Pointers and the C++ Memory Model
Variables and Memory

- Each variable in a program is stored in a block of memory.
- The block of memory that stores a variable's value has three attributes:
  1. Size - how big is it?
  2. Address - where is it?
  3. Value - what does it contain?

```
int 01101001 01011100 11001100 00011001
```

\[0xf601be72\]
sizeof Operator - How big is it?

- The sizeof operator tells you how many bytes of memory are needed to store a particular variable or data type.

```cpp
struct Student { long id; string name; };

Student s;
Student t[10];

int longSize = sizeof(long);
int stringSize = sizeof(string);
int studentSize = sizeof(Student);

int idSize = sizeof(s.id);
int nameSize = sizeof(s.name);
int sSize = sizeof(s);
int tSize = sizeof(t);
```
& Operator - Where is it?

- The & operator returns the memory address at which the operand is stored
- In C++, address values are called "pointers"

```cpp
struct Student { long id; string name; };

Student s;
Student t[10];

Student * sAddr = &s;
cout << "s is at address " << sAddr << endl;

Student * elemAddr = &t[4];
cout << "t[4] is at address " << elemAddr << endl;

long * idAddr = &s.id;
cout << "s.id is at address " << idAddr << endl;
```
* Operator - What does it contain?

- The * operator returns the value pointed to by a pointer
- This is called "dereferencing" the pointer
- Result of * can be used as an l-value or r-value

```c
struct Student { long id; string name; };

int x = 100;
int * xAddr = &x;
int xCopy = *xAddr;
*xAddr = 212;

Student s = {12345L, "fred"};
Student * sAddr = &s;
Student sCopy = *sAddr;
string nameCopy = (*sAddr).name;
(*sAddr).name = "barney";
```
The -> Operator

- When you have a pointer to a structure, the syntax for referencing a member of the structure is (*p).member
- The -> operator provides a more compact syntax for doing the same thing

```c
struct Student { long id; string name; };

Student s = {12345L, "fred"};
Student * sAddr = &s;

long idCopy = sAddr->id;
sAddr->id = 98765L;

string nameCopy = sAddr->name;
sAddr->name = "barney";
```
Arrays and Pointers

- The name of an array (without a subscript) evaluates to the address of the array.
- If the array elements are of type X, the type of this expression is X * (i.e., pointer to X).
- The address of an array is the same as the address of its first element.
- Any pointer can be indexed like an array.

```c
short data[100];

short * addr1 = data;
short * addr2 = &data[0];
// (addr1 == addr2)

short s = addr1[32];
addr1[32] = -50;
```
Pointer Arithmetic

- Pointer values can be compared using relational operators: `==, !=, <, <=, >, >=`
  
  ```
  if (p1 < p2) {...}
  ```

- The `++` operator can be used to move a pointer forward one position in memory
  
  - If `p` has type `X *`, `++p` adds `sizeof(X)` to `p`, not 1

- The `--` operator can be used to move a pointer backward one position in memory
  
  - If `p` has type `X *`, `--p` subtracts `sizeof(X)` from `p`, not 1
Pointer Arithmetic

- The `+` and `+=` operators can be used to move a pointer forward \( n \) positions in memory
  - \((p + n)\) adds \( n \times \text{sizeof}(X)\) to \( p \), not \( n \)

- The `-` and `-=` operators can be used to move a pointer backward \( n \) positions in memory
  - \((p - n)\) subtracts \( n \times \text{sizeof}(X)\) from \( p \), not \( n \)

- The `-` operator can be used to subtract one pointer from another
  - \((p - q)\) returns the number of array elements (not bytes) between \( q \) and \( p \)
Let's rewrite this code using pointer arithmetic

```c
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
int i = 0;
while (i < 5) {
    sum += data[i];
    ++i;
}
```

```c
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
```
Pointer Arithmetic

short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
Pointer Arithmetic

```
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
```
```c
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
```
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
Pointer Arithmetic

short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}

12 4 22 43 9

sum

38
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
Pointer Arithmetic

short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}

12
4
22
43
9

81
Pointer Arithmetic

short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
Pointer Arithmetic

```c
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}
```

```plaintext
data
12 4 22 43 9
```

```plaintext
sum
90
```
Pointer Arithmetic

short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}

12  4  22  43  9

data

90

sum

end cur
Null Pointers

- A pointer with value 0 (zero) is called a "null pointer"
- A null pointer doesn't point to anything
  ```c
  char * ptr = 0;
  ```
- Dereferencing a null pointer is a fatal error
  ```c
  // assume that p1 and p2 are null
  *p1 = 'X';         // disaster!
  p2->name = "fred"; // disaster!
  ```
The C++ Memory Model

- A C++ program's address space is divided into several different areas
  - Code
  - Static data
  - Runtime stack
  - Heap
Parameters and Local Variables

- Parameters and local variables are pushed onto the runtime stack when a function is called, and popped off the stack when the function returns.
- Never use the address of a parameter or local variable after the function returns.

```cpp
Student * CreateStudent() {
    Student student;
    Student * s = &student;
    s->id = 0L;  // ok
    s->name = "";  // ok
    return s;  // disaster!
}

int main() {
    Student * s = CreateStudent();
    cout << "id: " << s->id << ", name: " << s->name << endl;
    return 0;
}
```
Global Variables

- Global variable - a variable that is declared outside of any function or class
- Unlike local variables, global variables are created once and are available as long as the program executes
- Global variables are stored in the static data area of the address space
Dynamic Memory Allocation

- Programs can dynamically allocate memory from the heap
- The new operator is used to allocate heap memory
- The delete operator is used to free heap memory
- Heap memory should be freed whenever possible so that the program won't run out of memory

```cpp
Student * CreateStudent() {
    Student * s = new Student;
    s->id = 0L;       // ok
    s->name = "";    // ok
    return s;        // ok
}

int main() {
    Student * s = CreateStudent();
    cout << "id: " << s->id << ", name: " << s->name << endl;
    delete s;
    return 0;
}
```
Dynamic Memory Allocation

- Use [] when allocating and deallocating arrays

```cpp
Student * CreateStudentArray(int n) {
    Student * s = new Student[n];
    for (int x=0; x < n; ++x) {
        s[x].id = 0L;
        s[x].name = "";
    }
    return s;
}

int main() {
    int number;
    cout << "How many students? ";
    cin >> number;
    Student * s = CreateStudentArray(number);
    for (int x=0; x < number; ++x) {
        cout << "id: " << s[x].id << ", name: " << s[x].name << endl;
    }
    delete [] s;
    return 0;
}
```