CME 211: Lecture 11

Topics

- Dynamic arrays
- Conditionals
- Basic file operations in C++

Conditional statements in C++

C++ has three conditional statements:

- `if`
- `switch`
- C++ ternary operator: `(x == y) ? a : b`

C++ `if`

```
#include <iostream>

int main()
{
  int n = 2;

  std::cout << "n = " << n << std::endl;
  if (n > 0)
  {
    std::cout << "n is positive" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./if1
n = 2
n is positive
```

Note: brackets `{...}` are not needed for a single line `if` block. However, I recommend always putting them in.

`else if`

```
#include <iostream>

int main() {
  int n = -3;

  std::cout << "n = " << n << std::endl;

  if (n > 0)
  {
    std::cout << "n is positive" << std::endl;
  }
  else
    if (n < 0)
    {
      std::cout << "n is negative" << std::endl;
    }

  return 0;
}
```

Output:

```
$ ./if2
n = -3
n is negative
```

else

```
#include <iostream>

int main()
{
  int n = 0;

  std::cout << "n = " << n << std::endl;

  if (n > 0)
  {
    std::cout << "n is positive" << std::endl;
  }
  else if (n < 0)
  {
    std::cout << "n is negative" << std::endl;
  }
  else
  {
    std::cout << "n is zero" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./if3
n = 0
n is zero
```

Common mistakes

Empty `if` due to extraneous semi-colon:

```
if (n < 0);
  std::cout << "n is negative" << std::endl;
```

Assignment in the conditional expression:

```
if (n = 0)
  std::cout << "n is zero" << std::endl;
```

Note: some people recommend always putting the ‘literal’ before the variable. This is known as a Yoda Condition.

`break`

The `break` keyword breaks out of the current loop.

```
#include <iostream>

int main()
{
  for (unsigned int n = 0; n < 10; n++)
  {
    std::cout << n << std::endl;
    if (n > 3) break;
  }

  return 0;
}
```

Output:

```
$ ./break
0
1
2
3
4
```

`continue`

The `continue` keyword moves to the next loop iteration.

```
#include <iostream>

int main()
{
  for (unsigned int n = 0; n < 10; n++)
  {
    if (n < 7)
      continue;
    std::cout << n << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./continue
7
8
9
```

Logical operators

- C++ has two choices for logical operators
- Newer style `and`, `or`, `not`
- Older style `&&`, `||`,
- Latter are backwards compatible with C

Logical AND

```cpp
#include <iostream>

int main()
{
  int a = 7;
  int b = 42;

  // the following are equivalent

  if (a == 7 and b == 42)
    std::cout << "a == 7 and b == 42 is true" << std::endl;

  if (a == 7 && b == 42)
    std::cout << "a == 7 && b == 42 is true" << std::endl;

  return 0;
}
```

Output:

```
$ ./logical1
a == 7 and b == 42 is true
a == 7 && b == 42 is true
```

`0` is false, everything else is true

```cpp
#include <iostream>

int main()
{
  int a[] = {-1, 0, 1, 2};

  for (int n = 0; n < 4; n++)
  {
    if (a[n])
      std::cout << a[n] << " is true" << std::endl;
    else
```

```
      std::cout << a[n] << " is false" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./logical2
-1 is true
0 is false
1 is true
2 is true
```

Bitwise results

```
#include <iostream>

int main()
{
  int a = 1;
  int b = 2;

  if (a)
    std::cout << "a is true" << std::endl;
  else
    std::cout << "a is false" << std::endl;

  if (b)
    std::cout << "b is true" << std::endl;
  else
    std::cout << "b is false" << std::endl;

  if (a & b)
    std::cout << "a & b is true" << std::endl;
  else
    std::cout << "a & b is false" << std::endl;

  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra logical3.cpp -o logical3
$ ./logical3
a is true
b is true
a & b is false
```

`switch`

- `if`, `else if`, `else`, etc. gets verbose if you have many paths of execution
- Can use a switch statement instead:

```
if (choice == `C')
  clearRecord();
else if (choice == `D')
  deleteRecord();
else if (choice == `A')
  addRecord();
else if (choice == `P')
  printRecord();
else
  std::cout << "Bad choice\n";
```

Becomes:

```
switch (choice) {
  case `C': clearRecord(); break;
  case `D': deleteRecord(); break;
  case `A': addRecord(); break;
  case `P': printRecord(); break;
  default: std::cout << "Bad choice\n";
}
```

`switch` and `enum` example

```
enum direction
{
  left,
  right,
  up,
  down
};

int main()
{
  direction d = right;

  std::string txt = "you are going ";
  switch (d)
  {
    case left:
      txt += "left"; break;
    case right:
      txt += "right"; break;
    case up:
      txt += "up"; break;
    case down:
      txt += "down"; break;
  }
  std::cout << txt << std::endl;
  return 0;
}
```

Output:

```
$ ./switch1
you are going right
```

Advantage

Compiler warnings will tell you if you are missing some cases.

```
switch (d)
{
  case left:
    txt += "left"; break;
  case right:
    txt += "right"; break;
  case down:
    txt += "down"; break;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra switch2.cpp -o switch2
switch2.cpp: In function 'int main()':
switch2.cpp:16:10: warning: enumeration value 'up' not handled in switch [-Wswitch]
switch (d)
^
```

Common mistake

Neglecting to add `break` in each case.

```
std::string txt = "you are going ";
switch (d)
{
  case left:
    txt += "left";
  case right:
    txt += "right";
  case up:
    txt += "up";
  case down:
    txt += "down";
}
std::cout << txt << std::endl;
```

Output:

```
$ g++ -Wall -Wconversion -Wextra switch3.cpp -o switch3
$ ./switch3
you are going rightupdown
```

Ternary operator

This is called the "ternary" operator:

```
a = b < 0 ? -b : b;
```

Equivalent code:

```
if (b < 0)
  a = -b;
```

```
else
  a = b;
```

Anatomy:

```
[conditional] ? [return expression if true] : [return expression if false];
```

`goto`

"If you find yourself using a `goto` statement within a program, then you have not thought about the problem and its implementation for long enough"

See: http://xkcd.com/292/

Figure 1: fig

C/C++ memory model

- All data in your application is stored in the same physical memory
- The memory used by each application is logically divided into the *stack* and the *heap*

Stack

- Fixed memory allocation provided to your application
- It is the operating system that specifies the size of the stack
- Stack memory is automatically managed for you by the compiler / operating system
- Limited to local variables of fixed size

Static array example

`src/stack4.cpp`:

```cpp
#include <iostream>

int main() {
  int a[2048][2048];
  a[0][0] = 42;
  std::cout << "a[0][0] = " << a[0][0] << std::endl;
  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion src/stack4.cpp -o src/stack4
$ ./src/stack4
Segmentation fault (core dumped)
```

Array `a` exceeded available stack size. This is your first stack overflow.

Heap

- Can contain data of arbitrary size (subject to available computer resources like total memory)
- Accessible by any function (global scope)
- Has the life of the program
- *Managed by programmer*

Using heap memory

- You need to allocate heap memory
- The location of the allocated memory is stored in a pointer, a special variable which stores a memory address
- When you are done using the memory you need to free the memory

Pointers

Declaration of a pointer is denoted by a `*` in front of the variable name (after the type)

- `int a;` – variable `a` will contain an integer
- `int *b;` – variable `b` will contain a memory address where an integer is stored
- `int* b;` – equivalent to `int *b;`. This is my prefered style. I would read it as: "`b` is a variable containing a pointer to an int". Hint: read C and C++ type declarations backwards.

Pointers contain addresses

Address-of operator `&`

Address of a variable is returned by applying operator `&` to a variable. For example,

```
int  a = 42;
int* b = &a;
```

will assign address of integer `a` to the pointer to integer `b`.

Dereferencing operator `*`

- We've already seen that the asterisk is used to denote the declaration of a pointer
- The asterisk is also used to access the data at the memory address stored in a pointer
- Expression `*b` returns variable pointed by the pointer `b`.
- This operation is called *dereferencing*

`src/pointer1.cpp`:

```
#include <iostream>

int main() {
  int a = 42;
  int* b; // b is a pointer to an int
```

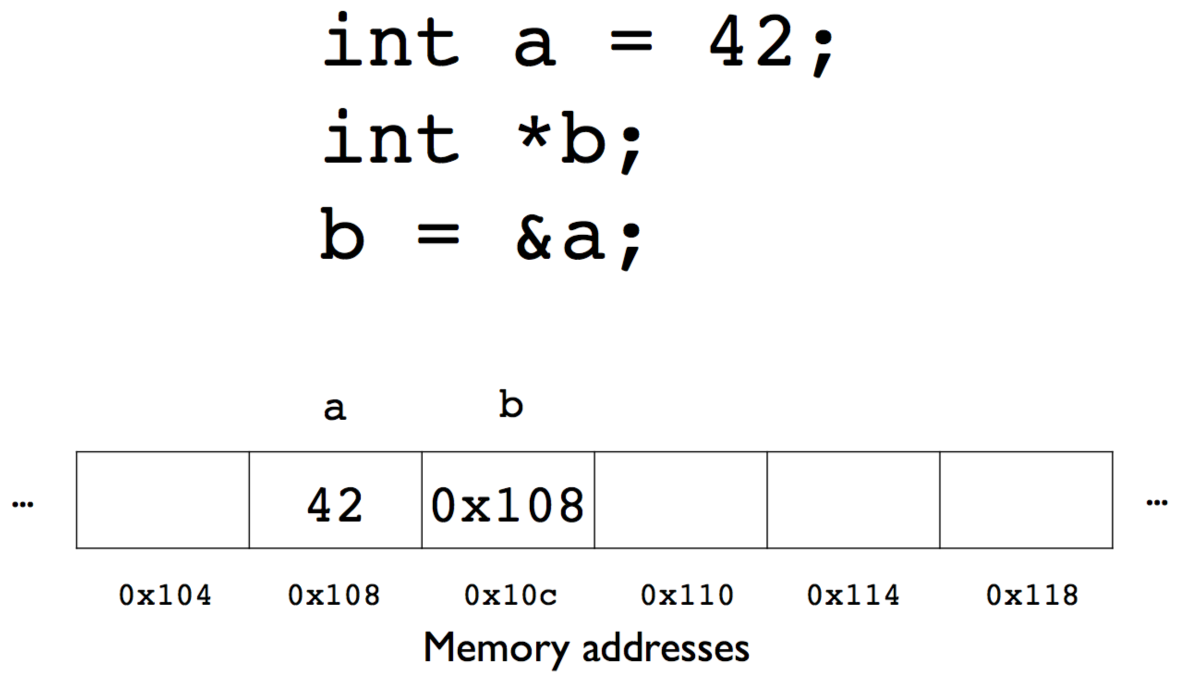

Figure 2: fig

```
  std::cout << " a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;

  b = &a; // here & is the "address of" operator

  // show the value of the pointer
  std::cout << " b = " << b << std::endl;

  // dereference the pointer
  std::cout << "*b = " << *b << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer1.cpp -o src/pointer1
$ ./src/pointer1
 a = 42
&a = 0x7fff5a43fad8
 b = 0x7fff5a43fad8
*b = 42
```

Storing a value

Pointer dereferencing allows you to store values at specific memory addresses.

`src/pointer2.cpp`:

```cpp
#include <iostream>

int main() {
  int a = 42;
  int *b;
  b = &a;

  std::cout << " a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << " b = " << b << std::endl;
  std::cout << "*b = " << *b << std::endl;

  // Store the value 7 at the
  // memory address stored in b
  *b = 7;

  std::cout << " a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << " b = " << b << std::endl;
  std::cout << "*b = " << *b << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer2.cpp -o src/pointer2
$ ./src/pointer2
 a = 42
&a = 0x7fff5ebc9a98
 b = 0x7fff5ebc9a98
*b = 42
 a = 7
&a = 0x7fff5ebc9a98
 b = 0x7fff5ebc9a98
*b = 7
```

Increment

`src/increment.cpp`:

```cpp
#include <iostream>

void increment(int *a) {
  // Value at the memory
  // address is incremented
  (*a)++;
}

int main() {
  int a = 2;
  std::cout << "a = " << a << std::endl;

  // increment() receives copy of memory address for a
  increment(&a);
  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/increment.cpp -o src/increment
$ ./src/increment
a = 2
a = 3
```

Returning pointers

`src/func.cpp`:

```
#include <iostream>

int* func(void) {
  int b = 2;
  return &b;
}

int main() {
  int *a = func();

  std::cout << " a = " << a << std::endl;
  std::cout << "*a = " << *a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/func.cpp -o src/func
src/func.cpp:5:11: warning: address of stack memory associated with local variable 'b' returned [-Wretu
  return &b;
          ^
1 warning generated.
$ ./src/func
 a = 0x7fff5bcf4acc
*a = 32767
```

Common mistake: pointer declaration

(There are many!)

`double *a, b;`

- `a` is a pointer to a double
- `b` is a double

`double *a, *b;`

- `a` is a pointer to a double
- `b` is a pointer to a double

```
double* a, b;
```

- `a` is a pointer to a double
- `b` is a **double**

Best way to define `a` and `b`:

```
double* a;
double  b;
```

Many uses of `*`

`src/pointer3.cpp`:

```
#include <iostream>

int main() {
  int a = 4;
  int *b = &a;

  // * used for dereferencing, multiplication, and storage
  *b = *b**b;

  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer3.cpp -o src/pointer3
$ ./src/pointer3
a = 16
```

Common mistake: uninitialized pointer

`src/pointer4.cpp`:

```
#include <iostream>

int main() {
  int *a;
  std::cout << "*a = " << *a << std::endl;
  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer4.cpp -o src/pointer4
src/pointer4.cpp:5:28: warning: variable 'a' is uninitialized when used here [-Wuninitialized]
  std::cout << "*a = " << *a << std::endl;
                           ^
src/pointer4.cpp:4:9: note: initialize the variable 'a' to silence this warning
  int *a;
        ^
         = nullptr
1 warning generated.
```

```
$ ./src/pointer4
/bin/sh: line 1: 61024 Segmentation fault: 11  ./src/pointer4
```

Suggestion

`src/pointer5.cpp`:

```
#include <iostream>

int main() {
  int *a = nullptr;
  std::cout << "*a = " << *a << std::endl;
  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer5.cpp -o src/pointer5
$ ./src/pointer5
/bin/sh: line 1: 61031 Segmentation fault: 11  ./src/pointer5
```

Dynamic memory allocation

- The `new` keyword *allocates* dynamic memory on the *heap*
- The `delete` keyword *frees* dynamic memory on the *heap*
- Works by setting aside a specified amount of *contiguous memory* and returning the *starting address*
- No guarantees about the state of initialization (i.e. the memory will have “random” data in it)

Memory allocation

`src/new1.cpp`:

```
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  // Allocate storage for n double values and
  // store the starting address in a
  double *a = new double[n];
  std::cout << "a = " << a << std::endl;

  for (unsigned int i = 0; i < n; i++)
    a[i] = i+3;

  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  // Free the memory
  delete[] a;
  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/new1.cpp -o src/new1
src/new1.cpp:6:20: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-con
  unsigned int n = std::stoi(argv[1]);
               ~   ^~~~~~~~~~~~~~~~~~
1 warning generated.
$ ./src/new1 2
a = 0x7fb562e00000
a[0] = 3
a[1] = 4
a = 0x7fb562e00000
$ ./src/new1 4
a = 0x7fc033c031a0
a[0] = 3
a[1] = 4
a[2] = 5
a[3] = 6
a = 0x7fc033c031a0
```

Out of bounds access

src/new2.cpp:

```
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  double *a = new double[n];
  std::cout << "a = " << a << std::endl;

  delete[] a;
  std::cout << "a = " << a << std::endl;

  for (unsigned int i = 0; i < n; i++)
    a[i] = i+3;

  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/new2.cpp -o src/new2
```

```
$ ./src/new2 2
a = 0xe98040
a = 0xe98040
a[0] = 3
a[1] = 4
$ ./src/new2 1048576
a = 0x7f8bf1c0b010
a = 0x7f8bf1c0b010
Segmentation fault (core dumped)
```

Suggestion

src/new3.cpp:

```
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  double *a = new double[n];

  delete[] a;
  a = nullptr;

  for (unsigned int i = 0; i < n; i++)
    a[i] = i+3;

  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  return 0;
}
```

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/new3.cpp -o src/new3
$ ./src/new3 2
Segmentation fault (core dumped)
```

Memory leaks

src/new5.cpp:

```
#include <iostream>
#include <string>

void ProcessData(double *a, unsigned int n)
{
  // temporary allocation for processing a
  // Memory is allocated but never freed
  double *tmp = new double[n];
  for (unsigned int i = 0; i < n; i++) tmp[i] = 0.;
  // Process a
  a[0] = tmp[0];

  return;
}

int main(int argc, char *argv[])
{
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  double *a = new double[n];

  // Process a
  ProcessData(a, n);

  delete[] a;
  a = nullptr;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -g -Wall -Wextra -Wconversion src/new5.cpp -o src/new5
src/new5.cpp:18:20: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-co
  unsigned int n = std::stoi(argv[1]);
               ~   ^~~~~~~~~~~~~~~~~~
1 warning generated.
$ valgrind ./src/new5 4
==61060== Memcheck, a memory error detector
==61060== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==61060== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==61060== Command: ./src/new5 4
==61060==
==61060==
==61060== HEAP SUMMARY:
==61060==     in use at exit: 22,100 bytes in 190 blocks
==61060==   total heap usage: 255 allocs, 65 frees, 27,844 bytes allocated
==61060==
==61060== LEAK SUMMARY:
==61060==    definitely lost: 32 bytes in 1 blocks
==61060==    indirectly lost: 0 bytes in 0 blocks
==61060==      possibly lost: 0 bytes in 0 blocks
==61060==    still reachable: 0 bytes in 0 blocks
==61060==         suppressed: 22,068 bytes in 189 blocks
==61060== Rerun with --leak-check=full to see details of leaked memory
==61060==
==61060== For counts of detected and suppressed errors, rerun with: -v
==61060== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C++ file I/O

- Like outputting to the screen, file I/O is also handled via streams

- Three stream options:
- `ofstream`: output file stream (i.e. write)
- `ifstream`: input file stream (i.e. read)
- `fstream`: file stream (i.e. read or write)

`ofstream`

```
#include <iostream>
#include <fstream>

int main() {
  std::ofstream f;

  f.open("hello.txt");
  if (f.is_open()) {
    f << "Hello" << std::endl;
    f.close();
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }

  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra ofstream1.cpp -o ofstream1
$ rm -f hello.txt
$ ./ofstream1
$ cat hello.txt
```

Using a variable for the filename

Code:

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
  std::string filename = "file.txt";

  std::ofstream f;
  f.open(filename);
  if (f.is_open()) {
    f << "Hello" << std::endl;
    f.close();
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }
  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra ofstream2.cpp -o ofstream2
ofstream2.cpp: In function 'int main()':
ofstream2.cpp:10:18: error: no matching function for call to
'std::basic_ofstream<char>::open(std::string&)'
f.open(filename);
^
ofstream2.cpp:10:18: note: candidate is:
In file included from ofstream2.cpp:2:0:
/usr/include/c++/4.8/fstream:713:7: note: void std::basic_ofstream<_CharT,
_Traits>::open(const char*, std::ios_base::openmode) [with _CharT = char; _Traits =
std::char_traits<char>; std::ios_base::openmode = std::_Ios_Openmode]
open(const char* __s,
^
/usr/include/c++/4.8/fstream:713:7: note:
no known conversion for argument 1 from
'std::string {aka std::basic_string<char>}' to 'const char*'
```

Change to:

```
  f.open(filename.c_str());
```

Output:

```
$ g++ -Wall -Wconversion -Wextra ofstream3.cpp -o ofstream3
$ rm -f file.txt
$ ./ofstream3
$ cat file.txt
```

C++ 2011 standard

Specify usage of the C++ 2011 standard. Passing an `std::string` to `f.open` is supported:

```
g++ -std=c++11 -Wall -Wconversion -Wextra ofstream2.cpp -o ofstream2
rm -f file.txt
./ofstream2
cat file.txt
```

Writing an array of values

```
#include <iostream>

//  Define constants to size the static array
#define ni 2
#define nj 3

int main() {
  int a[ni][nj];

  // Initialize the array values
  int n = 0;
  for (int i = 0; i < ni; i++) {
    for (int j = 0; j < nj; j++) {
      a[i][j] = n;
      n++;
    }
  }

  // Store the array values in a file
  std::ofstream f("array.txt");
  if (f.is_open()) {
    f << ni << " " << nj << std::endl;
      for (int i = 0; i < ni; i++) {
        f << a[i][0];
        for (int j = 1; j < nj; j++) {
          f << " " << a[i][j];
        }
        f << std::endl;
      }
    f.close();
  }
  return 0;
}
```

`fstream`

```
#include <iostream>
#include <fstream>

int main() {
  std::fstream f;

  // specify output mode with second argument
  f.open("hello.txt", std::ios::out);
  if (f.is_open()) {
    f << "Hello" << std::endl;
    f.close();
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }

  return 0;
}
```

Reading from a file

- Not as easy or convenient as in Python
- We will start by looking at how to read the simple array file we previously wrote

ifstream

```
#include <iostream>
#include <fstream>

int main() {
  // Read the array values from the file
  std::ifstream f("array.txt");
  if (f.is_open()) {
    int i;
    while (f >> i) { // Stream extraction operator
      std::cout << i << std::endl;
    }
    f.close();
  }
  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra ifstream1.cpp -o ifstream1
$ ./ifstream1
2
3
0
1
2
3
4
5
```

Reading the array

```
// Read the array values from the file
std::ifstream f("array.txt");

if (f.is_open()) {
  // Read the size of the data and make sure storage is sufficient
  int nif, njf; // Values of ni and nj read to be read from file
  f >> nif >> njf;
  if (nif > ni or njf > nj) {
    std::cout << "Not enough storage available" << std::endl;
    return 0; // quit the program
  }

  // Read the data and populate the array
  for (int i = 0; i < nif; i++) {
    for (int j = 0; j < njf; j++) {
      f >> a[i][j];
    }
  }
  f.close();
}
```

Reading

C++ Primer, Fifth Edition by Lippman et al:

- Chapter 1: Statements: Sections 5.3 - 5.5
- Section 2.3.2: Pointers
- Section 12.2: Dynamic Arrays
- Section 7.1.5: Destruction
- Chapter 8: The IO Library: Section 8.2