PIC 10A
Pointers, Arrays, and Dynamic Memory Allocation

Ernest Ryu
UCLA Mathematics
A variable is stored somewhere in memory. The *address-of* operator & returns the memory address of the variable.

```cpp
double d = 0.1;
double* d_p = &d;
cout << d_p << endl;
```

The output happens to be 008FF780.
Pointers

double* d_p;

d_p is a variable of type double* (read as “double pointer”). Or we say d_p a pointer to a double.

double* d_p = &d;

&d returns the address of d, and it is of type double*.

cout << d_p << endl;

This outputs the memory address at which d is stored. A double is 8 bytes. So the memory addresses 008FF780, 008FF781, ..., 008FF787 are used to store d.
Pointers

The \textit{indirection} or \textit{dereference} operator \* allows us to access the variable a pointer is pointing to.

\begin{verbatim}
double d = 0.1;
double* d_p = &d;
cout << *d_p << endl;
*d_p = 3.0;
cout << d << endl;
\end{verbatim}

The output is 0.1 and 3.0.

\*d_p returns the double of value 0.1.
\*d_p = 3.0; assigns 3.0 to the double at location d_p.

You now know 3 separate uses for the symbol \*: multiplication, dereferencing, and defining a pointer.
sizeof operator

sizeof returns the number of bytes it takes to store a variable.

```cpp
cout << sizeof(int) << endl;
cout << sizeof(double) << endl;
```

The output (on my machine) is 4 and 8.
Arrays

An array is a sequence of variables.

```cpp
int ages[10];

for (int i=1; i<=10; i++) {
    int thisAge;
    cout << "Student number " << i << " has age: ";
    cin >> thisAge;
    ages[i-1] = thisAge; // 0-based indexing
}

cout << "The ages are" << endl;

for (int i=0; i<10; i++)
    cout << ages[i] << endl;
```

(Elements of an array is accessed with 0-based indexing.)
Arrays are pointers

Actually arrays are (almost) pointers!

```cpp
int ages[10];
int* int_p = ages; //ages is of type int*

for (int i=1; i<=10; i++) {
    ... //Initialize ages[
}

cout << ages << endl;
cout << int_p << endl;

for (int i=0; i<10; i++)
    cout << int_p[i] << endl; //You can do this

int_p and ages are the same.
```
Arrays are pointers

Since ages is of type \texttt{int*}, it points to a memory address at which an integer is stored.

\begin{verbatim}
int ages[10];

for (int i=1; i<=10; i++) {
    ... //Initialize ages[
}

cout << ages << endl;
\end{verbatim}

The output is the address of \texttt{ages[0]}. 
The memory layout of an array
You can do basic arithmetic with pointers. Since `sizeof(int)` is 4, `int_p+1` points to the address 4 bytes away from `int_p`.

```cpp
cout << int_p << endl;
cout << (int_p+1) << endl;
```

The output happens to be `00F3FC64` and `00F3FC68`. 
Arrays are pointers

We can also access an array with pointer arithmetic (as opposed to using []).

```cpp
for (int i=0; i<10; i++)
  cout << ages[i] << endl;

for (int i=0; i<10; i++)
  cout << *(ages + i) << endl;

for (int* iter = ages; iter<(ages+10); iter++)
  cout << *iter << endl;
```

These 3 for loops have the same output.
**Array initialization**

We can declare and initialize a variable in one statement using an *initializer list*.

```plaintext
double d = 0.3;
```

We can do the same with arrays.

```plaintext
// w has length 3 and holds 1,2,3
int w[] = {1,2,3};
// x has length 5 and holds 1,2,3000,0,0
float x[5] = {1.f,2.f,3E3f};
// y has length 3 and holds all zeroes
int y[3] = {0};  // Better style?
// z has length 10 and holds all zeroes
double z[10] = {};
```
Pointers reveal the physical location of variables. Where do variables go?

- The operating system (OS) keeps track of which patch of memory is available.
- When the C++ program declares a variable, it asks the OS for memory.
- Which patch of memory you get depends on what part of the memory is available, which in turns depends on the what other programs are running and the current state of the OS.
- Arrays are placed on a contiguous patch of memory. This has to be true for pointer arithmetic to work.
Type conversion

Type conversion or type casting is the conversion of a variable of one type to another.

\[ \text{double } d1 = 3; \]

3 is a literal of type \texttt{int}, and it is (implicitly) converted/casted into a double.

There are explicit and implicit conversions.
Explicitly convert $T_1 \rightarrow T_2$.

- **C-style cast**: $(T_2)\ T_1$.

```cpp
float f = (float) 3; // cast int to float
double d = (double) f; // cast float to double
// cast 1 and 3 into double before division
cout << ((double) 1) / ((double) 3) << endl;
```
Type conversion: explicit

Explicitly convert $T_1 \rightarrow T_2$.

- *Functional cast or function style cast*: $T_2(T_1)$.

```c
int i = 2000000000; // i = 2 billion
cout << 10 * i << endl;
cout << 10 *( long long int(i) ) << endl;
```

The output is $-1474836480$ and $20000000000$. 
Explicitly convert $T_1 \rightarrow T_2$.

- **Static cast**: $\text{static\_cast}<T_2>(T_1)$.

```cpp
bool b1 = static_cast<bool>(0);
bool b2 = static_cast<bool>(3);
cout << b1 << endl;
cout << b2 << endl;
```

The output is 0 and 1.

We will later discuss where the names C-style cast and functional cast come from.
Type conversion: explicit

Upon encountering these casts, the compiler will attempt to carry them out. If there is no a suitable way, the cast will fail (i.e., your code won’t compile).

```cpp
int i = 3;
string s1 = static_cast<string>(i);  // Error!
```

We’ll later learn what “a suitable way of conversion” means.
Type conversion: implicit

Implicit conversion from type $T_1$ to type $T_2$ is attempted when type $T_1$ is used where type $T_2$ is expected.

```java
double d = 5;  // Implicit cast from int to double
int i = 3;
float f = i;  // Implicit cast from int to float
```
Type conversion: which one?

Which cast should you use?

- Ordered in verbosity:
  static < functional = C-style < implicit.

- Ordered in readability (subjective):
  funcational > static > implicit > C-style.

- Ordered in safety:
  static > functional = C-style > implicit.

Static cast is safe because it is conspicuous in that is is more visible and in that you can easily search (with ctrl+F) to find it. (There’s one more subtle reason that’s beyond the scope of this class.)

In my opinion, the functional cast is always better than the C-style cast. Otherwise, handle the trade-off on a case-by-case basis.
Use `typeid` to check what type your object is.

```cpp
#include <iostream>
#include <typeinfo>
using namespace std;

int main() {
    int i = 5;
    long long int j = 2;
    cout << typeid(i).name() << endl
         << typeid(j).name() << endl
         << typeid(i+j).name() << endl;
}
```

Output is `int`, `__int64`, and `__int64`.
The type char

The type char (short for character) is a fundamental type. sizeof(char) equals 1. A literal of type char is written as a single character in single quotes.

```cpp
char c1 = 'a';
cout << c1 << endl;
```
The type char

A char can take on 256 possible values. (Why 256?) The ASCII standard (American Standard Code for Information Interchange) specifies which value corresponds to which character.

http://www.asciitable.com/

```cpp
#include <iostream>
#include <bitset>
using namespace std;

int main() {
    char c = char(65);
    cout << c << endl;
    bitset<8> x(c);
    cout << x << endl; // see the bits stored in &c
}
```

The output is A and 01000001.
Arithmetic with char

We also can view `char` as an integer type with a value range of $-128$ to $127$. You can (implicitly) cast a `char` into an `int` and perform arithmetic.

```cpp
char c;
while (true) {
    cout << "Enter a lower case letter" << endl;
    cin >> c;
    cout << "We capitalized it: " << char(c-0x20) << endl;
    cout << typeid(c-0x20).name() << endl; // int
}
```

- `0x20` is a literal of type `int` written in hex. `0x` or `0X` means hex.
- In `c-0x20`, we have a `char` minus an `int`. So `c`, a `char`, is implicitly converted into an `int` to perform the arithmetic.
- `char(c-0x20)` converts the `int` back to a `char`. 
Relational operators with char

The comparison operators are quite useful.

```c
char c;
cout << "Input a letter" << endl;
cin >> c;
if ('a' <= c && c <= 'z')
    cout << "Input is lower case" << endl;
else if ('A' <= c && c <= 'Z')
    cout << "Input is upper case" << endl;
else if ('0' <= c && c <= '9')
    cout << "Input is a number" << endl;
else if (' ' <= c && c <= '~')
    cout << "Input is a special character" << endl;
else
    cout << "Input is something else" << endl;
```
A char array is more than just a sequence of chars.

- In programming, a string is a sequence of characters. In C++ there are 2 common ways to represent a string: with type `string` and with a `char` array.
- A C-style string is a `char` array terminated by the null character, written as `'\0'` or `char(0)`.
- Strings are better than C-style strings, but you must know both.
**char array initialization**

Char arrays are special in that they have an additional way to initialize.

```c
char s1[] = {'H','e','l','l','o','\0'};
char s2[] = "Hello"; // special syntax
char s3[] = {'H','e','l','l','o'};
```

`s1` and `s2`, valid c-strings, are char arrays of size 6. 
`s3`, not a valid c-string, is of size 5.
The normal initializer list syntax works with char arrays.

```c
char s1[5] = {'a','b'};
char s2[5] = {'a','b','\0','\0','\0'};
cout << int(s1[2]) << endl;
```

The output is 0. s1 and s2 are the same valid C-style strings.
char array cout

cout treats char arrays specially.

```cpp
int i_a[] = {1,2,3};
char s1[] = {'H','e','l','l','o','\0'};
char s2[] = "Hello"; // special syntax
char s3[] = {'H','e','l','l','o'};
cout << i_a << endl; // pointer address
cout << s1 << endl; // Hello
cout << s2 << endl; // Hello
cout << s3 << endl; // Fail
```

Instead of printing the address s1, cout performs something like

```cpp
for (int i=0; s1[i]!=\0; i++)
    cout << s1[i];
cout << endl;
```

So cout needs the terminating null character to know when to stop.
"Hello" isn't a literal of type string. It's actually a literal of type `const char` array.

```cpp
cout << typeid("Hello").name() << endl;
const char * s = "Hello";
cout << s << endl;
```

When we assign a C-string literal to a string, implicit type casting happens.

```cpp
//string s = "Hello world";
//the same as
string s = static_cast<string>("Hello world");
```
Limitations of arrays (on the stack)

The length of the array must be a constant (i.e., specified in the code).

```cpp
int n; cin >> n;
double d[n]; // Error
```

Using `const int` doesn't solve the problem.

```cpp
int n; cin >> n;
const int m = n;
double d[m]; // Error
```

These arrays can’t be too large.

```cpp
int arr[1000000]; // Error
```

1,000,000 ints occupy merely 4 megabytes.
Within memory, there is the *stack* and the *heap*.

- The stack is small but fast (for reasons beyond the scope of this class). The variable and array definitions we’ve seen so far place the variables on the stack. These are also called *local* or *automatic variables*.

- The heap, also called *dynamic memory*, is large but slower (but still fast compared to secondary storage). We will learn *dynamic memory allocation*, which places variables on the heap.

- In C++, the opposite of dynamic variables isn’t static variables. Static variables mean something else unrelated to the stack.
Dynamic allocation

Use the `new` operator to declare a variable on the heap.

```cpp
double* d_p = new double;
```

`new T` creates a variable of type `T` and returns a pointer pointing to it.

Variables on the stack don’t use `new`.

```cpp
double d = 0.0;
```
You must *deallocare* a dynamically allocated variable with the delete operator when you’re done with it. In other words, you must *free* the memory when you’re done with it.

```cpp
double* d_p = new double;
...  // code using d_p;
delete d_p;
```

The deallocation tells the OS that we’re no longer using the memory address.

Variables on the stack are deallocated automatically when they go out of scope.
Dynamic arrays

Use `new T[n]` to declare an array of type `T` of length `n` on the heap.

```cpp
int n; cin >> n;
double* d_a = new double[n];
delete[] d_a;
```

You must use `delete[]` to deallocate a dynamic array.

Don’t use `delete[x]`. Don’t use `delete`. Use `delete[]`. 
Dynamic arrays

Advantages of dynamic arrays over local arrays:
- Array length does not have to be predetermined.
- Array can be large enough to hold gigabytes.

```cpp
#include <iostream>
using namespace std;

int main() {
    int n; cin >> n;
    char* c_a = new char[n];
    cout << "wait a second";
    char c; cin >> c;
    //check the memory monitor
    delete[] c_a;
    cout << "memory has been freed";
    cin >> c;
}
```
Memory leak

When you don’t free memory you allocate, the memory is *leaked* and becomes unavailable until the end of the program. (The OS reclaims the memory when the program is done.)

```cpp
int main() {
    int count = 0;
    while (true) {
        char* c_p;
        c_p = new char[1000000];
        count ++;
        cout << count << " Mbytes of memory leaked.\n";
    }
}
```

Memory leaks are one of the worst kinds of bugs! They are very hard to track down as they often do no immediate harm. However, they can lead your program to run out of memory and crash over time.
Arrays vs. pointers

Actually, there is a small difference between an array and a pointer. Write \( T[\] \) for an array of type \( T \).

```cpp
int main() {
    int* n;
    int m[2];
    int* k = new int[10];
    cout << typeid(n).name() << '\n';
    cout << typeid(m).name() << '\n';
    cout << typeid(k).name() << '\n';
    delete[] k;
}
```

The distinction is rarely necessary. Only local arrays can have type \( T[\] \).