C++ Programming

The Basics

The Basics

- Structure of a C++ program
- Statements and expressions
- Control flow
- Structures
- Pointers
- Memory structure
- Examples

Before We Begin

- When programming in C++, always keep in mind its C roots
  - It inherits most of its syntax and structure from C
  - Most (but not quite all) C code is valid C++ code; as a result, C++ is not quite a strict superset of C
  - Entire programs in C++ can be written using only regular functions not defined in any class; in other words, classes are not mandatory
- This is absolutely not the right way to write an object-oriented program; use classes, objects, and methods!

Structure of a C++ Program

- The basic elements of a C++ program are
  - The classes (i.e., a notion of Abstract Data Types),
  - The methods (i.e., functions encapsulated in classes), and
  - The data members (i.e., data fields encapsulated in classes)

Structure of a C++ Program

- Most programs are made up of multiple classes (with methods and data members) and functions
- A main function is required for a program as an entry point to bootstrap the rest of its functionality
  - One main function must exist
  - No more than one can exist in the same program

The Simplest C++ Program

```
int main()
{
}
```

C's Hello World is a C++ Program

```
#include <stdio.h>

/* Simple Hello World program. */

int main()
{
   printf("Hello World!\n");
}
```

A More C++-ish Hello World

```
#include <iostream>

// Simple Hello World program.

int main()
{
   std::cout << "Hello World!" << std::endl;
}
```

A Slightly Better More C++-ish Hello World

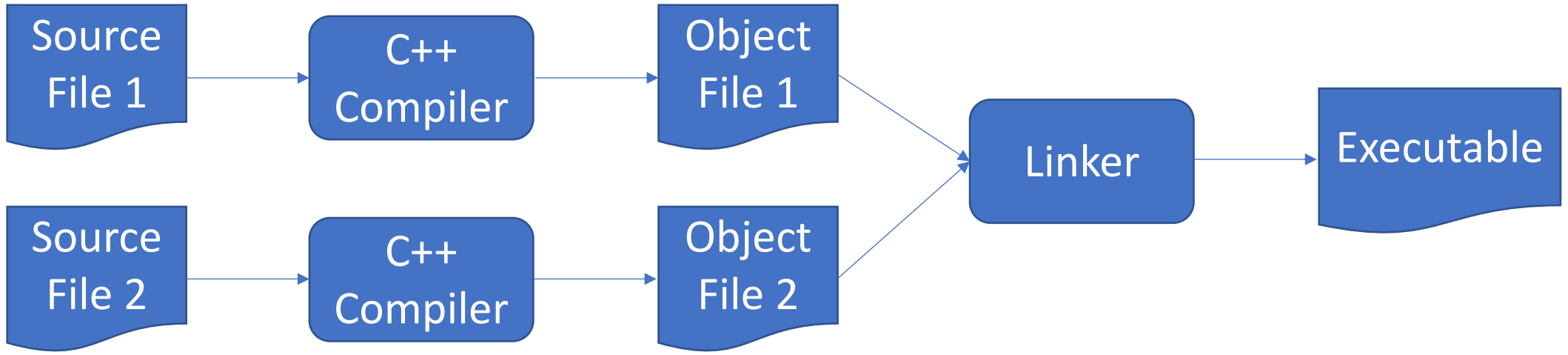
```
#include <iostream>
using namespace std;

// Simple Hello World program.

int main()
{
   cout << "Hello World!" << endl;
}
```

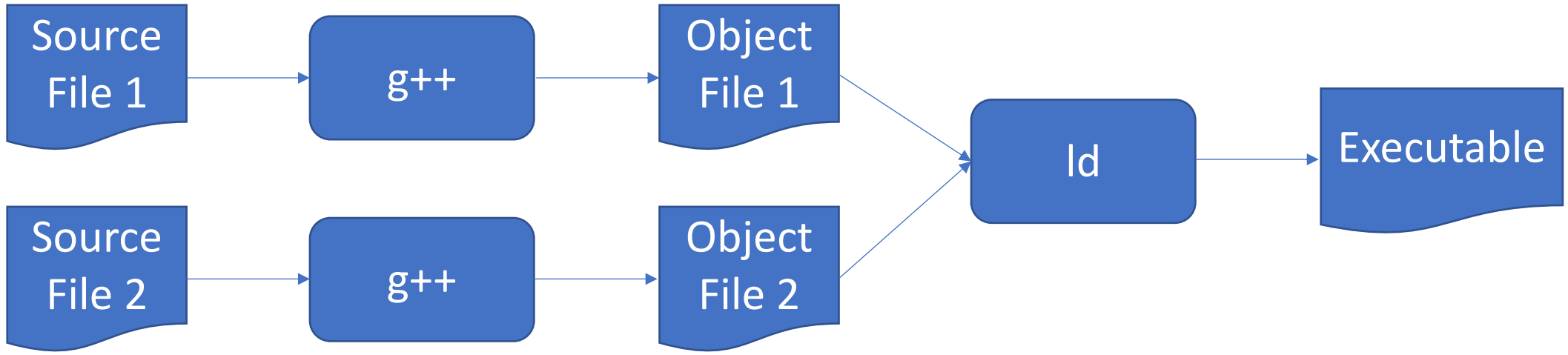
Building C++ Programs

- C++ is a compiled language and so to run a C++ program, its source code must be compiled and linked to produce an executable



Building C++ Programs

- On Linux and most Unix-like systems, things would typically be built using g++ and ld (though sometimes c++ is used instead, and ld is often hidden)



Building C++ Programs

- From a command line, building in one step would look like:

```
> g++ HelloWorld.cpp -o HelloWorld
> ./HelloWorld
```

- Alternatively, you can build and keep the object files and do things in multiple steps like:

```
> g++ -c HelloWorld.cpp
> g++ HelloWorld.o -o HelloWorld
> ./HelloWorld
```

Statements

- As in C, a statement in C++ is a command in a program to direct the program to take a particular action
- Likely the simplest statement in C++ is an expression statement
  - This is simple an expression (which is optional) followed by a semicolon(;)
  - This does mean that the following (the null statement) is a valid statement:

```
;
```

- Usually though, an expression is given and that expression is evaluated when the statement is executed

Expressions and Types

- Every expression has a type that determines the operations that may be performed on it
- A declaration is a statement that introduces a name into a program, specifying a type for the named entity
- For example:

```
int foo;
```

introduces a named variable foo that is of type integer

Expressions and Types

- C++ has a similar set of basic types to C
- This includes things like bool, char, int, and double
- Each fundamental type corresponds directly to hardware facilities and has a fixed size that dictates the range of values that it can represent
- C++ also has a collection of other types that C traditionally does not; we will touch on some of those later on

Constant Expressions

- C++ has multiple concepts of constants:
  - `#define` constants – preprocessor definitions that do a simple replacement during preprocessing and before compiling
  - `const` constants – named constant declarations where the programmer commits to not changing a value, and this promise is enforced by the compiler
  - `constexpr` constants – constant expressions evaluated at compile time, allowing them to be placed in read-only memory or to improve performance (new in C++11)

Constant Expressions

```
#define PI 3.14
constexpr double circleArea(double x) { return PI*x*x; }

int main() {
    const double pi = 3.14;
    constexpr double radius1 = 5.0;
    const double radius2 = 10.0;

    constexpr double area1 = circleArea(radius1);
    constexpr double area2_error = circleArea(radius2);
    const double area2_good = circleArea(radius2);
}
```

Expressions and Operators

- C++ also uses a similar set of operators to C
- Arithmetic operators like +, -, *, /, and %
- Comparison operators like ==, !=, <, >, <=, and >=
- Assignment and initialization operator as =
- In assignments and arithmetic operations, C++ does do conversions between basic types as in C

Expressions and Operators

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

int main()
{
   int sum;
   bool bigEnough;
   sum = 50 + 50;
   bigEnough = sum >= 100;
   cout << "Sum is " << sum << ". Is it big
     enough?  " << bigEnough << endl;
}
```

Statements and Blocks

- A block, or compound statement, is the term given to a collection of statements enclosed in {…}
  - This is the way that a program can group multiple statements together into a single statement, often for the purposes of control flow
  - A block also defines a scope for variables declared in it; local variables are put on the stack for the duration of the execution of the block statement
  - Such variable declarations are optional, and no new variables need be introduced in a block
  - Originally all such declarations had to be at the beginning of the block, but they can now generally be interwoven with statements through the code

Control Flow

- In C++, the flow of control in a program behaves very similarly to how things are done in C
  - Sequence: The normal stepping through from one statement to the next
  - Selection: To enable selection for the next statement or block from amongst a number of possibilities
  - Repetition: Repetition of a single statement or of a block; repetition can take many forms depending on the needs of the program, including for(...), while(...), and do...while loops.

Control Flow – Selection

- One-way selection is accomplished using a simple if

```
if (<logical expression>)
     <statement1> //performed if expression is true
```

Control Flow – Selection

- Two-way selection is accomplished using if-else

```
if (<logical expression>)
     <statement1> //performed if expression is true
else
     <statement2> //performed if false
```

Control Flow – Selection

- Multi-way selection using the else if

```
if(<expression1>)
   <statement1>
else if (<expression2>)
   <statement2>
else if (<expression3>)
   <statement3>
else
   <statement4> // otherwise -- default
```

Control Flow – Selection

- Multi-way selection using the switch / case statement

```
switch (<expression>){
case <constant-exp1>: <statement(s)1> [break;]
case <constant-exp2>: <statement(s)2> [break;]
case <constant-exp3>: <statement(s)3> [break;]
default: <statement(s)d>; [break;]
}
```

Control Flow – Repetition

- while loops:

```
while(<expr>) <statement>
```

```
int count = 0;
while (count < 10) {
     cout << "Count is: " << count << endl;
     count++;
}
```

Control Flow – Repetition

- for loops:

```
for(<expr-init>;<condition expr>;<expr-iter>)
     <statement>
```

```
for(int count = 0; count < 10; count++) {
     cout << "Count is: " << count << endl;
}
```

Control Flow – Repetition

- do-while loops:

```
do
     <statement>
while (<expr>);
```

```
int count = 0;
do {
     cout << "Count is: " << count << endl;
     count++;
} while (count < 10);
```

Structures

- In C++, a user-defined record (aggregate type) is called a structure and is referred to as a `struct` in a program
- They are handled in much the same way as they are in C, with slight differences (that ultimately make them easier to use and refer to)
- Interestingly in C++, structures and classes are more-or-less equivalent except for their default visibility:
  - In a structure, members default to public; whereas,
  - In a class, members default to private.
- Much more on classes shortly!

Structures

- As noted previously, structures are defined using the `struct` keyword

```
struct Point {
  int x;
  int y;
};
Point p1, p2;
```

- Notice the difference between how C++ and C would handle declaring things using our new `Point` structure?

Structures

- C would require us to refer to our structure as `struct Point` or else it would generate an error during compilation
- As this would get tedious, you could use the `typedef` directive to name a new type to avoid having to do this
- In C++, however, this is not necessary; you can do things the C way, but you are not required to do so

Structures

- Accessing fields: Individual fields of a structure can be accessed using the "." syntax

```
Point p1;
p1.x = 3;
p1.y = 2;
```

When we have a pointer to a structure (more on pointers in a minute), we can instead access fields using "->"

Structures

- Assignment: A structure can be assigned as a complete unit:

```
Point p1, p2;
p1.x = 3;
p1.y = 2;
p2 = p1;
```

This last statement is equivalent to:

```
p2.x = p1.x;
p2.y = p1.y;
```

Pointers

- Pointers in C++ function very similarly to how they work in C
- In C++, a pointer variable can point to any sort of memory
  - Pointer to basic types (like an integer)
  - Pointer to a structure
  - Pointer to an array (more on this soon)
  - Pointer to a class (more on this soon as well)
  - etc.
- This is only natural as a pointer holds the address of something in memory and everything in memory has to have an address …

Pointers

- There are two main uses for pointers in C++:
  - As a way of referring to memory allocated dynamically off the heap (using the `malloc()` function or the operator `new`)
    - This allows for data that can be created on the fly or dynamically sized (such as linked lists, trees, etc.)
  - When passing large chunks of data to a function or method, passing a pointer can be more efficient, as it reduces copying and speeds up processing
    - For example, when passing a large array (and, again, we'll see more on arrays soon)

Pointers

- Recall that as C++ has no built-in garbage collection, the programmer is responsible for freeing up dynamically allocated storage when they are done with it (using `free()` or `delete`, depending on how it was allocated in the first place)
- This can have performance advantages, but is cumbersome and can be painful / error-prone so it has to be done carefully
- If a programmer fails to do this properly, their program will leak memory leading to performance and stability problems (and crashes!)

Declaring Pointers

- Pointers are declared using the "*" notation, and are designated to point to certain types at the time, as in the examples below

```
int *pi;     // pi is a pointer to an integer

double *pd; // pd is a pointer to a double

char *s;     // s is a pointer to a char
```

Declaring Pointers

- This said, C++ also has a notion of a pointer without a specific type attached to it, which can be handy at times
- These are declared as:

```
void *vp;
```

- We will come back to these in a bit to show how they can be used

Reference (Address-of) Operator (&)

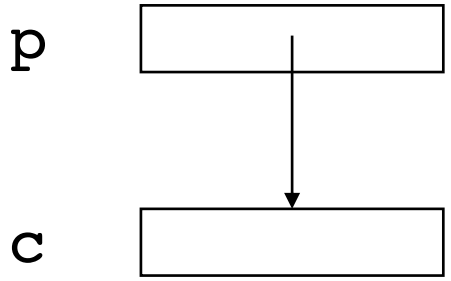
- Reference Operator &: When applied to a variable, & generates a pointer-to (or address-of) the variable
- Example:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
```

- Because p now has the address of and points to c, it can manipulate c indirectly! (This can be useful, but one must exercise caution!)

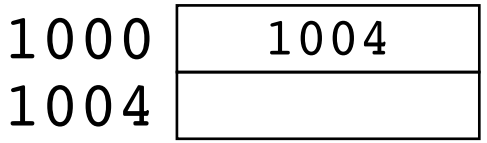
Reference (Address-of) Operator (&)

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
```



Reference (Address-of) Operator (&)

```
int *p;  // suppose p is at address 1000 in memory
int c;   // and c is at address 1004 in memory
p = &c;  // stores the address of c in p as a pointer
```



Pointers and Assignment

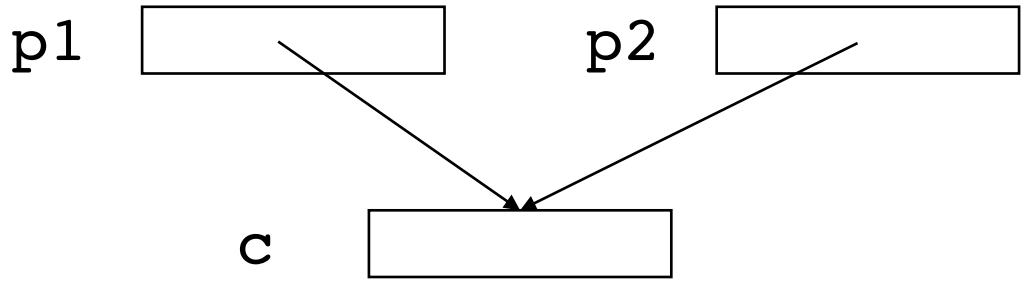
- You can make a pointer point at whatever another pointer is pointing at using the standard assignment operator (=)
- Example:

```
int *p1; // p1 is a pointer to int (a declaration)
int *p2; // p2 is a pointer to int (a declaration)
int c;
p1 = &c; // causes p1 to point to c
p2 = p1; // causes p2 to point to c as well
```

- If we want a pointer to point at nothing we can assign it the value of NULL

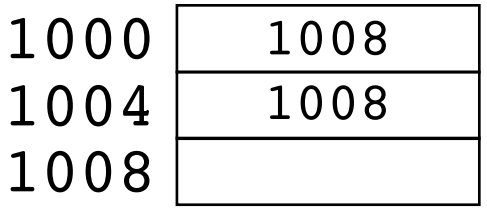
Pointers and Assignment

```
int *p1; // p1 is a pointer to int (a declaration)
int *p2; // p2 is a pointer to int (a declaration)
int c;
p1 = &c; // causes p1 to point to c
p2 = p1; // causes p2 to point to c
```



Pointers and Assignment

```
int *p1;   // suppose p1 is at address 1000 in memory
int *p2;   // and p2 is at address 1004 in memory
int c;     // and c is at address 1008 in memory
p1 = &c;   // stores the address of c in p1 as a pointer
p2 = p1;   // stores the address of c in p2 as a pointer
```



Dereference Operator (*)

- When we want to access or manipulate what a pointer is pointing at, we dereference the pointer first. Example:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c through p
```

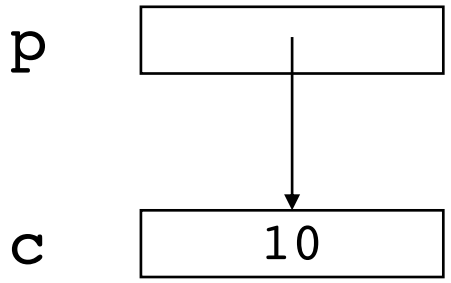
- Note that if we just said:

```
p = 10;
```

  we would be giving p the address 10, and having it effectively point to whatever is sitting in memory at that address!

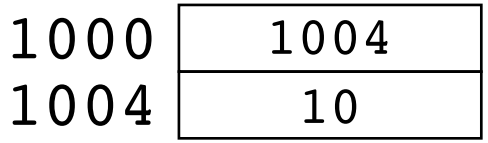
Dereference Operator (*)

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c via p
```



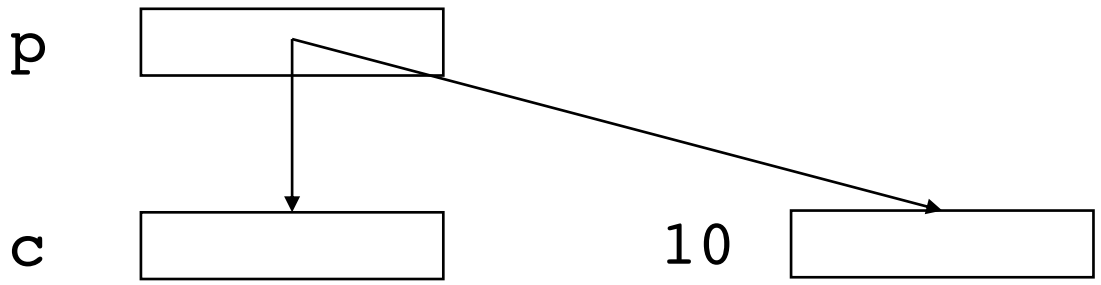
Dereference Operator (*)

```
int *p;  // suppose p is at address 1000 in memory
int c;   // and c is at address 1004 in memory
p = &c;  // stores the address of c in p as a pointer
*p = 10; // assigns the value of 10 to address 1004
```



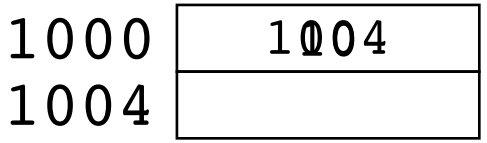
Dereference Operator (*)

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
p = 10;  // causes p to point to address 10
```



Dereference Operator (*)

```
int *p;  // suppose p is at address 1000 in memory
int c;   // and c is at address 1004 in memory
p = &c;  // stores the address of c in p as a pointer
p = 10;  // stores the address 10 in p as a pointer
```



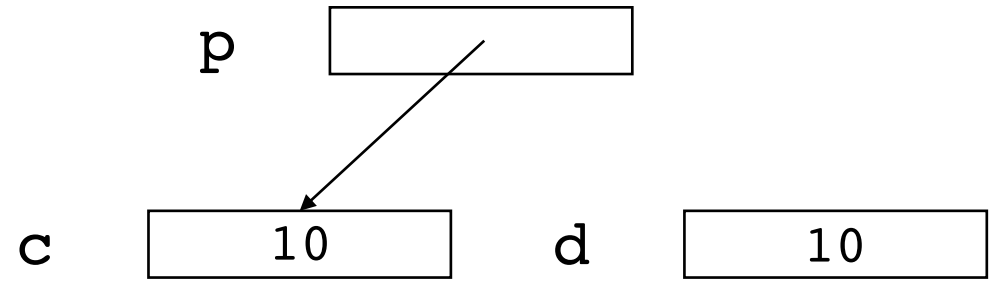
Dereference Operator (*)

- On the left hand side of an assignment, dereferencing a pointer lets you assign into the location pointed to by the pointer
- On the right hand side of an assignment, dereferencing a pointer lets you access the current value in the location it points at

```
int *p;  // p is a pointer to int (a declaration)
int c;
int d;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c through p
d = *p;  // assigns the value from variable c (10) to variable d
```

Dereference Operator (*)

```
int *p;  // p is a pointer to int (a declaration)
int c;
int d;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c via p
d = *p;  // assigns the value from c (10) to d
```



Dereference Operator (*)

```
int *p;  // suppose p is at address 1000 in memory
int c;   // and c is at address 1004 in memory
int d;   // and d is at address 1008 in memory
p = &c;  // stores the address of c in p as a pointer
*p = 10; // assigns the value of 10 to address 1004
d = *p;  // assigns the value from address 1004 to 1008
```

1000	1004
1004	10
1008	10

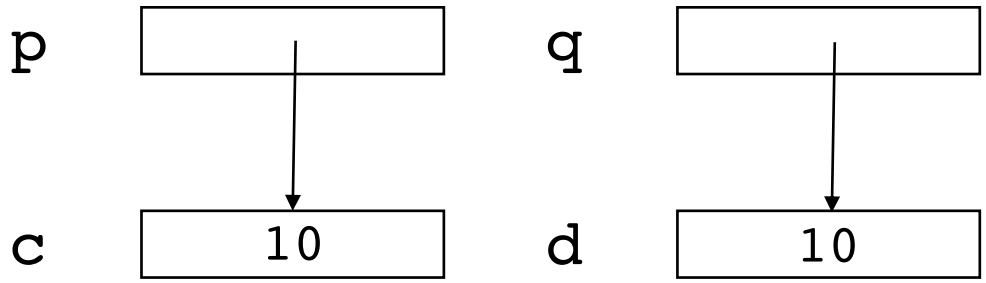
Dereference Operator (*)

- We can also dereference pointers on the left hand side and the right hand side of a single assignment statement

```
int *p, *q;  // p and q are pointers to ints
int c, d;
p = &c;      // causes p to point to c
q = &d;      // causes q to point to d.
c = 10;      // assigns the value of 10 to variable c
*q = *p;     // assigns the value from variable c (10) to variable d
             // through pointers p and q
```

Dereference Operator (*)

```
int *p, *q;   // p and q are pointers to ints
int c, d;
p = &c;       // causes p to point to c
q = &d;       // causes q to point to d
c = 10;       // assigns the value of 10 to variable c
*q = *p;      // assigns the value from c (10) to d
```



Dereference Operator (*)

```
int *p, *q;   // suppose p, q are at 1000 and 1004
int c, d;     // and c, d are at 1008 and 1012
p = &c;       // stores the address of c in p
q = &d;       // stores the address of d in q
c = 10;       // assigns the value of 10 to address 1008
*q = *p;      // assigns the value from 1008 to 1012
```

1000	1008
1004	1012
1008	10
1012	10

Dereference Operator (*)

- One must exercise care when dereferencing a pointer or else various forms of badness can happen. This includes:
- Dereferencing a pointer that hasn't been told to point at something yet:

```
int *p;  // p is a pointer to int (a declaration)
*p = 10; // assigns the value of 10 to what? *Boom!*
```

- Dereferencing a NULL pointer:

```
int *p = NULL;  // p is an integer pointer with a value of NULL
*p = 10;        // assigns the value of 10 to what? *Boom!*
```

- Dereferencing a pointer that points at inaccessible memory (e.g. by doing the bad assignment `p = 10;`)

A Quick Note on Our Pointer Examples

- This is one of those times when we're showing something as an example that you typically wouldn't do in practice
- The code we've been using for the last few slides:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c.
```

is valid code, but pointers should be mainly used for dynamic storage (from the heap) and not generally used to point to local data (from the stack) within a function (like `c` in the above code)

A Quick Note on Our Pointer Examples

- Why?
  - We now have two ways of referring to the same location or variable in memory, and this can lead to confusion and difficulties in understanding and maintaining the code
  - If you were to return the pointer outside of the function or method containing this code and tried to use it, all kinds of bad things could happen as the stack frame is destroyed and the pointer no longer points at what you think it should