scope is typically called the *stack.*

Variables can be continuously allocated in the heap until all available memory is exhausted. At that point the program will *crash*, indicating that there is insufficient memory. It is up to the programmer to ensure that this catastrophe is avoided by carrying out *memory management* procedures. The programmer is responsible to remove unneeded variables from the heap in order to free up memory for allocating new variables. For example, a program may be carrying out a series of iterations to solve a differential equation on a finite element mesh. At each iteration there may be variables allocated that are to be only valid during the iteration. These variables have to be removed from the heap at the end of the iteration to make room for new variables in the next iteration.

The creation of a variable on the heap is achieved by the **`new`** operator. The operator returns a pointer to the location in heap memory where the variable has been allocated. Examples of allocating heap variables follows.

```
unsigned short* hs = new unsigned short;
*hs = 5;
```

```
char* hst = new char[4];
hst = "C++";
```

The **new** operator merely allocates the memory as required by the specified data type. The programmer must then initialize the memory with values. In the first statement, space for an unsigned short, **hs**, is created. In the next statement, the value of this variable is assigned to 5.

The second case appears to be the same as the first but is actually incorrect[1]. When a string constant is declared such as **"C++"** , the compiler allocates an array in a location which is not a valid address range for operations such as **delete**. After the assignment, **hst = "C++";** the pointer **hst** no longer accesses the memory allocated on the heap by the new operation and a memory leak has occurred. The safest assigment is,

```
// the const keyword is discussed in a future lecture
const char* hst = "C++";
```

A common error is to assume that the pointer type is applied to all variables in a multiple declaration statement. For example,

```
unsigned short* hs = new unsigned short, t =1;
```

The type of **t** is **unsigned short** not **unsigned short***. This confusion can be reduced by putting the * next to the variable as follows.

```
unsigned short *hs = new unsigned short, t =1;
```

This format makes it clear that all variables in the list are of **unsigned short** type, but because **hs** is dereferenced in the assignment it is actually a pointer. In order to declare multiple pointers in the same statement, the * dereference operator must be applied to each pointer variable in the list, i.e.,

```
unsigned short *hs = new unsigned short, *t =&v;
```
where **v** has been previously assigend to some **unsigned short** value.

The statement,

```
char* hst = new char[4];
```

causes the allocation of an array on the heap. In this case an array of 4 **char** variables is allocated, but not initialized.

[1] This error was not noticed in previous classes. A student in fall 2010 tried to execute the example and found the error.

Now that variables have been allocated, the mechanism for removing them from the heap must be described. A variable is removed from the heap using the **`delete`** operator. The variables previously created are removed as shown in the following code.

```
delete hs;
delete [] hst;
```

Note that the **`new`** and **`delete`** format must be matched. That is if a variable is created using the **`[]`** operator it must be deleted using the **`[]`** operator. **`delete`** must be passed a valid pointer to a location in heap memory in order to execute successfully. It is also possible to pass the pointer value **`0`** in which case no operation is performed. In general, the **`delete`** operation marks the heap memory as unused and it can be allocated to new variables.

What will happen if an attempt is made to access already deleted heap memory?

```
unsigned short* hs = new unsigned short;
*hs = 5;
delete hs;
unsigned short bad = *hs;
```

After allocation on the heap, **`hs`** has the memory address, **`0x00256448`**. The contents of the memory is,

```
0x00256408  05
0x00256409  00
```

as expected for the storage of an unsigned short. After the delete operator is applied the contents of heap memory is,

```
0x00256408  ee
0x00256409  fe
```

This content is nonsense and will lead to erroneous use of the variable “bad” in subsequent computations.

Another tragic situation arises if the programmer attempts to delete memory allocated on the stack. For example,

```
char a[4];
char* ap = a;
delete ap;
```

In this case, the program will immediately crash with an exception when the **`delete`** operation is executed.

A third unfortunate situation is the attempt to delete heap memory that has already been deleted.

```
unsigned short* hs = new unsigned short;
*hs = 5;
delete hs;
delete hs;
```

This program will also crash immediately on the second **`delete`**. If there is some danger of multiple deletions, the programmer can prevent crashing by setting the pointer to 0, as follows.

```
delete hs;
hs=0;
delete hs;//will not cause a crash.
```

<u>The memory leak</u>

The **`new`** and **`delete`** operators provide a mechanism for the programmer to manage memory allocation for program variables. However, it is easy to make programming errors that leaves memory allocated on the heap which cannot be deallocated. Consider the following program.

```
bool test = true;
if(test)
{
  char* a = new char[26];
  a = "I am soon to be an orphan";
 …
 // other computation but no delete
 …
} // a is now a memory leak
```

The local variable **`a`** is not visible outside the scope of the if condition. Indeed, **`a`** no longer exists since the stack allocation for variables within the scope is removed after the scope is exited. However, the memory allocated for **`a`** on the heap still exists and still contains the string characters assigned to it. There is now no way to **`delete`** the memory since there is no pointer to its location in the heap. This condition is called a *memory leak*.

The existence of memory leaks is difficult to detect and if they occur inside loops that are executed many times during operation of the application, the heap can gradually be exhausted and finally result in a crash. Therefore C++ programmers have developed special techniques to prevent memory leaks as will be described in a later chapter.

<u>Creating multi-dimensional arrays</u>

The basic C++ syntax allows for the declaration of a one-dimensional array of a given data type, as shown in previous examples. If it is necessary to represent an array of two or more dimensions, then additional program steps are necessary. It is typically the case that the programmer will want to create the array on the heap so that the data can be passe d to other functions. The following code illustrates the allocation and assignment of a two-dimensional array.

```
// allocate the array
unsigned short rows = 10, cols = 20;
int** a2d = new int*[rows];
for(unsigned r = 0; r<rows; ++r)
  a2d[r] = new int[cols];
// assign values to the array
for(unsigned r = 0; r<rows; ++r)
  for(unsigned c = 0; c<cols; ++c)
    a2d[r][c] = (r+1)*c;
```

Note that type of pointer to the start of the array is **`int**`**. That is a pointer to an array of pointers. In order to get at an element of the array it is necessary to apply two deference operations. In the program just above, access is achieved by double application of the **`[]`** operation, i.e., **`a2d[r][c]`**. The array is filled with values generated by the expression **`(r+1)*c`** , just for illustration. Note that the array element value can be either assigned or accessed by the **`[][]`** operator. The value of an array element can also be accessed by direct use of the dereference operator, i.e., **`a2d[r][c] == *(*(a2d + r)+c)`**.

The array is deleted in a similar but reverse manner to its allocation.

```
// deallocate the array
for(unsigned r = 0; r<rows; ++r)
  delete [] a2d[r];
delete [] a2d;
```

The deallocation proceeds by first removing the storage for the columns and then the row array of pointers.