Pointers

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: February 6, 2025

Directly Managing Memory

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array
  - Indicies between `0` and `2^64`

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array
    - Indicies between `0` and `2^64`
- With array indices, we could:

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array
    - Indicies between `0` and `2^64`
- With array indices, we could:
    - Get the element at a certain index

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array
    - Indicies between `0` and `2^64`
- With array indices, we could:
    - Get the element at a certain index
    - Set the element at a certain index

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array
  - Indicies between `0` and `2^64`
- With array indices, we could:
  - Get the element at a certain index
  - Set the element at a certain index
  - Do arithmetic on the indices

The C++ Memory Model

- We can imagine RAM/Memory as being like a giant array
    - Indicies between `0` and `2^64`
- With array indices, we could:
    - Get the element at a certain index
    - Set the element at a certain index
    - Do arithmetic on the indices
- Pointers let you do this *for all of memory*

Pointers vs. References

- Reference = alias for another variable

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory
- Like references, but much more powerful

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory
- Like references, but much more powerful
    - can be initialized to anything!

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory
- Like references, but much more powerful
  - can be initialized to anything!
  - can change over time (unlike references)

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory
- Like references, but much more powerful
    - can be initialized to anything!
    - can change over time (unlike references)
    - can do pointer arithmetic

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory
- Like references, but much more powerful
    - can be initialized to anything!
    - can change over time (unlike references)
    - can do pointer arithmetic
- We'll show examples of initialization, the & operator, and dereferencing (the `*` operator)

Pointers vs. References

- Reference = alias for another variable
- Pointer = address of another variable stored elsewhere in memory
- Like references, but much more powerful
  - can be initialized to anything!
  - can change over time (unlike references)
  - can do pointer arithmetic
- We'll show examples of initialization, the & operator, and dereferencing (the `*` operator)
  - x vs. &x vs. `*x`

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
    - Produces a pointer from a variable

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- `*` called *dereferencing*, gets the contents of the address

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- `*` called *dereferencing*, gets the contents of the address
  - Produces a value from a pointer

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
    - Produces a pointer from a variable
- `*` called *dereferencing*, gets the contents of the address
    - Produces a value from a pointer

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- `*` called *dereferencing*, gets the contents of the address
  - Produces a value from a pointer

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- `*` called *dereferencing*, gets the contents of the address
  - Produces a value from a pointer

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- * called *dereferencing*, gets the contents of the address
  - Produces a value from a pointer

```cpp
int *x, *y, p, q;
// vs. int* x, y, p ,q;

p = 5;
q = 6;

x = &p;
y = &q;

if(x==y){
  cout << "Hello";
  cout << "\n";
 }
```

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- * called *dereferencing*, gets the contents of the address
  - Produces a value from a pointer

```
int *x, *y, p, q;
// vs. int* x, y, p ,q;

p = 5;
q = 6;

x = &p;
y = &q;

if(x==y){
  cout << "Hello";
  cout << "\n";
 }
```

Address-Of and Dereferencing

- (A)mpersand for (A)ddress-of a variable
  - Produces a pointer from a variable
- * called *dereferencing*, gets the contents of the address
  - Produces a value from a pointer

```
int *x, *y, p, q;
// vs. int* x, y, p ,q;

p = 5;
q = 6;

x = &p;
y = &q;

if(x==y){
  cout << "Hello";
  cout << "\n";
 }
```

```
x = y;
cout << *x << "\n";

x = &p;
cout << *x << "\n";

*x = *y;
cout << *x << "\n";
cout << *&*x << "\n";
cout << **&x << "\n";
```

Review: Pass by Value

- Pass by value

Review: Pass by Value

- Pass by value
  - Arguments evaluated then copied

Review: Pass by Value

- Pass by value
    - Arguments evaluated then copied
    - Can be any value, not just variable

Review: Pass by Value

- Pass by value
    - Arguments evaluated then copied
    - Can be any value, not just variable

Review: Pass by Value

- Pass by value
  - Arguments evaluated then copied
  - Can be any value, not just variable

```
void doubleV(int a){
  a = a*2;
}

int main( ){
  int a = 2;
  doubleV(a+a);
  cout << a << endl;

  return 0;
}
```

Review: Pass by Reference

- Pass by reference

Review: Pass by Reference

- Pass by reference
    - Argument *must* be a variable

Review: Pass by Reference

- Pass by reference
    - Argument *must* be a variable

Review: Pass by Reference

- Pass by reference
  - Argument *must* be a variable

```
void doubleR(int &a){
  a = a*2;
}

int main() {
  int a = 4;
  doubleR(a);
  cout << a << endl;

  return 0;
}
```

New: Pass by Address

- Call by address

New: Pass by Address

- Call by address
    - Arguments evaluated and copied

New: Pass by Address

- Call by address
    - Arguments evaluated and copied
    - But the thing that’s copied is an address in memory

New: Pass by Address

- Call by address
    - Arguments evaluated and copied
    - But the thing that’s copied is an address in memory
- Must explicitly dereference to get/set value at that address

New: Pass by Address

- Call by address
  - Arguments evaluated and copied
  - But the thing that’s copied is an address in memory
- Must explicitly dereference to get/set value at that address

New: Pass by Address

- Call by address
  - Arguments evaluated and copied
  - But the thing that’s copied is an address in memory
- Must explicitly dereference to get/set value at that address

```
void doubleP(int *a){
  *a = (*a)*2;
}

int main( ){
  int a = 4;
  doubleP(&a);
  cout << a << endl;

  return 0;
}
```

Another example: Swap

Another example: Swap

Another example: Swap

- Call by address

Another example: Swap

- Call by address

Another example: Swap

- Call by address

```
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main( ){
  int a = 4, b = 6;
  swap(&a, &b);
  cout << a << endl;
  return 0;
}
```

Another example: Swap

- Call by address

```
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main( ){
  int a = 4, b = 6;
  swap(&a, &b);
  cout << a << endl;
  return 0;
}
```

- Call by reference

Another example: Swap

- Call by address

```
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main( ){
  int a = 4, b = 6;
  swap(&a, &b);
  cout << a << endl;
  return 0;
}
```

- Call by reference

Another example: Swap

- Call by address

```
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main( ){
  int a = 4, b = 6;
  swap(&a, &b);
  cout << a << endl;
  return 0;
}
```

- Call by reference

```
void doubleR(int &a, int &b){
  int temp = a;
  a = b;
  b = temp;
}

int main() {
  int a = 4, b = 6;
  swap(a, b);
  cout << a << endl;
  return 0;
}
```

Constant pointers

- Can get value at address, but not set

Constant pointers

- Can get value at address, but not set

Constant pointers

- Can get value at address, but not set

Constant pointers

- Can get value at address, but not set

Constant pointers

- Can get value at address, but not set

```
struct BigRecord {
  ...
};

void f(const BigRecord *pRec1){
  ...
    BigRecord pRec2;
  ...
    *pRec1 = *pRec2; // Wrong!
  pRec1 = pRec2;   // No issues
}
```

Constant pointers

- Can get value at address, but not set

```
struct BigRecord {
  ...
};

void f(const BigRecord *pRec1){
  ...
    BigRecord pRec2;
  ...
    *pRec1 = *pRec2; // Wrong!
  pRec1 = pRec2;   // No issues
}
```

Constant pointers

- Can get value at address, but not set

```
struct BigRecord {
  ...
};

void f(const BigRecord *pRec1){
  ...
    BigRecord pRec2;
  ...
    *pRec1 = *pRec2; // Wrong!
  pRec1 = pRec2;   // No issues
}
```

```
int main(){
  BigRecord x;
  ...
    f(&x);
  ...
    }

// Note: f( ) can't change x
// but f( ) can change pRec1!

// vs. BigRecord const *pRec1
```

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array
    - This is why functions don’t need the array dimensions in the type

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array
  - This is why functions don't need the array dimensions in the type
- Array element `A[i]` just adds `i` to the pointer A.

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array
    - This is why functions don't need the array dimensions in the type
- Array element `A[i]` just adds `i` to the pointer A.
    - Need the 2nd dimension size to do offset calculation for 2D array

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array
    - This is why functions don't need the array dimensions in the type
- Array element `A[i]` just adds `i` to the pointer A.
    - Need the 2nd dimension size to do offset calculation for 2D array
- This is also why arrays are always pass-by-reference

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array
    - This is why functions don't need the array dimensions in the type
- Array element `A[i]` just adds `i` to the pointer A.
    - Need the 2nd dimension size to do offset calculation for 2D array
- This is also why arrays are always pass-by-reference
    - The value of an array *is* its start location in memory, so copying an array just copies its address

Arrays are just pointers

- In C++, arrays are just pointers to the start of the array
  - This is why functions don't need the array dimensions in the type
- Array element `A[i]` just adds `i` to the pointer A.
  - Need the 2nd dimension size to do offset calculation for 2D array
- This is also why arrays are always pass-by-reference
  - The value of an array *is* its start location in memory, so copying an array just copies its address
  - So the resulting behaviour is pass-by-reference

Array Example

- Example:

Array Example

- Example:

Array Example

- Example:

```
int main(){
int A[5] = {1, 2, 3, 4, 5};
int i = 3;
cout << A[i] << endl; //Adds i to address A

// Also works, but is terrible
cout << i[A] << endl; //Adds A to address i, same result
}
```

```
4
4
```

Arrays and pointers example

Arrays and pointers example

Arrays and pointers example

Arrays and pointers example

```
// Every array variable can be
// used as a pointer to the first
// membe of the array
// (with certain restrictions)

int sumArray(int A[],
             unsigned int n){
  int sum = 0;
  for (int i = 0; i < n; i++){
    sum += A[i];
  }
  return sum;
}
```

Arrays and pointers example

```c
// Every array variable can be
// used as a pointer to the first
// membe of the array
// (with certain restrictions)

int sumArray(int A[],
             unsigned int n){
  int sum = 0;
  for (int i = 0; i < n; i++){
    sum += A[i];
  }
  return sum;
}
```

Arrays and pointers example

```c
// Every array variable can be
// used as a pointer to the first
// membe of the array
// (with certain restrictions)

int sumArray(int A[],
             unsigned int n){
  int sum = 0;
  for (int i = 0; i < n; i++){
    sum += A[i];
  }
  return sum;
}
```

```c
// This is why arrays are
// passed by references
// (by default)

// Array as pointer

int sumArray(int *A,
             unsigned int n){
  int sum = 0;
  for (int i = 0; i < n; i++){
    sum += A[i]; // or, *(A+i)
  }
  return sum;
}
```

Arrays using pointers

Arrays using pointers

```cpp
int A[5] = {1, 5, 10, 15, 20};

cout << A[0];
cout << *(A+0);
cout << *A;
cout << *(A+3);
cout << *A+3;
cout << *(A+3)+3;

A++; // Wrong!

// But this works!
int *B = A; // or int *B = &(A[0]);
B++; // line 11 (see below)
cout << *B;
// compiler automatically increments
// it to the proper location depending
// on the type of data B is pointing to,
// e.g. multiples of 4 for int/float and 8
// for double, etc.
```

C-strings and pointers

- C-string are just arrays of characters with the special `\0` at the end

C-strings and pointers

- C-string are just arrays of characters with the special `\0` at the end
  - Unlike c++ `string` which is a class

C-strings and pointers

- C-string are just arrays of characters with the special `\0` at the end
    - Unlike c++ `string` which is a class

C-strings and pointers

- C-string are just arrays of characters with the special `\0` at the end
  - Unlike c++ `string` which is a class

```
// computing the length of string #1
unsigned int cstringLength(const char s[]) {
  unsigned int length = 0;
  while (s[length] != '\0')
    length++;
  return length;
}

// computing the length of string #2
unsigned int cstringLength(const char *s) {
  unsigned int length = 0;
  while (*(s + length) != '\0')
    length++;
  return length;
}
```

C-strings and pointers (cont’d)

C-strings and pointers (cont’d)

```
// computing the length of string #3
// how it is actually implemented!

unsigned int cstringLength(const char *s) {
  const char *p = s;
  while (*p != '\0')
    p++;
  return p - s; // pointer difference
}
```

- In general, given two pointers p and q of the same type, (p - q) is the integer that can be added to p to obtain q.

Copying Strings using Arrays

Copying Strings using Arrays

```
// string copy using c-string

void cstringCopy(char des[], const char src[]){
  for (unsigned int int i = 0; src[i] != '\0'; i++)
    des[i] = src[i];
  des[i] = '\0';
}
```

Concatenation using Pointers

Concatenation using Pointers

```
// string concatenation
void cstringConcat(char des[], const char src[]){
  unsigned int i;
  // find the end of the destination c-string des
  for (i = 0; des[i] != '\0'; i++)
    ; // do nothing
  // append the source c-string src to the end of des
  for (unsigned int j = 0; src[j] != '\0'; j++){
    des[i] = src[j];
    i++;
  }
  // add a c-string terminator to the end of des
  des[i] = '\0';
}
```

Example: Book Records without Pointers

Example: Book Records without Pointers

```
// Book record
struct Book {
  string title;
  string author;
  string call_number;
};

// Global Book collection
Book collection[] = {
    {"Computer Security: Art and Science", "Matt Bishop",
     "QA 76.9.A25 B56 2002"},
    {"Applied Cryptography", "Bruce Schneier", "QA 76.9.A25 S35 1996"
    {"Practical Software Maintenance", "Thomas M. Pigoski",
     "QA 76.76.S64 P54 1996"}};
```

Example ctd.

Example ctd.

```
// function for printing Books
void printBook(const Book &book){
  cout << "title: " << book.title << endl;
  cout << "author: " << book.author << endl;
  cout << "call number: " << book.call_number << endl;
}

// function for finding a Book with some title
unsigned int findBook(const Book collection[], unsigned int n, const s
  for (unsigned int i = 0; i < n; i++){
    if (collection[i].title == title)
      return i;
  }
  return n;
}
```

Example Client Code

Example Client Code

```
const unsigned int COLLECTION_SIZE = sizeof(collection) / sizeof(Book

int main(){
  unsigned int i = findBook(collection,
                            COLLECTION_SIZE,
                            "Applied Cryptography");
  if (i == COLLECTION_SIZE)
    cout << "Book not found" << endl;
  else
    printBook(collection[i]);
  return 0;
}
```

Programming using pointers: Members

- What if we only had a *pointer* to a book?

Programming using pointers: Members

- What if we only had a *pointer* to a book?
  - `o->x` is the same as `(*o).x`

Programming using pointers: Members

- What if we only had a *pointer* to a book?
    - `o->x` is the same as `(*o).x`
    - Convenient for getting fields/methods from pointers

Programming using pointers: Members

- What if we only had a *pointer* to a book?
  - `o->x` is the same as `(*o).x`
  - Convenient for getting fields/methods from pointers

Programming using pointers: Members

- What if we only had a *pointer* to a book?
  - `o->x` is the same as `(*o).x`
  - Convenient for getting fields/methods from pointers

```
void printBook(const Book *book){
  cout << "title: " << book->title << endl;
  cout << "author: " << book->author << endl;
  cout << "call number: " << book->call_number << endl;
}
```

Programming using pointers: Failure

- You can never get the value from address `nullptr`

Programming using pointers: Failure

- You can never get the value from address `nullptr`
    - aka address `0`;

Programming using pointers: Failure

- You can never get the value from address `nullptr`
    - aka address `0`;
- So we can return `nullptr` when `findBook` fails to find the book

Programming using pointers: Failure

- You can never get the value from address `nullptr`
    - aka address `0`;
- So we can return `nullptr` when `findBook` fails to find the book

Programming using pointers: Failure

- You can never get the value from address `nullptr`
  - aka address `0`;
- So we can return `nullptr` when `findBook` fails to find the book

```
const Book *findBook(const Book collection[],
                     unsigned int n,
                     const string &title){
  for (const Book *p = collection; p < collection + n; p++){
    if (p->title == title)
      return p;
  }
  return nullptr;
}
```

Programming using pointers

Programming using pointers

```
const unsigned int COLLECTION_SIZE
  = sizeof(collection) / sizeof(Book);

int main(){
  const Book *b = findBook(collection,
                           COLLECTION_SIZE,
                           "Applied Cryptography");
  if (b == nullptr)
    cout << "Book not found" << endl;
  else
    printBook(b);
  return 0;
}
```

- YOU HAVE TO CHECK if the pointer is null

Programming using pointers

```
const unsigned int COLLECTION_SIZE
  = sizeof(collection) / sizeof(Book);

int main(){
  const Book *b = findBook(collection,
                           COLLECTION_SIZE,
                           "Applied Cryptography");
  if (b == nullptr)
    cout << "Book not found" << endl;
  else
    printBook(b);
  return 0;
}
```

- YOU HAVE TO CHECK if the pointer is null
- Otherwise you’ll get a lovely `segmentation fault` when you try to dereference

A note on Null Pointers

- Sir Tony Hoare, inventor of null pointers:

A note on Null Pointers

- Sir Tony Hoare, inventor of null pointers:
  I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

A note on Null Pointers

- Sir Tony Hoare, inventor of null pointers:
  I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
- But we're stuck with them

A note on Null Pointers

- Sir Tony Hoare, inventor of null pointers:
  I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
- But we're stuck with them
  - Newer languages like Rust and Swift have gotten rid of them