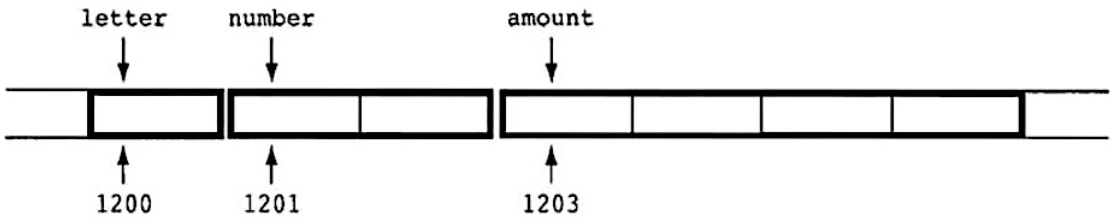
Pointers
Memory Allocation

Variable address

- `&` = address operator, returns the address
- `int a; &a` = address of a
- `int A[10]; &A[3]` = address of the 4-th element in A
- `char letter;` // address 1200
- `short number;` // address 1201
- `float amount;` // address 1203



Pointer = address variable

pointer variable = holds as value a memory address

i.e. ptr = 0x7fff5fbff564, ptr = 0x7fff5fbff568

memory addresses identify specific locations in the memory

pointers hold the address or location of other data

in other words, "points" to some piece of data

Pointers in Memory

- pointers have their own address
  - like any variable
  - pointer address not important for this class
- we care about the pointer value
  - which is the address of other data

Using pointer variables

- & (variable) = address/pointer of that variable
- * (pointer) = value stored at that address/pointer

Address operator in use

```
int num;  // creates an int variable called num

int *ptr; // ptr is the address of an int variable

ptr = &num; // ptr is assigned the address of num
```

Indirection operator in use

```
int num = 100; // num is initialized to 100

int *ptr = &num; // ptr points to num

cout << *ptr; // prints out the value of num

*ptr = 200; // num is now 200

*ptr += 100; // num is now 300
```

Arrays and Pointers

- array names are constant pointers
- `int nums[10]` declares an array of ints of size 10;
- `nums` works like a pointer to int which holds the starting address of the array
- Note however that `nums` is a constant pointer - its value cannot be changed to another address

Pointers as arrays

pointers can be used as array names

```
int nums[3] = {1, 2, 3};

int *ptr = nums;

cout << ptr[0]; // prints nums[0]
```

`ptr[1]` accesses `nums[1]` and `ptr[2]` accesses `nums[2]`

```
int * ptr[100] //declares an array of pointers
```

Comparing pointers

if one address comes before another in memory, the first one is considered "less than" the second

C++ relational operators `>, <, ==, !=, >=, <=` can be used to compare pointers

e.g., `&nums[0] < &nums[1]`.

Pointers as function arguments

a pointer can be used as a function parameter

a pointer parameter gives the function access to the location the pointer is pointing to

changes to the value "pointed" are reflected in the original call variable

Function declaration

```
void function(int *);
```

declares a function called foo that takes a pointer to integer as parameter

```
void function(int *ptr);
```

also valid but the formal name will be ignored

Function definition

```
void foo(int *ptr)  {

   *ptr += 100; // dereferences ptr, add 100

}
```

Note that += operates on the variable pointed to by ptr, this statement adds 100 to this original variable

Pointer and reference parameters

reference parameters give the function access to the actual argument but "hides" all the mechanics of dereferencing/indirection

pointer parameters are passed by value, but by dereferencing them using the indirection operator, the function gets access to the original variable that the pointer points to

Constant pointers

- constant pointer : constant address, variable value
  - `int * const ptr`
- pointer to constant : variable address, constant value
  - `const int * ptr`
  - a good idea for function declaration, to prevent the function to change the argument value
  - `void myfunction (const double * x)`

Dynamic Memory Allocation

Why dynamic memory allocation

if we know the number of variables needed in a program, we can define them up front

we need an array of size 24 to store the scores of 24 students

what do we do when we don't know how many variables are needed?

a program to store the scores of any number of students.

programs must be allowed to create variables on the fly - during runtime

Dynamic memory allocation

Variables can be created or destroyed while a program is running.

A program, while running, can put a request for a chunk of memory to hold a variable of a particular data type and can access the newly allocated memory through its address.

This is called dynamic memory allocation.

new operator

in C++, dynamic memory allocation is done using the new operator.

```
int *ptr;

ptr = new int;

*ptr = 100; *ptr += 1; cout << *ptr;
```

Dynamically allocating arrays

Not much point in dynamically allocating a single variable - the new operator can also be used to dynamically create an array.

```
int *ptr;

ptr = new int[100];

for(int i = 0; i < 100; i++) ptr[i] = 0;
```

`delete` operator

When a program is finished using dynamically allocated memory, it should release it.

The `delete` operator is used to free memory allocated with new operator.

```
delete ptr; // for single variable

delete [] ptr; // for array
```

malloc(), calloc(), free()

- `void * p = malloc ( 300 );` // allocates 300 bytes, returns the head address
- `void * p = calloc ( num, size );` //allocates num*size bytes, initializes with 0, returns head address
- `free (p);` //releases the memory BLOCK at address p, previously allocated with `malloc` or `calloc`
- prefer to use `new` and `delete`, whenever possible

Functions returning pointers

Functions can return pointers, but the item the pointer references must still exists after the function ends

A function can return a pointer only if it is

- a pointer to an item that was passed into the function as an argument
- a pointer to a dynamically allocated chunk of memory (see dynamic memory alloc.)

Pointers to access array elements

```
int nums[3] = {1, 2, 3};
```

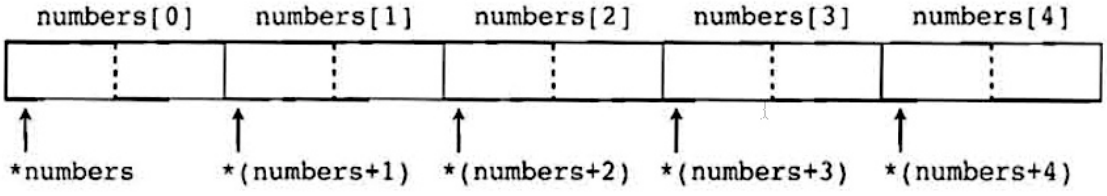
`*nums` accesses the first array element, same as `nums[0]`

- `*(nums + 1)` accesses the second element, same as `nums[1]`
- `*(nums + 2)` same as `nums[2]`

`arr_name[i]` same as `*(arr_name + i)`

- expr `*(arr_name + i*sizeof(<datatype>))`

```
short numbers[10] // sizeof(short)=2
```



Pointer arithmetic

pointers can be added and subtracted; multiplication and division are not allowed

use of ++, --, +=, and -= operators are allowed

note that adding a number to a pointer actually adds the number times the sizeof(type)

`ptr+3` means `ptr+3*sizeof(int)`

`*(ptr+3)` means value stored at address `ptr+3*4`, which is the same as `ptr[3]`

`(ptr+1)[2]` same as `ptr[3]`

Type Casting

- `int * ptr = new int; *p=21;`
- `short* p2 = ptr;` *// same address, only 2 bytes*
- `cout<< *p2;` *//still displays 21 -WHY?*
- LITTLE ENDIAN : significant bits last (to the right)
- BIG ENDIAN : significant bits first (to the left)
- Cool indeed, but very easy to make mistakes!

Bitwise operators

- shift operator (>>)
- bitwise "and" (&): 72 & 184 = 8
  - `01001000 &`
  - `10111000 =`
  - `--------`
  - `00001000`
- bitwise "or" (|)
  - `01001000 |`
  - `10111000 =`
  - `--------`
  - `11111000`