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?

<table>
<thead>
<tr>
<th>int</th>
<th>01101001</th>
<th>01011100</th>
<th>11001100</th>
<th>00011001</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf601be72</td>
<td>4 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```c
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
// simple integer copy
int x = 100;
int y = x;
x = 212;

// do the same thing with pointers
int x = 100;
int * xAddr = &x;
int y = *xAddr;
*xAddr = 212;
```
* Operator - What does it contain?

**A structure example**

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

// simple structure operations
Student s = {12345, "fred"};
Student t = s;
string n = s.name;
s.name = "barney";

// do the same thing with pointers
Student s = {12345, "fred"};
Student * sAddr = &s;
Student t = *sAddr;
string n = (*sAddr).name;
(*sAddr).name = "barney";
```
The \textbf{-}-> Operator

- When you have a pointer to a structure, the syntax for referencing a member of the structure is\((*p).\text{member}\)

- The \textbf{-}-> operator provides a more compact syntax for doing the same thing

\begin{verbatim}
// ugly syntax
Student s = {12345, "fred"};
Student * p = &s;
string n = (*p).name;
(*p).name = "barney";

// nicer syntax
Student s = {12345, "fred"};
Student * p = &s;
string n = p->name;
p->name = "barney";
\end{verbatim}
Arrays and Pointers

- The name of an array (without a subscript) evaluates to the address of the array
- The address of an array is the same as the address of its first element
- Any pointer can be indexed like an array (even if it doesn’t point to an array)

```c
short data[100];

short * p1 = data;
short * p2 = &data[0];
// (p1 == p2)

short s = p1[32];
p1[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
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;
}
```
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;
}

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;
}
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;
}

data

12  4  22  43  9

cur

end

sum

0
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;
}
```

![Diagram showing data array and pointer movements](image-url)
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}

16
short data[5] = {12, 4, 22, 43, 9};
long sum = 0;
short * end = (data + 5);
short * cur = data;
while (cur < end) {
    sum += *cur;
    ++cur;
}

sum 16

data 12 4 22 43 9

cur

end
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;
}

![Diagram of 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;
}

Pointer Arithmetic

```
data
  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;
}
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;
}

sum 81

data

12 4 22 43 9

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

```
data
12  4  22  43  9

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

- A pointer with value 0 (zero) is called a "null pointer"
- A null pointer doesn't point to anything

    char * ptr = 0;

- Dereferencing a null pointer is a fatal error

    // 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
  - Heap
  - Runtime stack

- Maximum sizes of heap and stack can be set using `ulimit` before running program
  - `ulimit -d #kb`
  - `ulimit -s #kb`
Static Variables

- Stored in static data area
- Allocated when program is loaded, never deallocated
- Initialized by compiler to all zeros (guaranteed by C++)

Kinds of static variables

- Global variables
  - variables declared outside of any function or class
- Static variables inside a class
  - all instances of the class share one instance of the variable
- Static local variables
  - local variables that retain their values between function invocations because they’re not on the runtime stack
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.

```c
int f(char a, int b, char c) {
    char * p;
    float q, r;
    ...
}

r = f('q', 3, '?');
```
Parameters and Local Variables

Never use the address of a parameter or local variable after the function returns

Student * CreateStudent(long id, string name) {
    Student s;
    s.id = id;    // ok
    s.name = name;  // ok
    return &s;   // disaster!
}

int main() {
    Student * a = CreateStudent(4978L, "Fred");
    Student * b = CreateStudent(3925L, "Barney");
    cout << "Fred's ID: " << a->id << endl;
    return 0;
}
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(long id, string name) {
    Student * s = new Student;
    s->id = id;  // ok
    s->name = name; // ok
    return s;    // ok
}

int main() {
    Student * a = CreateStudent(4978L, "Fred");
    Student * b = CreateStudent(3925L, "Barney");
    cout << "Fred's ID: " << a->id << endl;
    delete a;
    delete b;
    return 0;
}
```
Dynamic Memory Allocation

- Use [] when allocating and deallocating arrays

```c++
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);

    // use student array for something ...

    delete [] s;
    return 0;
}
```
Runtime stack vs. Heap

- Runtime Stack:
  - Memory is automatically allocated/deallocated by the compiler (easy for programmer)
  - Allocation/deallocation is very fast (just move the stack pointer)
  - Stack has a limited size, much smaller than heap (although this can be changed)
  - Stack can’t be used to store dynamic data structures (e.g., linked list, BST, array whose size isn’t known until runtime, etc.)
  - Programmer has no control over variable’s lifetime (when subroutine exits, variable is popped no matter what)

- Heap:
  - Programmer must call new and remember to call delete (more work)
  - New and delete are expensive operations, much slower than adjusting stack pointer
  - Heap is normally much larger than the stack
  - Dynamic data structures must be heap allocated
  - Programmer completely controls the time of birth and death of an object