How can you learn C++?

Now that you've grasped the basics of starting out on C++, you should aim to expand on it with good tutorials and dedication from your side.

Recommended Books in C++ Programming

It is a must for you to have a good book in hand if you want to better yourself in programming.

It won't just familiarize you with the programming language itself but also teach you to write good code, follow standard principles and understand core programming concepts.

Here are the top 2 books we feel is best for you.

The C++ Programming Language (4th Edition)

The book provides complete guide to C++ language, its features, and the design techniques used. It is authored by the creator of C++ himself, Bjarne Stroustrup.

Programming -- Principles and Practice Using C++ (2nd Edition)

The best book for beginners. It assumes no previous programming experience before and gives you the complete picture of C++ while starting out. A must have.

C++ Coding Standards: 101 Rules, Guidelines, and Best Practices

A must for writing clean and efficient C++ programs. The book is intended to provide beginners with necessary rules, guidelines and coding practices.

Syllabus

Functions

Stored and user define functions, Function Declaration Syntax, Returning a value, Function Overloading, Recursion, Global Variables, Pass by value vs by reference, Returning multiple values, Libraries.

Arrays and Strings

Arrays , declare an array with and without a dimension, 1D and multidimensional arrays, Strings, Character arrays , Special functions provided by the C/C++ libraries

Pointers

Variables and Memory, Motivating Pointers, The Nature of Pointers, Pointer Syntax/Usage, const Pointers, Null, Uninitialized, and Deallocated Pointers, References, The Many Faces of * and &, Pointers and Arrays, Pointer Arithmetic, char * Strings.

User-defined Data types

Representing a (Geometric) Vector, Class, class definition syntax, Fields can have different types, Instances, Declaring an Instance, Accessing Fields, Passing classes to functions, Implementing Methods Separately, Constructors, Access Modifiers, Structs, Default Access Modifiers.

Object-Oriented Programming (OOP) and Inheritance

The Basic Ideas of OOP, Encapsulation, Inheritance, Is-a vs. Has-a, Overriding Methods, Programming by Difference, Access Modifiers and Inheritance, Polymorphism, virtual Functions, Pure virtual Functions, Multiple Inheritance.

Memory Management

Constructors, Scoping and Memory, A Problematic Task, The new operator, The delete operator, Allocating Arrays, Destructor.

Advanced Topics

Templates, Standard Template Library, Operator Overloading, File handling, Reading Strings, Structuring Your Project, Review, References, const, Exceptions, friend Functions/Classes, Preprocessor Macros, Casting.

Lecture 1 Notes: Introduction

1 Compiled Languages and C++

1.1 Why Use a Language Like C++?

At its core, a computer is just a processor with some memory, capable of running tiny instructions like "store 5 in memory location 23459." Why would we express a program as a text file in a programming language, instead of writing processor instructions?

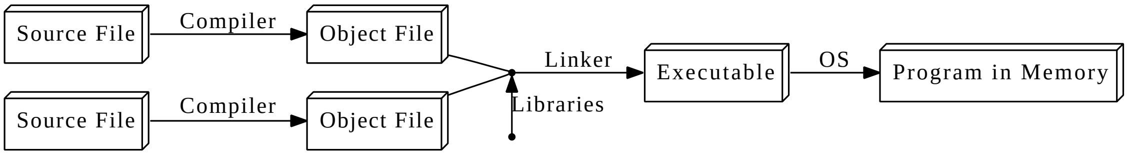
The advantages:

1. **Conciseness:** programming languages allow us to express common sequences of commands more concisely. C++ provides some especially powerful shorthands.
2. **Maintainability:** modifying code is easier when it entails just a few text edits, instead of rearranging hundreds of processor instructions. C++ is *object oriented* (more on that in Lectures 7-8), which further improves maintainability.
3. **Portability:** different processors make different instructions available. Programs written as text can be translated into instructions for many different processors; one of C++'s strengths is that it can be used to write programs for nearly any processor.

C++ is a *high-level* language: when you write a program in it, the shorthands are sufficiently expressive that you don't need to worry about the details of processor instructions. C++ does give access to some lower-level functionality than other languages (e.g. memory addresses).

1.2 The Compilation Process

A program goes from text files (or *source files*) to processor instructions as follows:

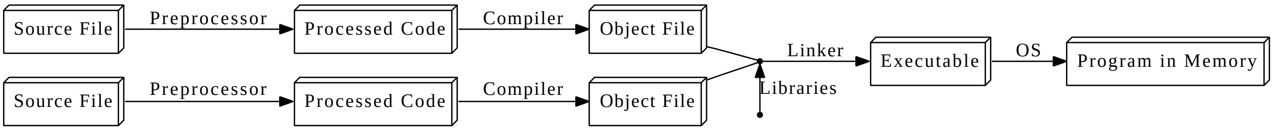


Object files are intermediate files that represent an incomplete copy of the program: each source file only expresses a piece of the program, so when it is compiled into an object file, the object file has some markers indicating which missing pieces it depends on. The linker takes those object files and the compiled libraries of predefined code that they rely on, fills in all the gaps, and spits out the final program, which can then be run by the operating system (OS).

The compiler and linker are just regular programs. The step in the compilation process in which the compiler reads the file is called *parsing*.

In C++, all these steps are performed ahead of time, before you start running a program. In some languages, they are done during the execution process, which takes time. This is one of the reasons C++ code runs far faster than code in many more recent languages.

C++ actually adds an extra step to the compilation process: the code is run through a *preprocessor*, which applies some modifications to the source code, before being fed to the compiler. Thus, the modified diagram is:



1.3 General Notes on C++

C++ is immensely popular, particularly for applications that require speed and/or access to some low-level features. It was created in 1979 by Bjarne Stroustrup, at first as a set of extensions to the C programming language. C++ extends C; our first few lectures will basically be on the C parts of the language.

Though you can write graphical programs in C++, it is much hairier and less portable than text-based (*console*) programs. We will be sticking to console programs in this course.

Everything in C++ is case sensitive: `someName` is not the same as `SomeName`.

2 Hello World

In the tradition of programmers everywhere, we'll use a "Hello, world!" program as an entry point into the basic features of C++.

2.1 The code

```
// A Hello World program
#include <iostream>

```

```
4 int main() {
5   std::cout << "Hello, world!\n";
6
7   return 0;
8 }
```

2.2 Tokens

Tokens are the minimals chunk of program that have meaning to the compiler – the smallest meaningful symbols in the language. Our code displays all 6 kinds of tokens, though the usual use of operators is not present here:

Token type	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	`int, double, for, auto`
Identifiers	Names of things that are not built into the language	`cout, std, x, myFunction`
Literals	Basic constant values whose value is specified directly in the source code	`"Hello, world!", 24.3, 0, ’c’`
Operators	Mathematical or logical operations	`+, -, &&, %, <<`
Punctuation/Separators	Punctuation defining the structure of a program	`{ } ( ) , ;`
Whitespace	Spaces of various sorts; ignored by the compiler	Spaces, tabs, newlines, comments

2.3 Line-By-Line Explanation

1. `//` indicates that everything following it until the end of the line is a comment: it is ignored by the compiler. Another way to write a comment is to put it between `/*` and `*/` (e.g. `x = 1 + /*sneaky comment here*/ 1;`). A comment of this form may span multiple lines. Comments exist to explain non-obvious things going on in the code. Use them: document your code well!

2. Lines beginning with `#` are preprocessor commands, which usually change what code is actually being compiled. `#include` tells the preprocessor to dump in the contents of another file, here the `iostream` file, which defines the procedures for input/output.

4. `int main() {...}` defines the code that should execute when the program starts up. The curly braces represent grouping of multiple commands into a *block*. More about this syntax in the next few lectures.

5. - `cout << :` This is the syntax for outputting some piece of text to the screen.

   - **Namespaces:** In C++, identifiers can be defined within a context – sort of a directory of names – called a *namespace*. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the *scope resolution operator* (`::`). Here, we're telling the compiler to look for `cout` in the `std` namespace, in which many standard C++ identifiers are defined.

     A cleaner alternative is to add the following line below line 2:

     ```
     using namespace std;
     ```

     This line tells the compiler that it should look in the `std` namespace for any identifier we haven't defined. If we do this, we can omit the `std::` prefix when writing `cout`. This is the recommended practice.

   - **Strings:** A sequence of characters such as `Hello, world` is known as a *string*. A string that is specified explicitly in a program is a *string literal*.

   - **Escape sequences:** The `\n` indicates a *newline* character. It is an example of an *escape sequence* – a symbol used to represent a special character in a text literal. Here are all the C++ escape sequences which you can include in strings:

Escape Sequence	Represented Character
`\a`	System bell (beep sound)
`\b`	Backspace
`\f`	Formfeed (page break)
`\n`	Newline (line break)
`\r`	"Carriage return" (returns cursor to start of line)
`\t`	Tab
`\\`	Backslash
`\'`	Single quote character
`\"`	Double quote character
\some integer x	The character represented by x

7. `return 0` indicates that the program should tell the operating system it has completed successfully. This syntax will be explained in the context of functions; for now, just include it as the last line in the `main` block.

Note that every statement ends with a semicolon (except preprocessor commands and blocks using {}). Forgetting these semicolons is a common mistake among new C++ programmers.

3 Basic Language Features

So far our program doesn't do very much. Let's tweak it in various ways to demonstrate some more interesting constructs.

3.1 Values and Statements

First, a few definitions:

- A *statement* is a unit of code that does something – a basic building block of a program.
- An *expression* is a statement that has a *value* – for instance, a number, a string, the sum of two numbers, etc. `4 + 2`, `x - 1`, and `"Hello, world!\n"` are all expressions.

Not every statement is an expression. It makes no sense to talk about the value of an `#include` statement, for instance.

3.2 Operators

We can perform arithmetic calculations with *operators*. Operators act on expressions to form a new expression. For example, we could replace `"Hello, world!\n"` with `(4 + 2) / 3`, which would cause the program to print the number 2. In this case, the `+` operator acts on the expressions `4` and `2` (its *operands*).

Operator types:

- **Mathematical:** `+`, `-`, `*`, `/`, and parentheses have their usual mathematical meanings, including using `-` for negation. `%` (the *modulus* operator) takes the remainder of two numbers: `6 % 5` evaluates to 1.
- **Logical:** used for "and," "or," and so on. More on those in the next lecture.
- **Bitwise:** used to manipulate the binary representations of numbers. We will not focus on these.

3.3 Data Types

Every expression has a type – a formal description of what kind of data its value is. For instance, `0` is an integer, `3.142` is a *floating-point* (decimal) number, and `"Hello, world!\n"` is a *string* value (a sequence of characters). Data of different types take a different amounts of memory to store. Here are the built-in datatypes we will use most often:

Type Names	**Description**	**Size**	**Range**
`char`	Single text character or small integer. Indicated with single quotes (`'a'`, `'3'`).	1 byte	signed: -128 to 127 unsigned: 0 to 255
`int`	Larger integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean (true/false). Indicated with the keywords `true` and `false`.	1 byte	Just `true` (1) or `false` (0).
`double`	"Doubly" precise floating point number.	8 bytes	+/- 1.7e +/- 308 (15 digits)

Notes on this table:

- A *signed* integer is one that can represent a negative number; an *unsigned* integer will never be interpreted as negative, so it can represent a wider range of positive numbers. Most compilers assume signed if unspecified.
- There are actually 3 integer types: `short`, `int`, and `long`, in non-decreasing order of size (`int` is usually a synonym for one of the other two). You generally don't need to worry about which kind to use unless you're worried about memory usage or you're using really huge numbers. The same goes for the 3 floating point types, `float`, `double`, and `long double`, which are in non-decreasing order of precision (there is usually some imprecision in representing real numbers on a computer).
- The sizes/ranges for each type are not fully standardized; those shown above are the ones used on most 32-bit computers.

An operation can only be performed on compatible types. You can add `34` and `3`, but you can't take the remainder of an integer and a floating-point number.

An operator also normally produces a value of the same type as its operands; thus, `1 / 4` evaluates to `0` because with two integer operands, `/` truncates the result to an integer. To get 0.25, you'd need to write something like `1 / 4.0`.

A text string, for reasons we will learn in Lecture 5, has the type `char *`.

4 Variables

We might want to give a value a name so we can refer to it later. We do this using *variables*. A variable is a named location in memory.

For example, say we wanted to use the value `4 + 2` multiple times. We might call it `x` and use it as follows:

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int main() {
 5   int x;
 6   x = 4 + 2;
 7   cout << x / 3 << ' ' << x * 2;
 8
 9   return 0;
10 }
```

(Note how we can print a sequence of values by "chaining" the `<<` symbol.)

The name of a variable is an identifier token. Identifiers may contain numbers, letters, and underscores (`_`), and may not start with a number.

Line 5 is the *declaration* of the variable `x`. We must tell the compiler what type `x` will be so that it knows how much memory to reserve for it and what kinds of operations may be performed on it.

Line 6 is the *initialization* of `x`, where we specify an initial value for it. This introduces a new operator: `=`, the assignment operator. We can also change the value of `x` later on in the code using this operator.

We could replace lines 5 and 6 with a single statement that does both declaration and initialization:

```
  int x = 4 + 2;
```

This form of declaration/initialization is cleaner, so it is to be preferred.

5 Input

Now that we know how to give names to values, we can have the user of the program input values. This is demonstrated in line 6 below:

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int main() {
 5   int x;
 6   cin >> x;
 7
 8   cout << x / 3 << ' ' << x * 2;
 9
10   return 0;
11 }
```

Just as `cout <<` is the syntax for outputting values, `cin >>` (line 6) is the syntax for inputting values.

Memory trick: if you have trouble remembering which way the angle brackets go for `cout` and `cin`, think of them as arrows pointing in the direction of data flow. `cin` represents the terminal, with data flowing from it to your variables; `cout` likewise represents the terminal, and your data flows to it.

6 Debugging

There are two kinds of errors you'll run into when writing C++ programs: *compilation errors* and *runtime errors*. Compilation errors are problems raised by the compiler, generally resulting from violations of the syntax rules or misuse of types. These are often caused by typos and the like. Runtime errors are problems that you only spot when you run the program: you did specify a legal program, but it doesn't do what you wanted it to. These are usually more tricky to catch, since the compiler won't tell you about them.

Lecture 2 Notes: Flow of Control

1 Motivation

Normally, a program executes statements from first to last. The first statement is executed, then the second, then the third, and so on, until the program reaches its end and terminates. A computer program likely wouldn't be very useful if it ran the same sequence of statements every time it was run. It would be nice to be able to change which statements ran and when, depending on the circumstances. For example, if a program checks a file for the number of times a certain word appears, it should be able to give the correct count no matter what file and word are given to it. Or, a computer game should move the player's character around when the player wants. We need to be able to alter the order in which a program's statements are executed, the *control flow*.

2 Control Structures

Control structures are portions of program code that contain statements within them and, depending on the circumstances, execute these statements in a certain way. There are typically two kinds: *conditionals* and *loops.*

2.1 Conditionals

In order for a program to change its behavior depending on the input, there must a way to test that input. Conditionals allow the program to check the values of variables and to execute (or not execute) certain statements. C++ has *if* and *switch-case* conditional structures.

2.1.1 Operators

Conditionals use two kinds of special operators: *relational* and *logical.* These are used to determine whether some condition is true or false.

The relational operators are used to test a relation between two expressions:

Operator	**Meaning**
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

They work the same as the arithmetic operators (e.g., a > b) but return a Boolean value of either `true` or `false`, indicating whether the relation tested for holds. (An expression that returns this kind of value is called a Boolean expression.) For example, if the variables `x` and y have been set to 6 and 2, respectively, then `x > y` returns `true`. Similarly, `x < 5` returns `false`.

The logical operators are often used to combine relational expressions into more complicated Boolean expressions:

Operator	Meaning
&&	and
||	or
!	not

The operators return true or false, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

The ! operator is a unary operator, taking only one argument and negating its value:

a	!a
true	false
false	true

Examples using logical operators (assume x = 6 and y = 2):

```
!(x > 2) → false
(x > y) && (y > 0) → true
(x < y) && (y > 0) → false
(x < y) || (y > 0) → true
```

Of course, Boolean variables can be used directly in these expressions, since they hold true and false values. In fact, any kind of value can be used in a Boolean expression due to a quirk C++ has: false is represented by a value of 0 and anything that is not 0 is true. So, “Hello, world!” is true, 2 is true, and any int variable holding a non-zero value is true. This means !x returns false and x && y returns true!

2.1.2 *if, if-else* and *else if*

The *if* conditional has the form:

```
if(condition)
{
	statement1
	statement2
	…
}
```

The condition is some expression whose value is being tested. If the condition resolves to a value of true, then the statements are executed before the program continues on. Otherwise, the statements are ignored. If there is only one statement, the curly braces may be omitted, giving the form:

```
if(condition)
	statement
```

The *if-else* form is used to decide between two sequences of statements referred to as *blocks*:

```
if(condition)
{
	statementA1
	statementA2
	…
}
else
{
	statementB1
	statementB2
	…
}
```

If the condition is met, the block corresponding to the if is executed. Otherwise, the block corresponding to the else is executed. Because the condition is either satisfied or not, one of the blocks in an if-else *must* execute. If there is only one statement for any of the blocks, the curly braces for that block may be omitted:

```
if(condition)
	statementA1
else
	statementB1
```

The *else if* is used to decide between two or more blocks based on *multiple* conditions:

```
if(condition1)
{
	statementA1
	statementA2
	…
}
else if(condition2)
{
	statementB1
	statementB2
	…
}
```

If condition1 is met, the block corresponding to the if is executed. If not, then *only if* condition2 is met is the block corresponding to the else if executed. There may be more than one else if, each with its own condition. Once a block whose condition was met is executed, any else ifs after it are ignored. Therefore, in an *if-else-if* structure, either one or no block is executed.

An else may be added to the end of an if-else-if. If none of the previous conditions are met, the else block is executed. In this structure, one of the blocks *must* execute, as in a normal if-else.

Here is an example using these control structures:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int x = 6;
6     int y = 2;
7
8     if(x > y)
9           cout << “x is greater than y\n”;
10    else if(y > x)
11          cout << “y is greater than x\n”;
12    else
13          cout << “x and y are equal\n”;
14
15    return 0;
16 }
```

The output of this program is `x is greater than y`. If we replace lines 5 and 6 with

```
int x = 2;
int y = 6;
```

then the output is `y is greater than x`. If we replace the lines with

```
int x = 2;
int y = 2;
```

then the output is `x and y are equal`.

2.1.3 *switch-case*

The *switch-case* is another conditional structure that may or may not execute certain statements. However, the switch-case has peculiar syntax and behavior:

```
switch(expression)
{
      case constant1:
            statementA1
            statementA2
            ...
            break;
      case constant2:
            statementB1
            statementB2
            ...
            break;
      ...
      default:
            statementZ1
            statementZ2
            ...
}
```

The `switch` evaluates `expression` and, if `expression` is equal to `constant1`, then the statements beneath `case constant 1:` are executed until a `break` is encountered. If `expression` is not equal to `constant1`, then it is compared to `constant2`. If these are equal, then the statements beneath `case constant 2:` are executed until a `break` is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath `default:` are executed.

Due to the peculiar behavior of `switch-case`s, curly braces are not necessary for cases where there is more than one statement (but they *are* necessary to enclose the entire `switch-case`). `switch-case`s generally have `if-else` equivalents but can often be a cleaner way of expressing the same behavior.

Here is an example using `switch-case`:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int x = 6;
6
7     switch(x) {
8           case 1:
9                 cout << “x is 1\n”;
10                break;
11          case 2:
12          case 3:
13                cout << "x is 2 or 3";
14                break;
15          default:
16                cout << "x is not 1, 2, or 3";
17    }
18
19    return 0;
20 }
```

This program will print `x is not 1, 2, or 3`. If we replace line 5 with `int x = 2;` then the program will print `x is 2 or 3`.

2.2 Loops

Conditionals execute certain statements if certain conditions are met; loops execute certain statements *while* certain conditions are met. C++ has three kinds of loops: *while, do-while,* and *for.*

2.2.1 *while* and *do-while*

The *while* loop has a form similar to the if conditional:

```
while(condition)
{
      statement1
      statement2
      …
}
```

As long as condition holds, the block of statements will be repeatedly executed. If there is only one statement, the curly braces may be omitted. Here is an example:

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

int main() {
   int x = 0;

   while(x < 10)
         x = x + 1;
```

```

    cout << “x is “ << x << “\n”;

    return 0;
}
```

This program will print `x is 10`.

The *do-while* loop is a variation that guarantees the block of statements will be executed *at least once:*

```
do
{
      statement1
      statement2
      …
}
while(condition);
```

The block of statements is executed and then, if the condition holds, the program returns to the top of the block. Curly braces are *always* required. Also note the semicolon after the `while` condition.

2.2.2 *for*

The *for* loop works like the while loop but with some change in syntax:

```
for(initialization; condition; incrementation)
{
      statement1
      statement2
      …
}
```

The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. Curly braces may be omitted if there is only one statement. Here is an example:

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

int main() {

    for(int x = 0; x < 10; x = x + 1)
        cout << x << “\n”;

    return 0;
}
```

This program will print out the values `0` through `9`, each on its own line.

If the counter variable is already defined, there is no need to define a new one in the initialization portion of the for loop. Therefore, it is valid to have the following:

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

int main() {

   int x = 0;
   for(; x < 10; x = x + 1)
         cout << x << “\n”;

   return 0;
}
```

Note that the first semicolon inside the for loop's parentheses is still required.

A for loop can be expressed as a while loop and vice-versa. Recalling that a for loop has the form

```
for(initialization; condition; incrementation)
{
      statement1
      statement2
      …
}
```

we can write an equivalent while loop as

```
initialization
while(condition)
{
      statement1
      statement2
      …
      incrementation
}
```

Using our example above,

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

int main() {

   for(int x = 0; x < 10; x = x + 1)
         cout << x << “\n”;

   return 0;
}
```

is converted to

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

int main() {

   int x = 0;
   while(x < 10) {
         cout << x << “\n”;
         x = x + 1;
   }

   return 0;
}
```

The incrementation step can technically be anywhere inside the statement block, but it is good practice to place it as the last step, particularly if the previous statements use the current value of the counter variable.

2.3 Nested Control Structures

It is possible to place ifs inside of ifs and loops inside of loops by simply placing these structures inside the statement blocks. This allows for more complicated program behavior.

Here is an example using nesting if conditionals:

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

int main() {
   int x = 6;
   int y = 0;

   if(x > y) {
         cout << “x is greater than y\n”;
         if(x == 6)
               cout << “x is equal to 6\n”;
         else
               cout << “x is not equalt to 6\n”;
   } else
         cout << “x is not greater than y\n”;

   return 0;
}
```

This program will print `x is greater than y` on one line and then `x is equal to 6` on the next line.

Here is an example using nested loops:

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

int main() {
   for(int x = 0; x < 4; x = x + 1) {
         for(int y = 0; y < 4; y = y + 1)
               cout << y;
         cout << “\n”;
   }

   return 0;
}
```

This program will print four lines of `0123`.

Lecture 3: Functions

How to reuse code

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

int main() {
 int threeExpFour = 1;
 for (int i = 0; i < 4; i = i + 1) {
  threeExpFour = threeExpFour * 3;
 }
 cout << "3^4 is " << threeExpFour << endl;
 return 0;
}
```

Copy-paste coding

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

int main() {
  int threeExpFour = 1;
  for (int i = 0; i < 4; i = i + 1) {
    threeExpFour = threeExpFour * 3;
  }
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = 1;
  for (int i = 0; i < 5; i = i + 1) {
    sixExpFive = sixExpFive * 6;
  }
  cout << "6^5 is " << sixExpFive << endl;
  return 0;
}
```

Copy-paste coding (bad)

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

int main() {
  int threeExpFour = 1;
  for (int i = 0; i < 4; i = i + 1) {
    threeExpFour = threeExpFour * 3;
  }
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = 1;
  for (int i = 0; i < 5; i = i + 1) {
    sixExpFive = sixExpFive * 6;
  }
  cout << "6^5 is " << sixExpFive << endl;
  int twelveExpTen = 1;
  for (int i = 0; i < 10; i = i + 1) {
    twelveExpTen = twelveExpTen * 12;
  }
  cout << "12^10 is " << twelveExpTen << endl;
  return 0;
}
```

With a function

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  return 0;
}
```

With a function

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = raiseToPower(6, 5);
  cout << "6^5 is " << sixExpFive << endl;
  return 0;
}
```

With a function

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = raiseToPower(6, 5);
  cout << "6^5 is " << sixExpFive << endl;
  int twelveExpTen = raiseToPower(12, 10);
  cout << "12^10 is " << twelveExpTen << endl;
  return 0;
}
```

Why define your own functions?

- Readability: sqrt(5) is clearer than copy-pasting in an algorithm to compute the square root
- Maintainability: To change the algorithm, just change the function (vs changing it everywhere you ever used it)
- Code reuse: Lets other people use algorithms you’ve implemented

Function Declaration Syntax

Function name

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

Function Declaration Syntax

Return type

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

Function Declaration Syntax

Argument 1

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

- Argument order matters:
  - raiseToPower(2,3) is 2^3=8
  - raiseToPower(3,2) is 3^2=9

Function Declaration Syntax

Argument 2

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

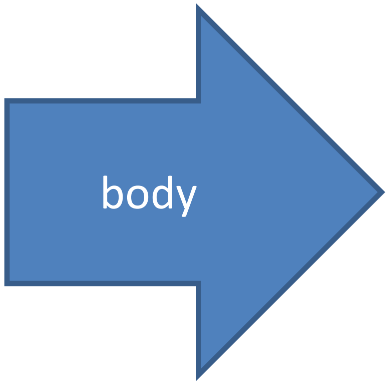
- Argument order matters:
  - raiseToPower(2,3) is 2^3=8
  - raiseToPower(3,2) is 3^2=9

Function Declaration Syntax

signature

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

Function Declaration Syntax



body

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

Function Declaration Syntax

```
int raiseToPower(int base, int exponent)
{
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

Return statement

Function declaration

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

int raiseToPower(int base, int exponent) {
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}
```

Function invocation

```
int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  return 0;
}
```

Returning a value

- Up to one value may be returned; it must be the same type as the return type.

```
int foo()
{
  return "hello"; // error
}
```

```
char* foo()
{
  return "hello"; // ok
}
```

Returning a value

- Up to one value may be returned; it must be the same type as the return type.
- If no values are returned, give the function a **void** return type

```
void printNumber(int num) {
  cout << "number is " << num << endl;
}

int main() {
  printNumber(4); // number is 4
  return 0;
}
```

Returning a value

- Up to one value may be returned; it must be the same type as the return type.
- If no values are returned, give the function a **void** return type
  - Note that you cannot declare a variable of type void

```
int main() {
  void x; // ERROR
  return 0;
}
```

Returning a value

- Return statements don’t necessarily need to be at the end.
- Function returns as soon as a return statement is executed.

```
void printNumberIfEven(int num) {
  if (num % 2 == 1) {
    cout << "odd number" << endl;
    return;
  }
  cout << "even number; number is " << num << endl;
}

int main() {
  int x = 4;
  printNumberIfEven(x);
  // even number; number is 3
  int y = 5;
  printNumberIfEven(y);
  // odd number
}
```

Argument Type Matters

```
void printOnNewLine(int x)
{
    cout << x << endl;
}
```

- `printOnNewLine(3)` works
- `printOnNewLine("hello")` will not compile

Argument Type Matters

```
void printOnNewLine(char *x)
{
    cout << x << endl;
}
```

- `printOnNewLine(3)` will not compile
- `printOnNewLine("hello")` works

Argument Type Matters

```
void printOnNewLine(int x)
{
    cout << x << endl;
}

void printOnNewLine(char *x)
{
    cout << x << endl;
}
```

- `printOnNewLine(3)` works
- `printOnNewLine("hello")` also works

Function Overloading

```
void printOnNewLine(int x)
{
    cout << "Integer: " << x << endl;
}

void printOnNewLine(char *x)
{
    cout << "String: " << x << endl;
}
```

- Many functions with the same name, but different arguments
- The function called is the one whose arguments match the invocation

Function Overloading

```
void printOnNewLine(int x)
{
    cout << "Integer: " << x << endl;
}

void printOnNewLine(char *x)
{
    cout << "String: " << x << endl;
}
```

- printOnNewLine(3) prints “Integer: 3”
- printOnNewLine(“hello”) prints “String: hello”

Function Overloading

```
void printOnNewLine(int x)
{
    cout << "1 Integer: " << x << endl;
}

void printOnNewLine(int x, int y)
{
    cout << "2 Integers: " << x << " and " << y << endl;
}
```

- printOnNewLine(3) prints “1 Integer: 3”
- printOnNewLine(2, 3) prints “2 Integers: 2 and 3”

- Function declarations need to occur before invocations

```
int foo()
{
    return bar()*2; // ERROR - bar hasn’t been declared yet
}

int bar()
{
    return 3;
}
```

- Function declarations need to occur before invocations
  - Solution 1: reorder function declarations

```
int bar()
{
    return 3;
}

int foo()
{
    return bar()*2; // ok
}
```

- Function declarations need to occur before invocations
  - Solution 1: reorder function declarations
  - Solution 2: use a function prototype; informs the compiler you’ll implement it later

```
int bar();

int foo()
{
    return bar()*2; // ok
}

int bar()
{
    return 3;
}
```

function prototype

- Function prototypes should match the signature of the method, though argument names don’t matter

```
int square(int);
```

function prototype

```
int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;
}
```

- Function prototypes should match the signature of the method, though argument names don't matter

```
int square(int x);
```

function prototype

```
int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;
}
```

- Function prototypes should match the signature of the method, though argument names don’t matter

```
int square(int z);
```

function prototype

```
int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;
}
```

- Function prototypes are generally put into separate header files
  - Separates specification of the function from its implementation

```cpp
// myLib.h - header
// contains prototypes

int square(int);
int cube (int);
```

```cpp
// myLib.cpp - implementation
#include "myLib.h"

int cube(int x)
{
	return x*square(x);
}

int square(int x)
{
	return x*x;
}
```

Recursion

- Functions can call themselves.
- fib(n) = fib(n-1) + fib(n-2) can be easily expressed via a recursive implementation

```
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

Recursion

- Functions can call themselves.
- fib(n) = fib(n-1) + fib(n-2) can be easily expressed via a recursive implementation

base case

```
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

Recursion

- Functions can call themselves.
- fib(n) = fib(n-1) + fib(n-2) can be easily expressed via a recursive implementation

recursive step

```
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

Global Variables

- How many times is function foo() called? Use a global variable to determine this.
  - Can be accessed from any function

```
int numCalls = 0;

void foo() {
  ++numCalls;
}

int main() {
  foo(); foo(); foo();
  cout << numCalls << endl; // 3
}
```

Global variable

Scope

- Scope: where a variable was declared, determines where it can be accessed from

```
int numCalls = 0;

int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}

int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  return result;
}
```

Scope

- Scope: where a variable was declared, determines where it can be accessed from
- numCalls has global scope – can be accessed from any function

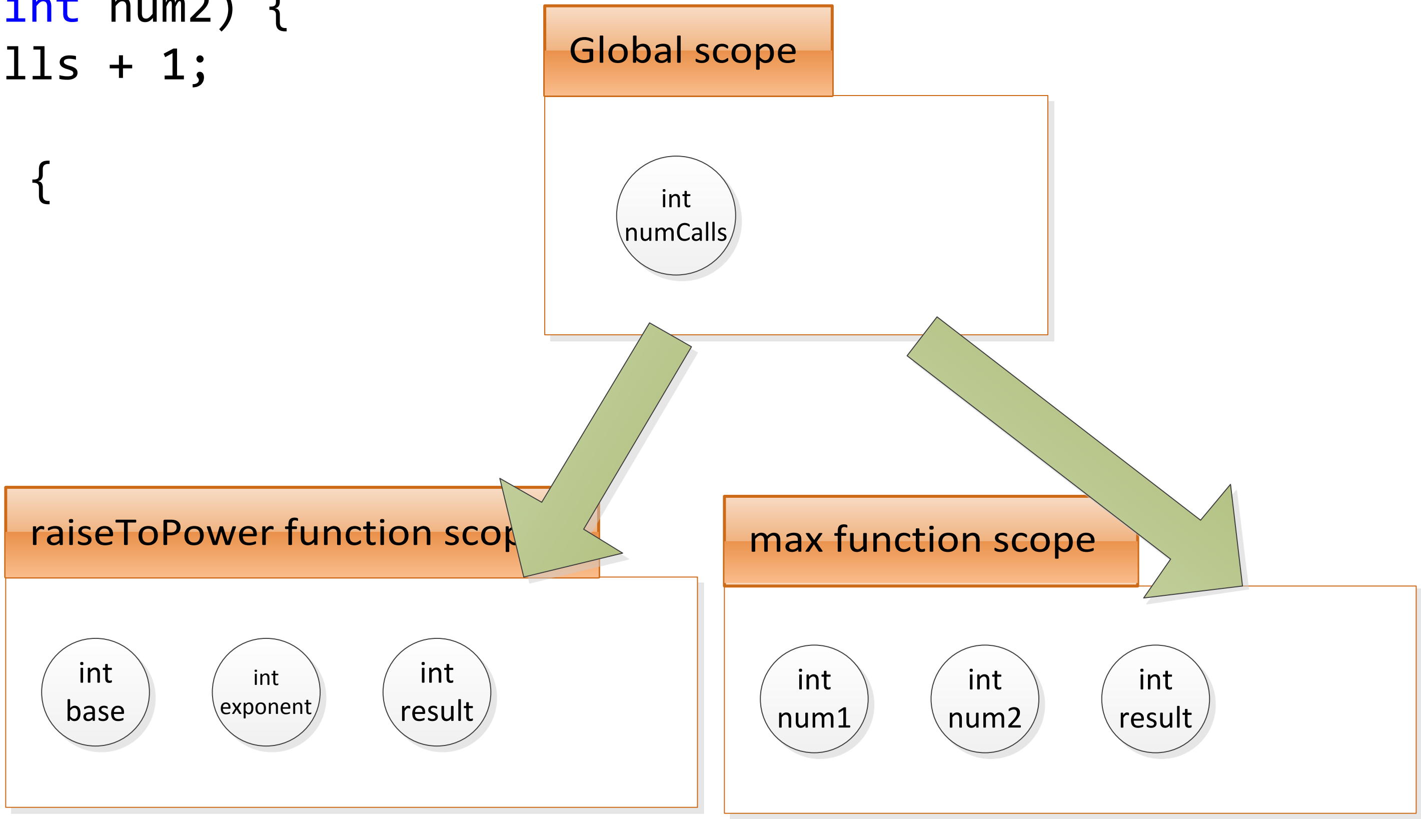
```
int numCalls = 0;

int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}

int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  return result;
}
```

Scope

- Scope: where a variable was declared, determines where it can be accessed from
- numCalls has global scope – can be accessed from any function
- result has function scope – each function can have its own separate variable named result

```
int numCalls = 0;

int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}

int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  return result;
}
```

```
int numCalls = 0;
int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  // A
  return result;
}
int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  // B
  return result;
}
```



Global scope

int numCalls

raiseToPower function scope

int base

int exponent

int result

max function scope

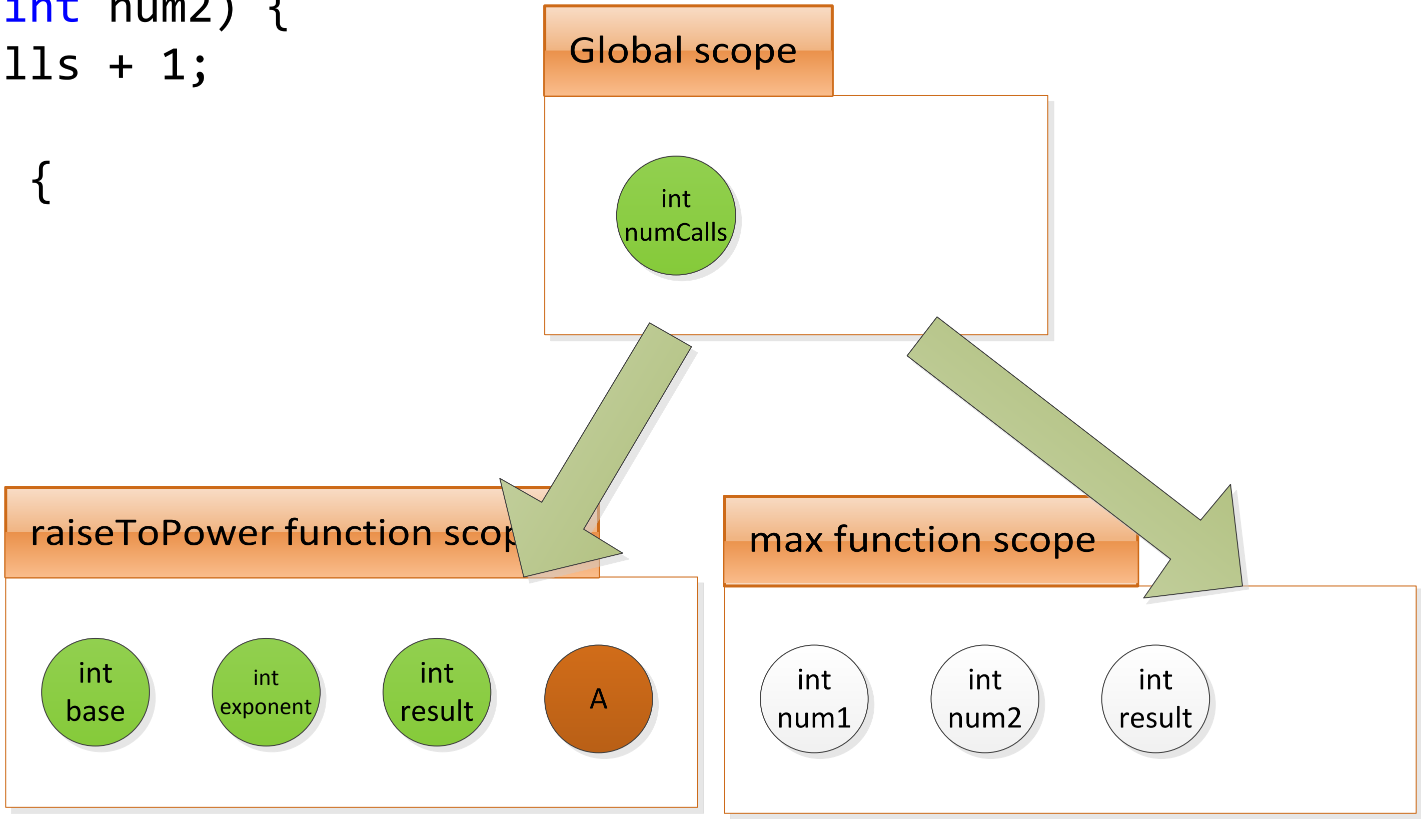
int num1

int num2

int result

```
int numCalls = 0;
int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  // A
  return result;
}
int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  // B
  return result;
}
```

- At A, variables marked in green are in scope



Global scope

int numCalls

raiseToPower function scope

int base

int exponent

int result

A

max function scope

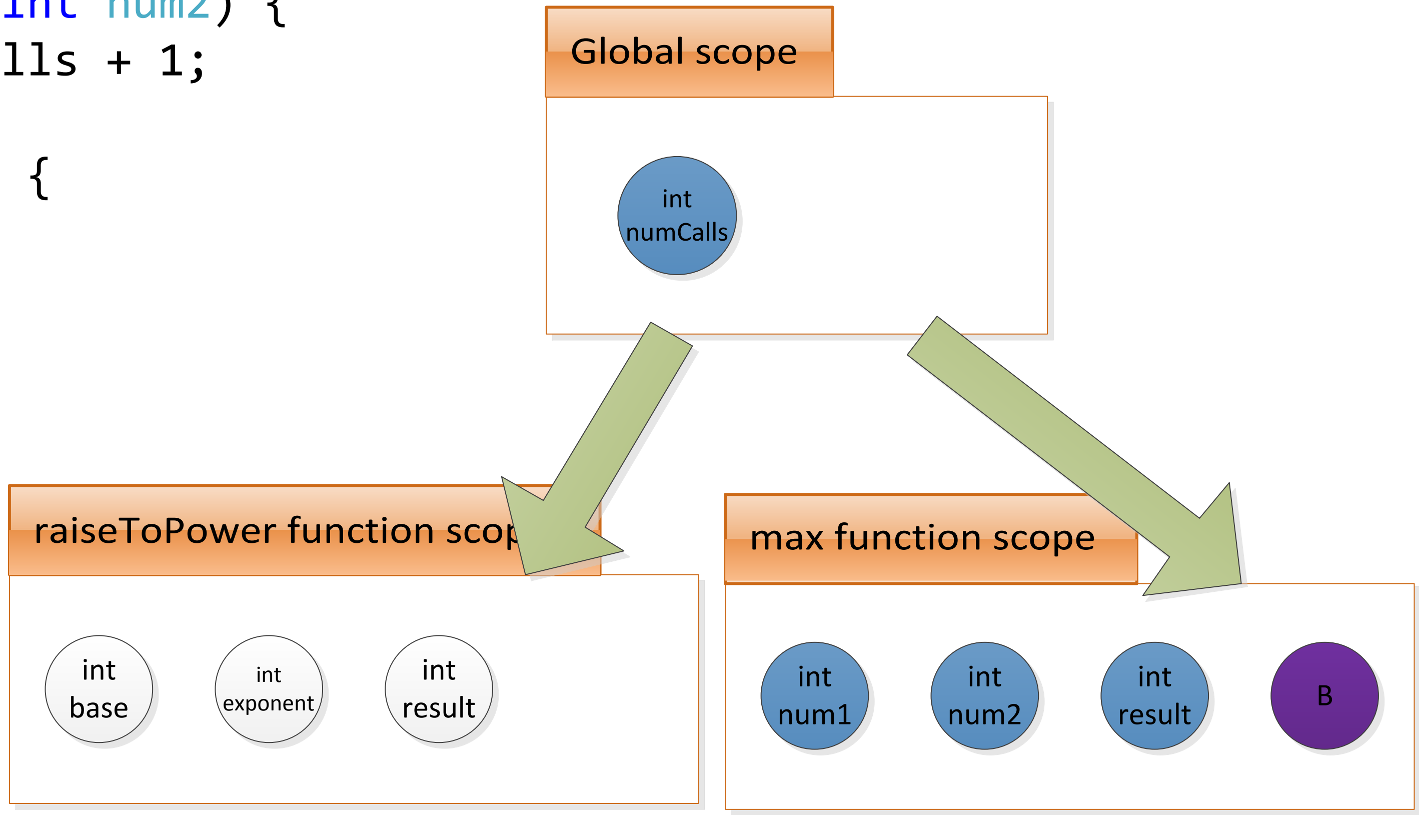
int num1

int num2

int result

```
int numCalls = 0;
int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  // A
  return result;
}
int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  // B
  return result;
}
```

- At B, variables marked in blue are in scope



Global scope

int numCalls

raiseToPower function scope

int base

int exponent

int result

max function scope

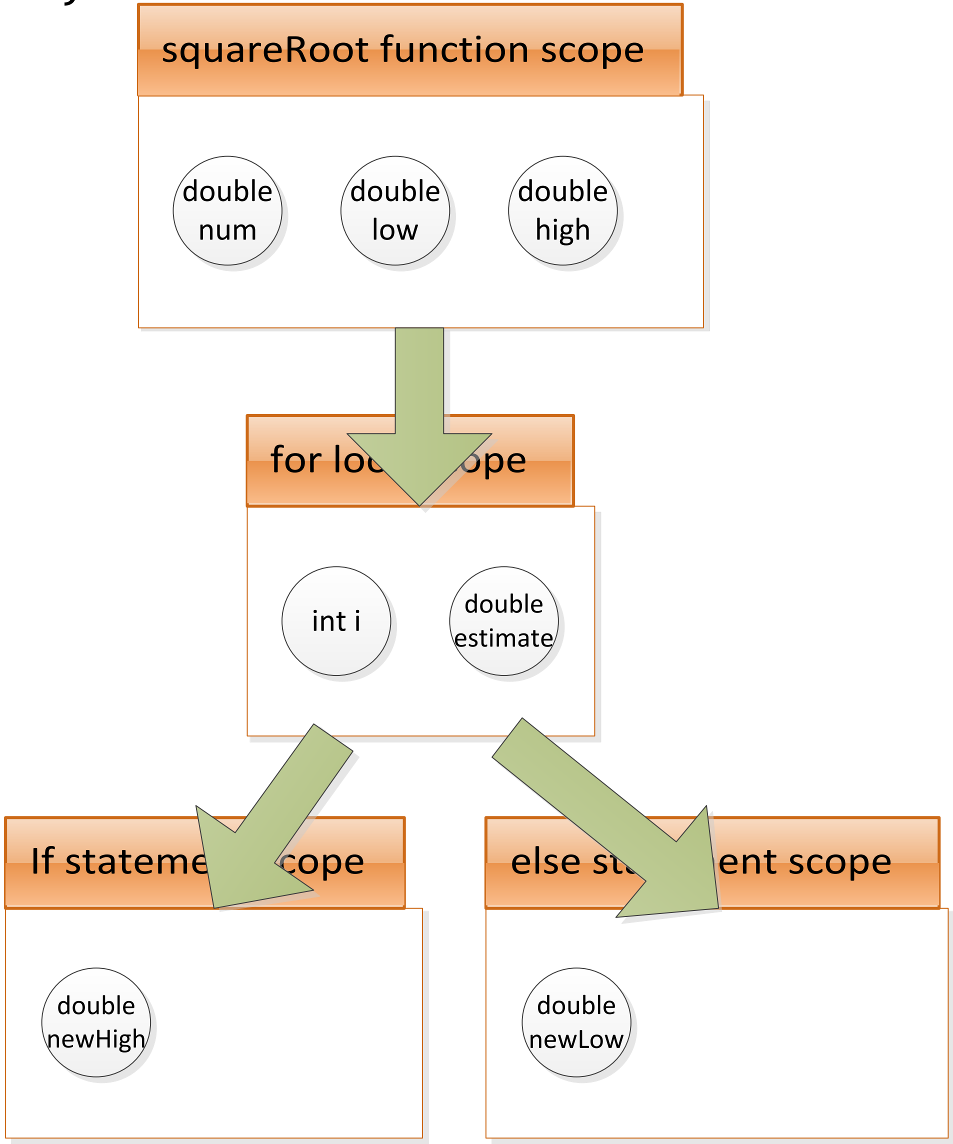
int num1

int num2

int result

B

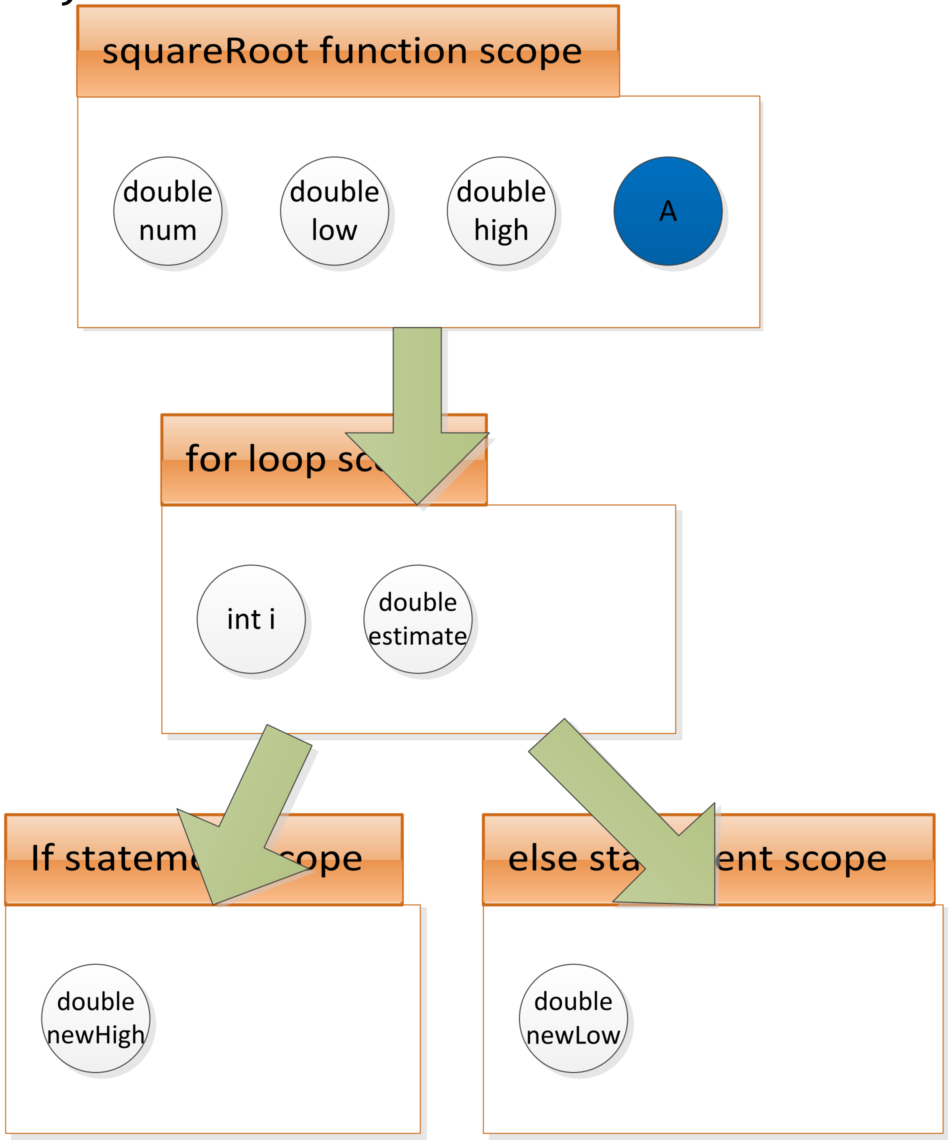
```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  for (int i = 0; i < 30; i = i + 1) {
    double estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
  }
  return (high + low) / 2;
}
```



- Loops and if/else statements also have their own scopes
  - Loop counters are in the same scope as the body of the for loop

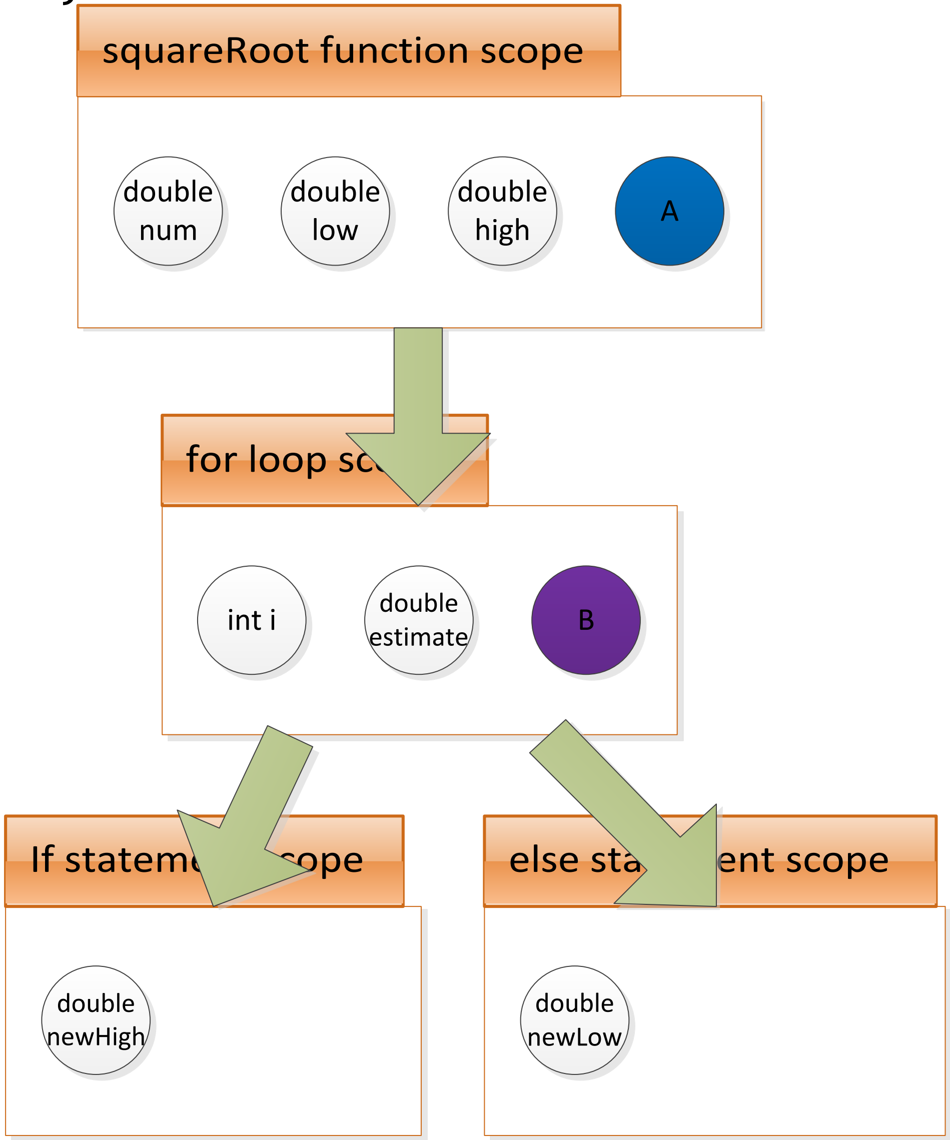
```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  for (int i = 0; i < 30; i = i + 1) {
    double estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
  }
  // A
  return estimate; // ERROR
}
```

- Cannot access variables that are out of scope

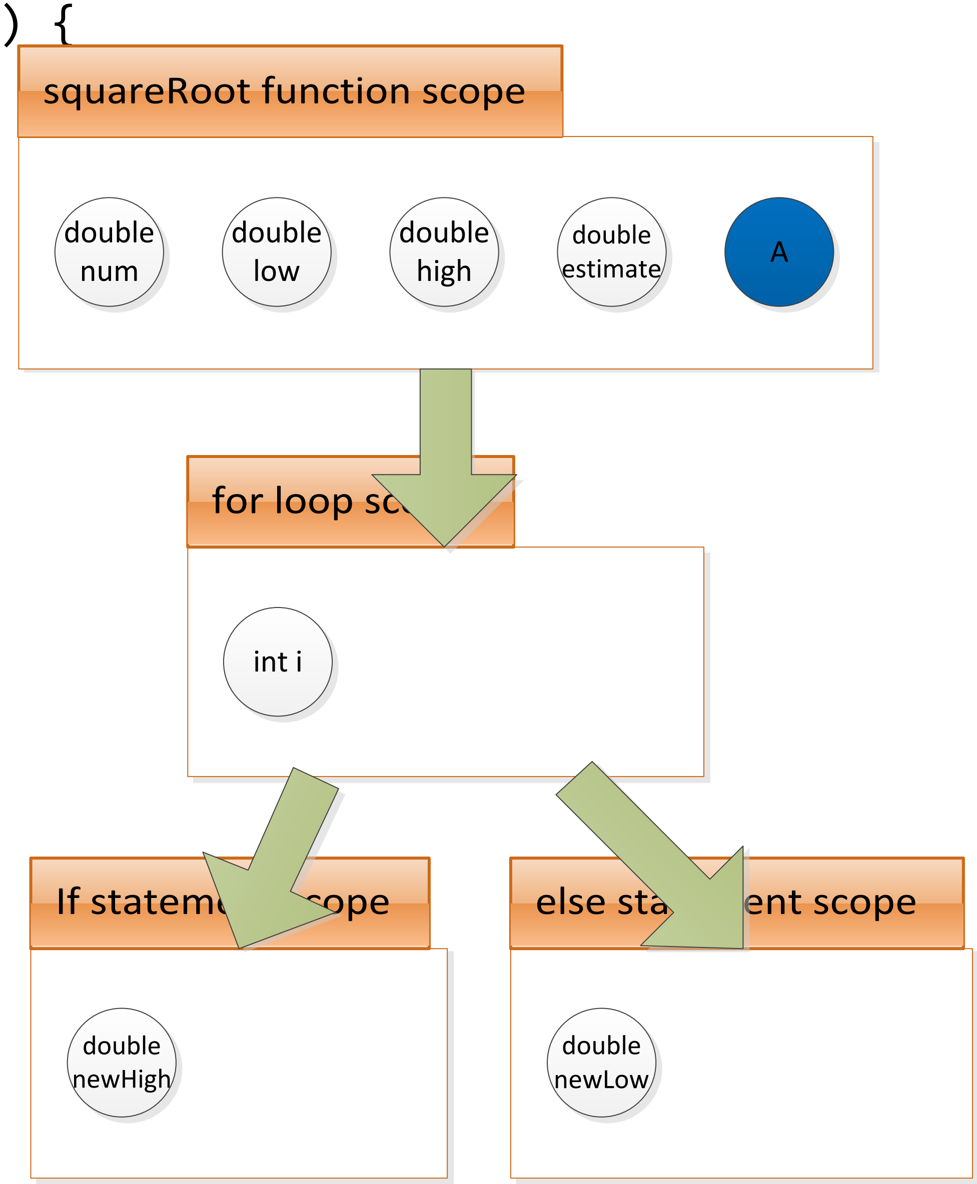


```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  for (int i = 0; i < 30; i = i + 1) {
    double estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
    if (i == 29)
      return estimate; // B
  }
  return -1; // A
}
```

- Cannot access variables that are out of scope
- Solution 1: move the code



```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  double estimate;
  for (int i = 0; i < 30; i = i + 1) {
    estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
  }
  return estimate; // A
}
```



- Cannot access variables that are out of scope
- Solution 2: declare the variable in a higher scope

Pass by value vs by reference

- So far we’ve been passing everything by value – makes a copy of the variable; changes to the variable within the function don’t occur outside the function

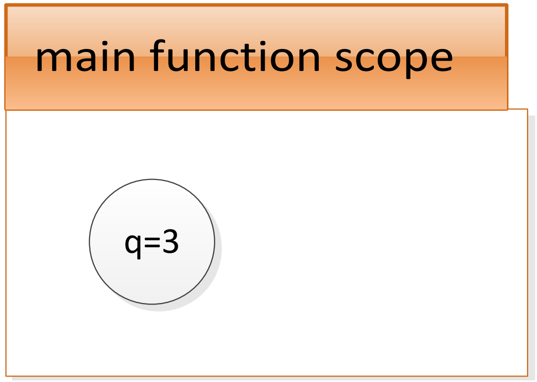
```
// pass-by-value
void increment(int a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

Output

a in increment 4

q in main **3**

Pass by value vs by reference



main function scope

q=3

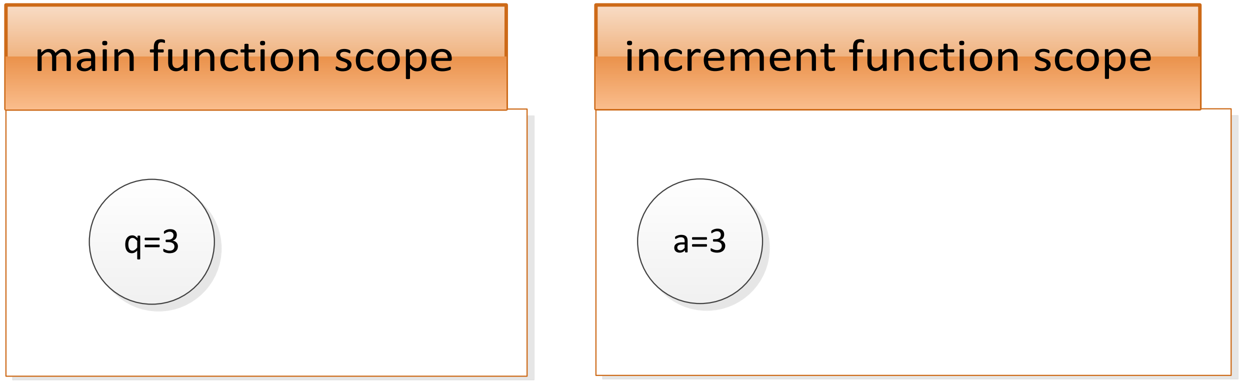
```
// pass-by-value
void increment(int a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3; // HERE
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

Output

a in increment 4

q in main **3**

Pass by value vs by reference

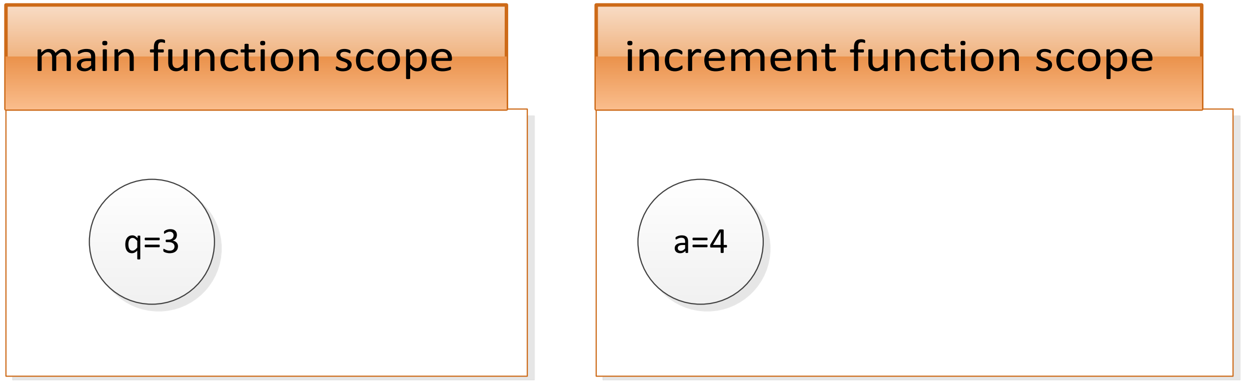


```
// pass-by-value
void increment(int a) { // HERE
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

Output

```
a in increment 4
q in main 3
```

Pass by value vs by reference



```
// pass-by-value
void increment(int a) {
  a = a + 1; // HERE
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main **3**

Pass by value vs by reference

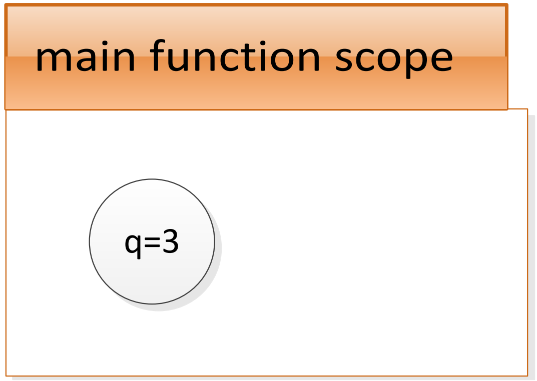
- If you want to modify the original variable as opposed to making a copy, pass the variable by reference (**int &a** instead of **int a**)

```
// pass-by-value
void increment(int &a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

Output

```
a in increment 4
q in main 4
```

Pass by value vs by reference


main function scope

q=3

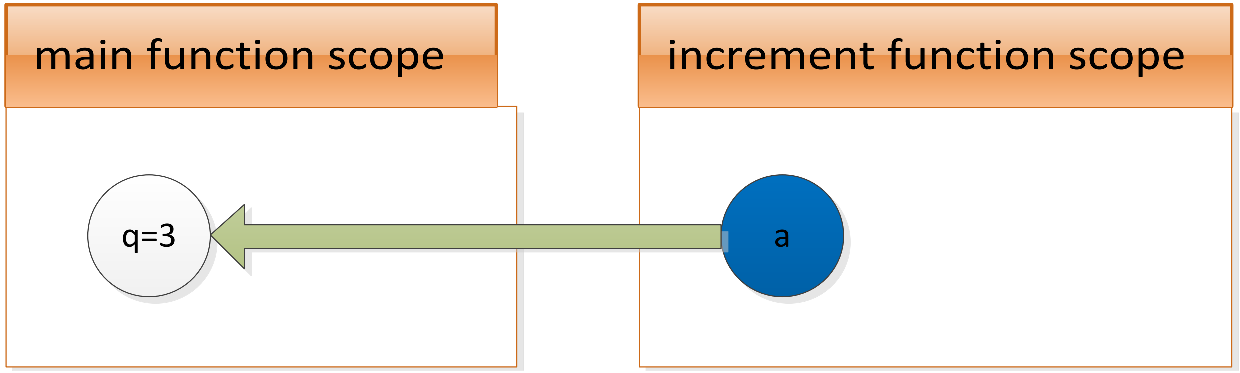
```
// pass-by-value
void increment(int &a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3; // HERE
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

Output

a in increment 4

q in main **4**

Pass by value vs by reference



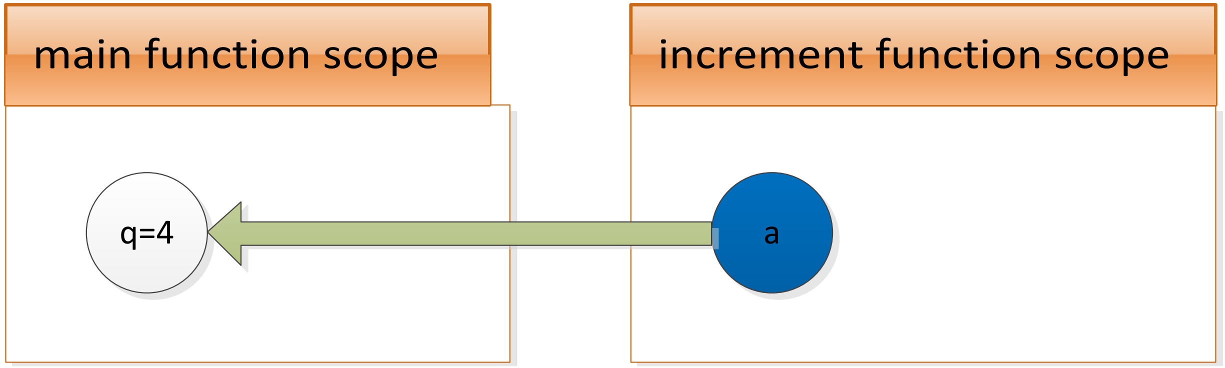
```
// pass-by-value
void increment(int &a) { // HERE
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

Output

a in increment 4

q in main **4**

Pass by value vs by reference



```
// pass-by-value
void increment(int &a) {
  a = a + 1; // HERE
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

Output

a in increment 4

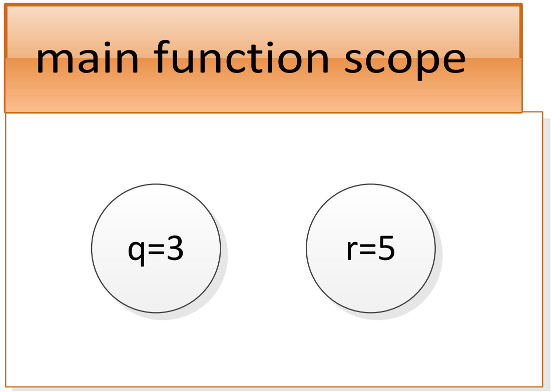
q in main **4**

Implementing Swap

```
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```

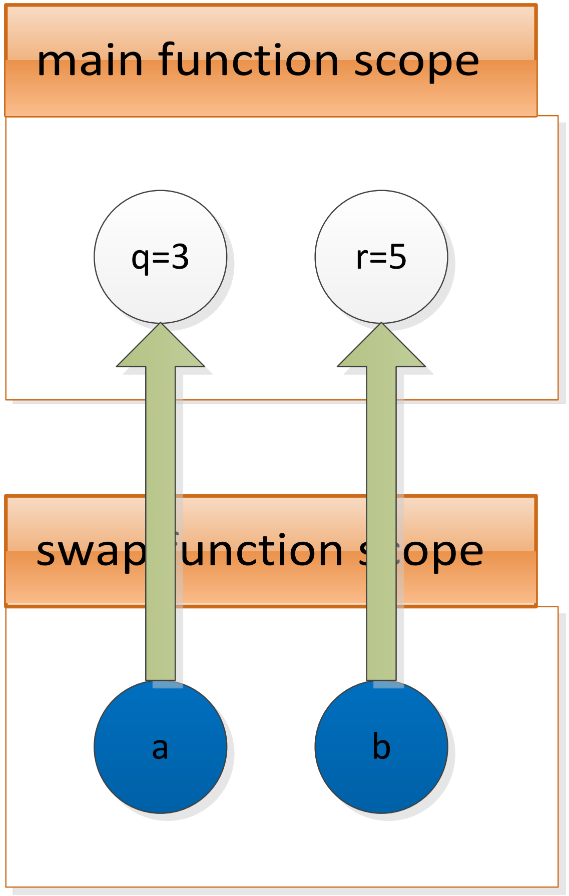
Implementing Swap

```
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5; // HERE
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



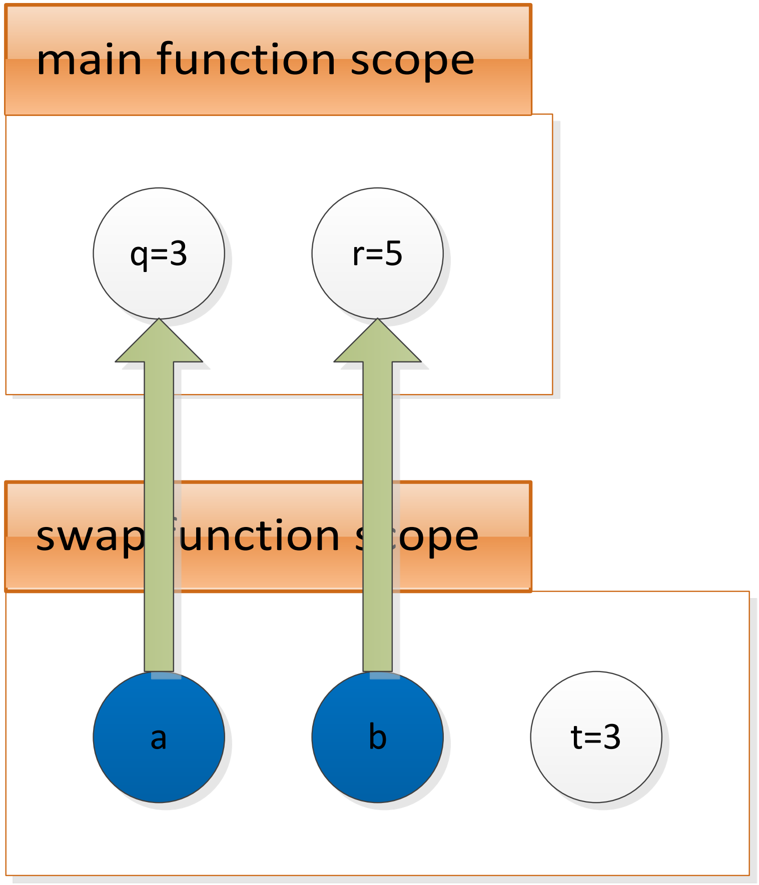
Implementing Swap

```
void swap(int &a, int &b) { // HERE
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



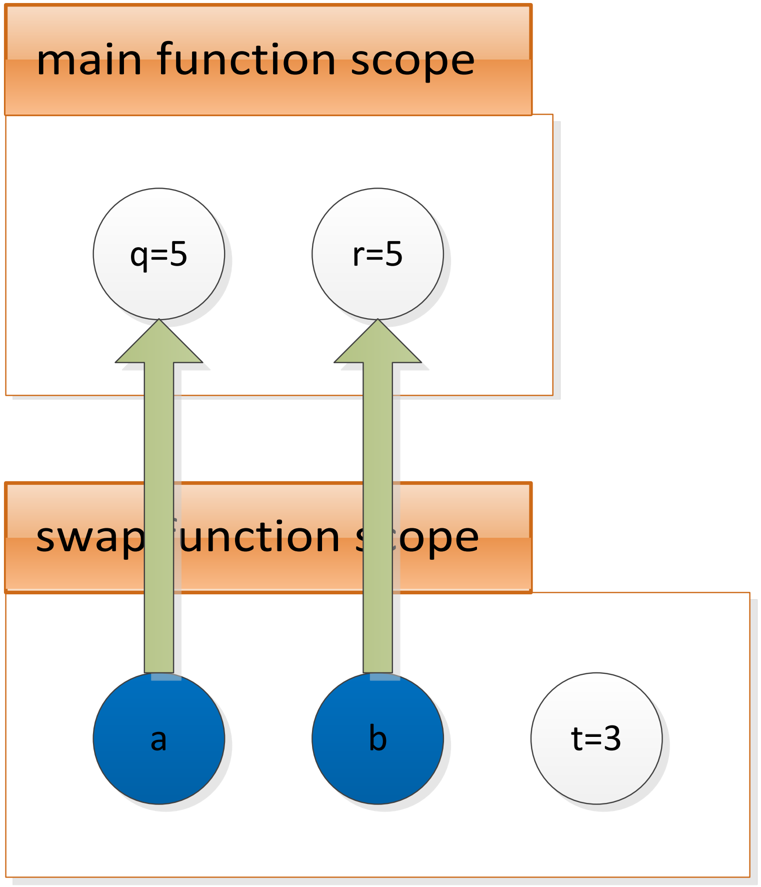
Implementing Swap

```
void swap(int &a, int &b) {
  int t = a; // HERE
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



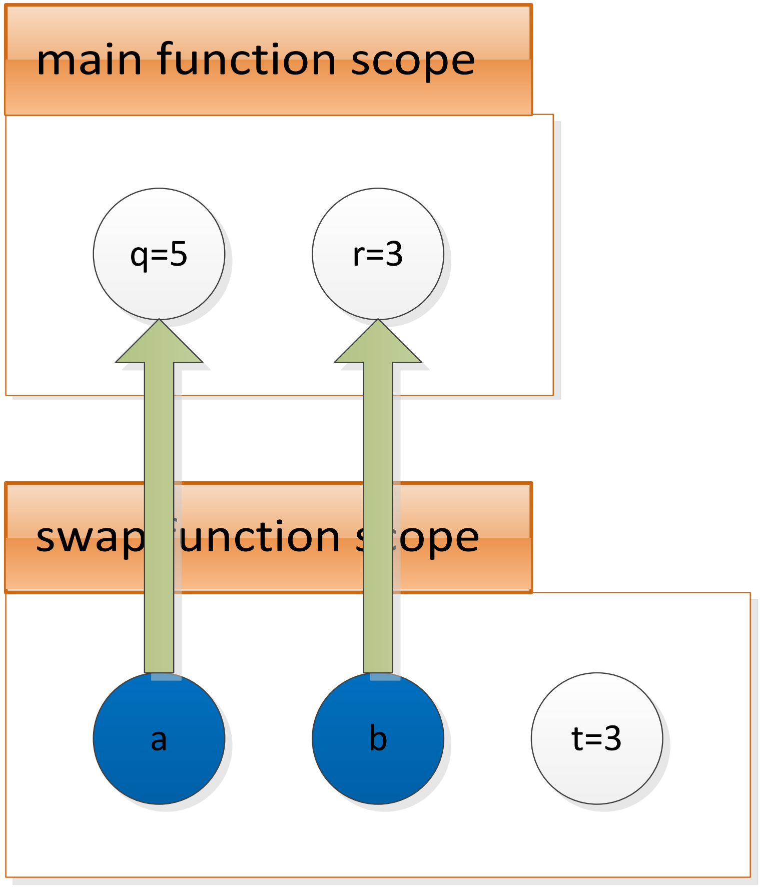
Implementing Swap

```
void swap(int &a, int &b) {
  int t = a;
  a = b; // HERE
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



Implementing Swap

```
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t; // HERE
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



Returning multiple values

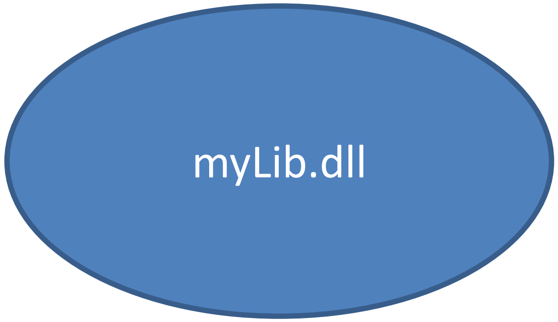
- The return statement only allows you to return 1 value. Passing output variables by reference overcomes this limitation.

```
int divide(int numerator, int denominator, int &remainder) {
  remainder = numerator % denominator;
  return numerator / denominator;
}

int main() {
  int num = 14;
  int den = 4;
  int rem;
  int result = divide(num, den, rem);
  cout << result << "*" << den << "+" << rem << "=" << num << endl;
  // 3*4+2=12
}
```

Libraries

- Libraries are generally distributed as the header file containing the prototypes, and a binary .dll/.so file containing the (compiled) implementation
  - – Don’t need to share your .cpp code

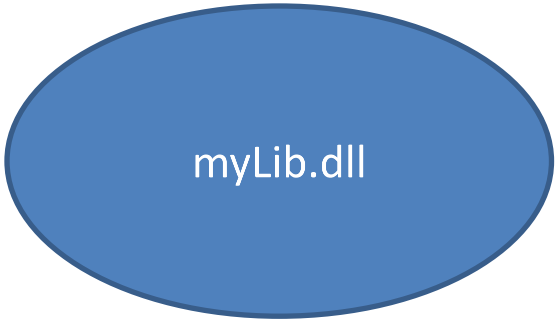
```
// myLib.h – header
// contains prototypes
double squareRoot(double num);
```



myLib.dll

- Library user only needs to know the function prototypes (in the header file), not the implementation source code (in the .cpp file)
  - The **Linker** (part of the compiler) takes care of locating the implementation of functions in the .dll file at compile time

```
// myLib.h – header
// contains prototypes
double squareRoot(double num);
```


myLib.dll

```
// libraryUser.cpp – some other guy’s code
#include "myLib.h"

double fourthRoot(double num) {
  return squareRoot(squareRoot(num));
}
```

Final Notes

- You don’t actually need to implement raiseToPower and squareRoot yourself; cmath (part of the standard library) contains functions **pow** and **sqrt**

```
#include <cmath>

double fourthRoot(double num) {
  return sqrt(sqrt(num));
}
```

Lecture 4 Notes: Arrays and Strings

1 Arrays

So far we have used variables to store values in memory for later reuse. We now explore a means to store *multiple* values together as one unit, the *array.*

An array is a fixed number of *elements* of the same type stored sequentially in memory. Therefore, an integer array holds some number of integers, a character array holds some number of characters, and so on. The size of the array is referred to as its *dimension.* To declare an array in C++, we write the following:

```
type arrayName[dimension];
```

To declare an integer array named `arr` of four elements, we write `int arr[4];`

The elements of an array can be accessed by using an *index* into the array. Arrays in C++ are zero-indexed, so the first element has an index of 0. So, to access the third element in arr, we write `arr[2];` The value returned can then be used just like any other integer.

Like normal variables, the elements of an array must be initialized before they can be used; otherwise we will almost certainly get unexpected results in our program. There are several ways to initialize the array. One way is to declare the array and then initialize some or all of the elements:

```
int arr[4];

arr[0] = 6;
arr[1] = 0;
arr[2] = 9;
arr[3] = 6;
```

Another way is to initialize some or all of the values at the time of declaration:

```
int arr[4] = { 6, 0, 9, 6 };
```

Sometimes it is more convenient to leave out the size of the array and let the compiler determine the array's size for us, based on how many elements we give it:

```
int arr[] = { 6, 0, 9, 6, 2, 0, 1, 1 };
```

Here, the compiler will create an integer array of dimension 8.

The array can also be initialized with values that are not known beforehand:

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

int main() {

    int arr[4];
    cout << “Please enter 4 integers:“ << endl;

    for(int i = 0; i < 4; i++)
            cin >> arr[i];

```

```
   cout << “Values in array are now:“;

   for(int i = 0; i < 4; i++)
         cout << “ “ << arr[i];

   cout << endl;

   return 0;
}
```

Note that when accessing an array the index given must be a positive integer from 0 to n-1, where n is the dimension of the array. The index itself may be directly provided, derived from a variable, or computed from an expression:

```
arr[5];
arr[i];
arr[i+3];
```

Arrays can also be passed as arguments to functions. When declaring the function, simply specify the array as a parameter, without a dimension. The array can then be used as normal within the function. For example:

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

int sum(const int array[], const int length) {
   long sum = 0;
   for(int i = 0; i < length; sum += array[i++]);
   return sum;
}

int main() {
   int arr[] = {1, 2, 3, 4, 5, 6, 7};
   cout << "Sum: " << sum(arr, 7) << endl;
   return 0;
}
```

The function `sum` takes a constant integer array and a constant integer length as its arguments and adds up `length` elements in the array. It then returns the sum, and the program prints out `Sum: 28`.

It is important to note that arrays are *passed by reference* and so any changes made to the array within the function will be observed in the calling scope.

C++ also supports the creation of multidimensional arrays, through the addition of more than one set of brackets. Thus, a two-dimensional array may be created by the following:

```
type arrayName[dimension1][dimension2];
```

The array will have *dimension1* x *dimension2* elements of the same type and can be thought of as an array of arrays. The first index indicates which of *dimension1* subarrays to access, and then the second index accesses one of *dimension2* elements within that subarray. Initialization and access thus work similarly to the one-dimensional case:

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

int main() {
   int twoDimArray[2][4];
   twoDimArray[0][0] = 6;
   twoDimArray[0][1] = 0;
   twoDimArray[0][2] = 9;
```

```
    twoDimArray[0][3] = 6;
    twoDimArray[1][0] = 2;
    twoDimArray[1][1] = 0;
    twoDimArray[1][2] = 1;
    twoDimArray[1][3] = 1;

    for(int i = 0; i < 2; i++)
        for(int j = 0; j < 4; j++)
            cout << twoDimArray[i][j];

    cout << endl;
    return 0;
}
```

The array can also be initialized at declaration in the following ways:

```
int twoDimArray[2][4] = { 6, 0, 9, 6, 2, 0, 1, 1 };

int twoDimArray[2][4] = { { 6, 0, 9, 6 } , { 2, 0, 1, 1 } };
```

Note that dimensions must *always* be provided when initializing multidimensional arrays, as it is otherwise impossible for the compiler to determine what the intended element partitioning is. For the same reason, when multidimensional arrays are specified as arguments to functions, all dimensions but the first *must* be provided (the first dimension is optional), as in the following:

```
int aFunction(int arr[][4]) { … }
```

Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory. Declaring `int arr[2][4];` is the same thing as declaring `int arr[8];`.

2 Strings

String literals such as “`Hello, world!`” are actually represented by C++ as a sequence of characters in memory. In other words, a string is simply a character array and can be manipulated as such.

Consider the following program:

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

int main() {
    char helloworld[] = {   'H', 'e', 'l', 'l', 'o', ',', ' ',
                            'w', 'o', 'r', 'l', 'd', '!', '\0' };

    cout << helloworld << endl;

    return 0;
}
```

This program prints `Hello, world!` Note that the character array `helloworld` ends with a special character known as the *null character*. This character is used to indicate the end of the string.

Character arrays can also be initialized using string literals. In this case, no null character is needed, as the compiler will automatically insert one:

```
char helloworld[] = “Hello, world!”;
```

The individual characters in a string can be manipulated either directly by the programmer or by using special functions provided by the C/C++ libraries. These can be included in a program through the use of the `#include` directive. Of particular note are the following:

- cctype (ctype.h): character handling
- cstdio (stdio.h): input/output operations
- cstdlib (stdlib.h): general utilities
- cstring (string.h): string manipulation

Here is an example to illustrate the cctype library:

```
1   #include <iostream>
2   #include <cctype>
3   using namespace std;
4
5   int main() {
6      char messyString[] = "t6H0I9s6.iS.999a9.STRING";
7
8      char current = messyString[0];
9      for(int i = 0; current != '\0'; current = messyString[++i]) {
10         if(isalpha(current))
11            cout << (char)(isupper(current) ? tolower(current) : current);
12         else if(ispunct(current))
13            cout << ' ';
14     }
15
16     cout << endl;
17     return 0;
18   }
```

This example uses the `isalpha`, `isupper`, `ispunct`, and `tolower` functions from the cctype library. The `is`- functions check whether a given character is an alphabetic character, an uppercase letter, or a punctuation character, respectively. These functions return a Boolean value of either `true` or `false`. The `tolower` function converts a given character to lowercase.

The for loop beginning at line 9 takes each successive character from `messyString` until it reaches the null character. On each iteration, if the current character is alphabetic and uppercase, it is converted to lowercase and then displayed. If it is already lowercase it is simply displayed. If the character is a punctuation mark, a space is displayed. All other characters are ignored. The resulting output is `this is a string`. For now, ignore the `(char)` on line 11; we will cover that in a later lecture.

Here is an example to illustrate the cstring library:

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

int main() {
    char fragment1[] = "I'm a s";
    char fragment2[] = "tring!";
    char fragment3[20];
    char finalString[20] = "";

    strcpy(fragment3, fragment1);
    strcat(finalString, fragment3);
    strcat(finalString, fragment2);

    cout << finalString;
    return 0;
}
```

This example creates and initializes two strings, `fragment1` and `fragment2`. `fragment3` is declared but not initialized. `finalString` is partially initialized (with just the null character).

`fragment1` is copied into `fragment3` using `strcpy`, in effect initializing `fragment3` to `I'm a s`. `strcat` is then used to concatenate `fragment3` onto `finalString` (the function overwrites the existing null character), thereby giving `finalString` the same contents as `fragment3`. Then `strcat` is used again to concatenate `fragment2` onto `finalString`. `finalString` is displayed, giving `I'm a string!`.

You are encouraged to read the documentation on these and any other libraries of interest to learn what they can do and how to use a particular function properly. (One source is http://www.cplusplus.com/reference/.)

Lecture 5 Notes: Pointers

1 Background

1.1 Variables and Memory

When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there.

When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the `&` and `*` operators:

1. `&x` evaluates to the address of `x` in memory.
2. `*( &x )` takes the address of `x` and *dereferences* it – it retrieves the value at that location in memory. `*( &x )` thus evaluates to the same thing as `x`.

1.2 Motivating Pointers

Memory addresses, or *pointers*, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. Just a taste of what we'll be able to do with pointers:

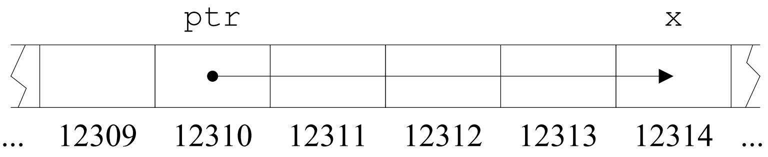
- More flexible pass-by-reference
- Manipulate complex data structures efficiently, even if their data is scattered in different memory locations
- Use polymorphism – calling functions on data without knowing exactly what kind of data it is

2 Pointers and their Behavior

2.1 The Nature of Pointers

Pointers are just variables storing integers – but those integers happen to be memory addresses, usually addresses of other variables. A pointer that stores the address of some variable `x` is said to *point to* `x`. We can access the value of `x` by dereferencing the pointer.

As with arrays, it is often helpful to visualize pointers by using a row of adjacent cells to represent memory locations, as below. Each cell represents 1 block of memory. The dot-arrow notation indicates that `ptr` "points to" `x` – that is, the value stored in `ptr` is 12314, `x`'s memory address.



2.2 Pointer Syntax/Usage

2.2.1 Declaring Pointers

To declare a pointer variable named `ptr` that points to an integer variable named `x`:

```
int *ptr = &x;
```

`int *ptr` declares the pointer to an integer value, which we are initializing to the address of `x`.

We can have pointers to values of any type. The general scheme for declaring pointers is:

```
data_type *pointer_name; // Add "= initial_value" if applicable
```

`pointer_name` is then a variable of type `data_type *` – a "pointer to a `data_type` value."

2.2.2 Using Pointer Values

Once a pointer is declared, we can dereference it with the `*` operator to access its value:

```
cout << *ptr; // Prints the value pointed to by ptr,
              // which in the above example would be x's value
```

We can use deferenced pointers as l-values:

```
*ptr = 5; // Sets the value of x
```

Without the * operator, the identifier x refers to the pointer itself, not the value it points to:

```
cout << ptr; // Outputs the memory address of x in base 16
```

Just like any other data type, we can pass pointers as arguments to functions. The same way we'd say `void func(int x) {...}`, we can say `void func(int *x){...}`. Here is an example of using pointers to square a number in a similar fashion to pass-by-reference:

```
1 void squareByPtr( int *numPtr ) {
2   *numPtr = *numPtr * *numPtr;
3 }
4
5 int main() {
6   int x = 5;
7   squareByPtr(&x);
8   cout << x; // Prints 25
9 }
```

Note the varied uses of the * operator on line 2.

2.2.3 const Pointers

There are two places the const keyword can be placed within a pointer variable declaration. This is because there are two different variables whose values you might want to forbid changing: the pointer itself and the value it points to.

```
const int *ptr;
```

declares a changeable pointer to a constant integer. The integer value cannot be changed through this pointer, but the pointer may be changed to point to a different constant integer.

```
int * const ptr;
```

declares a constant pointer to changeable integer data. The integer value can be changed through this pointer, but the pointer may not be changed to point to a different constant integer.

```
const int * const ptr;
```

forbids changing either the address ptr contains or the value it points to.

2.3 Null, Uninitialized, and Deallocated Pointers

Some pointers do not point to valid data; dereferencing such a pointer is a runtime error. Any pointer set to 0 is called a *null pointer*, and since there is no memory location 0, it is an invalid pointer. One should generally check whether a pointer is null before dereferencing it. Pointers are often set to 0 to signal that they are not currently valid.

Dereferencing pointers to data that has been erased from memory also usually causes runtime errors. Example:

```
int *myFunc() {
  int phantom = 4;
  return &phantom;
}
```

`phantom` is deallocated when `myFunc` exits, so the pointer the function returns is invalid.

As with any other variable, the value of a pointer is undefined until it is initialized, so it may be invalid.

3 References

When we write `void f(int &x) {...}` and call `f(y)`, the reference variable `x` becomes another name – an *alias* – for the value of `y` in memory. We can declare a reference variable locally, as well:

```
int y;
int &x = y; // Makes x a reference to, or alias of, y
```

After these declarations, changing `x` will change `y` and vice versa, because they are two names for the same thing.

References are just pointers that are dereferenced every time they are used. Just like pointers, you can pass them around, return them, set other references to them, etc. The only differences between using pointers and using references are:

- References are sort of pre-dereferenced – you do not dereference them explicitly.
- You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, references must always be initialized when they are declared.
- When writing the value that you want to make a reference to, you do not put an `&` before it to take its address, whereas you do need to do this for pointers.

3.1 The Many Faces of * and &

The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways:

1. When declaring a pointer, * is placed before the variable name to indicate that the variable being declared is a pointer – say, a pointer to an `int` or `char`, not an `int` or `char` value.
2. When using a pointer that has been set to point to some value, * is placed before the pointer name to dereference it – to access or set the value it points to.

A similar distinction exists for &, which can be used either

1. to indicate a reference data type (as in `int &x;`), or
2. to take the address of a variable (as in `int *ptr = &x;`).

4 Pointers and Arrays

The name of an array is actually a pointer to the first element in the array. Writing `myArray[3]` tells the compiler to return the element that is 3 away from the starting element of `myArray`.

This explains why arrays are always passed by reference: passing an array is really passing a pointer.

This also explains why array indices start at 0: the first element of an array is the element that is 0 away from the start of the array.

4.1 Pointer Arithmetic

Pointer arithmetic is a way of using subtraction and addition of pointers to move around between locations in memory, typically between array elements. Adding an integer `n` to a pointer produces a new pointer pointing to `n` positions further down in memory.

4.1.1 Pointer Step Size

Take the following code snippet:

```
long arr[] = {6, 0, 9, 6};
long *ptr = arr;
ptr++;
```

```
4 long *ptr2 = arr + 3;
```

When we add 1 to `ptr` in line 3, we don't just want to move to the next byte in memory, since each array element takes up multiple bytes; we want to move to the next element in the array. The C++ compiler automatically takes care of this, using the appropriate step size for adding to and subtracting from pointers. Thus, line 3 moves `ptr` to point to the second element of the array.

Similarly, we can add/subtract two pointers: `ptr2 - ptr` gives the number of array elements between `ptr2` and `ptr` (2). All addition and subtraction operations on pointers use the appropriate step size.

4.1.2 Array Access Notations

Because of the interchangeability of pointers and array names, *array-subscript notation* (the form `myArray[3]`) can be used with pointers as well as arrays. When used with pointers, it is referred to as *pointer-subscript notation*.

An alternative is *pointer-offset notation*, in which you explicitly add your offset to the pointer and dereference the resulting address. For instance, an alternate and functionally identical way to express `myArray[3]` is `*(myArray + 3)`.

4.2 `char *` Strings

You should now be able to see why the type of a string value is `char *`: a string is actually an array of characters. When you set a `char *` to a string, you are really setting a pointer to point to the first character in the array that holds the string.

You cannot modify string literals; to do so is either a syntax error or a runtime error, depending on how you try to do it. (String literals are loaded into read-only program memory at program startup.) You can, however, modify the contents of an array of characters. Consider the following example:

```
char courseName1[] = {'6', '.', '0', '9', '6', '\0' };
char *courseName2 = "6.096";
```

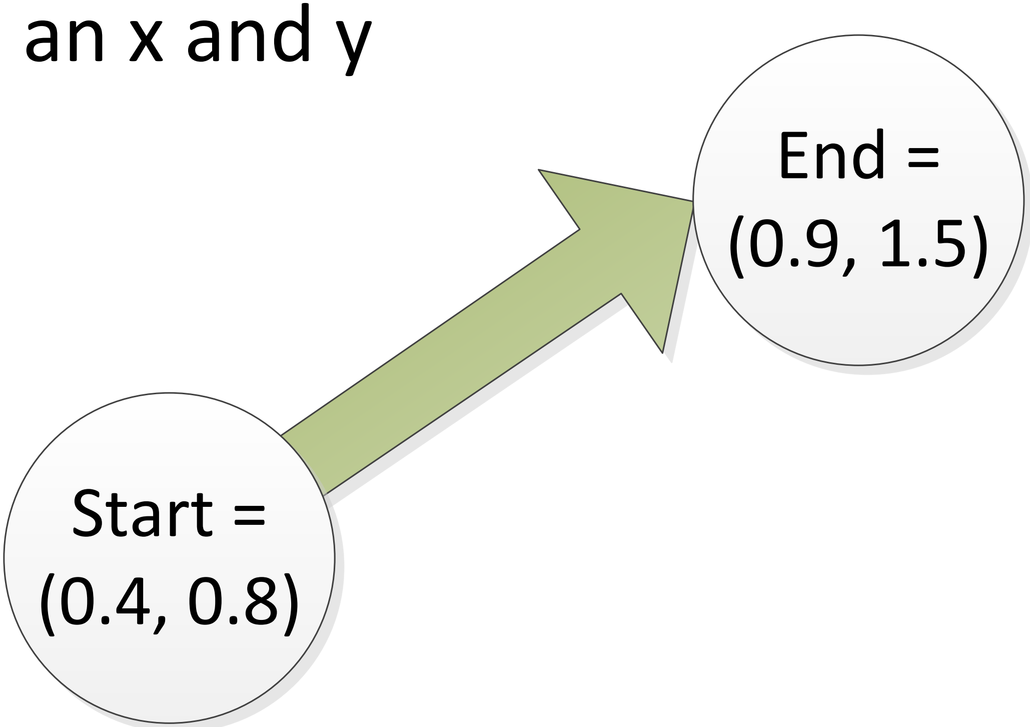
Attempting to modify one of the elements `courseName1` is permitted, but attempting to modify one of the characters in `courseName2` will generate a runtime error, causing the program to crash.

Lecture 6:
User-defined Datatypes

classes and structs

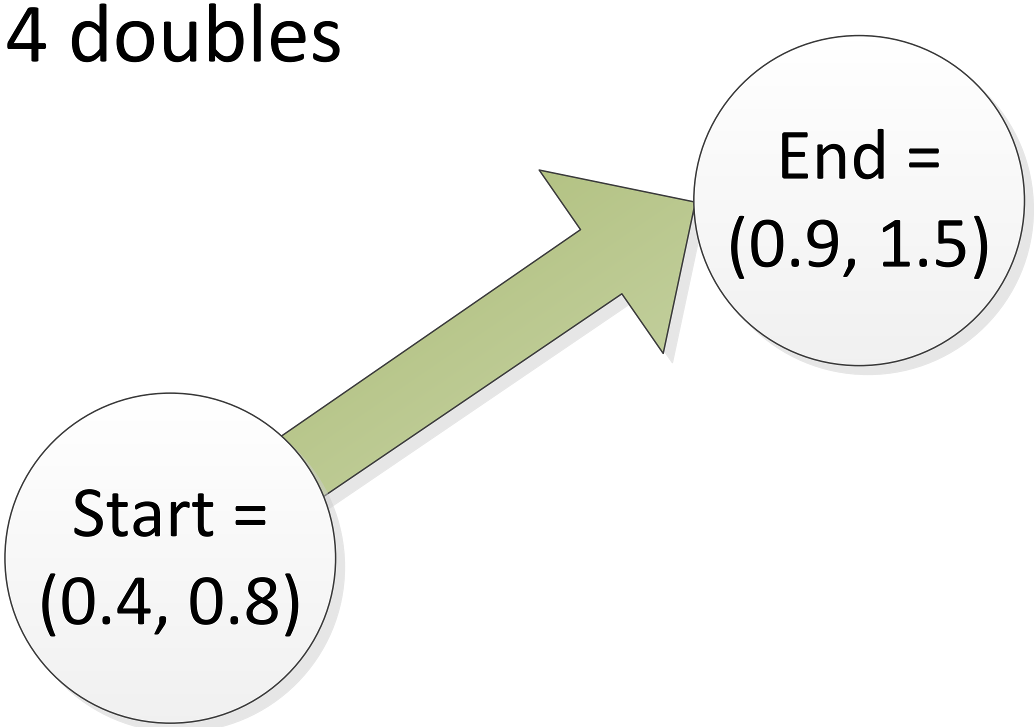
Representing a (Geometric) Vector

- In the context of geometry, a vector consists of 2 points: a start and a finish
- Each point itself has an x and y coordinate

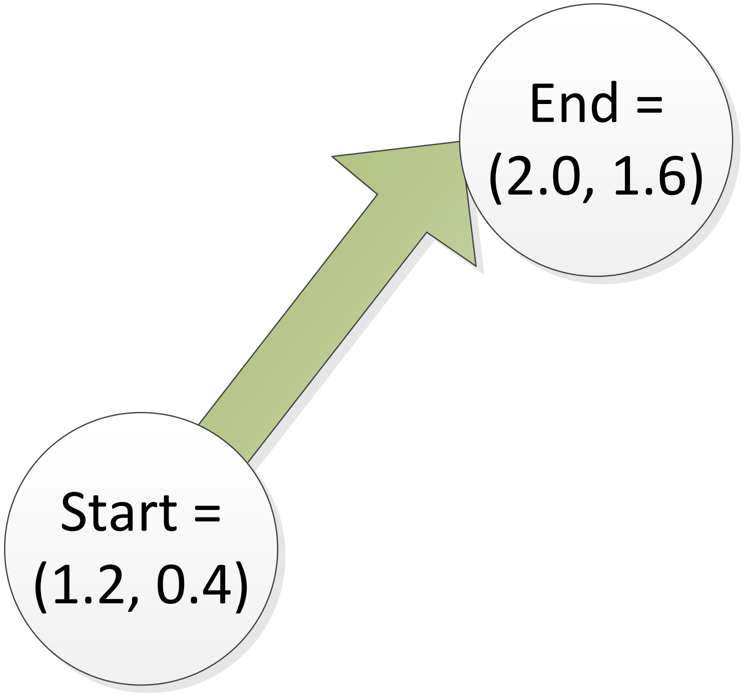


Representing a (Geometric) Vector

- Our representation so far? Use 4 doubles (startx, starty, endx, endy)
- We need to pass all 4 doubles to functions



```
int main() {
  double xStart = 1.2;
  double xEnd = 2.0;
  double yStart = 0.4;
  double yEnd = 1.6;
}
```



End =
(2.0, 1.6)

Start =
(1.2, 0.4)

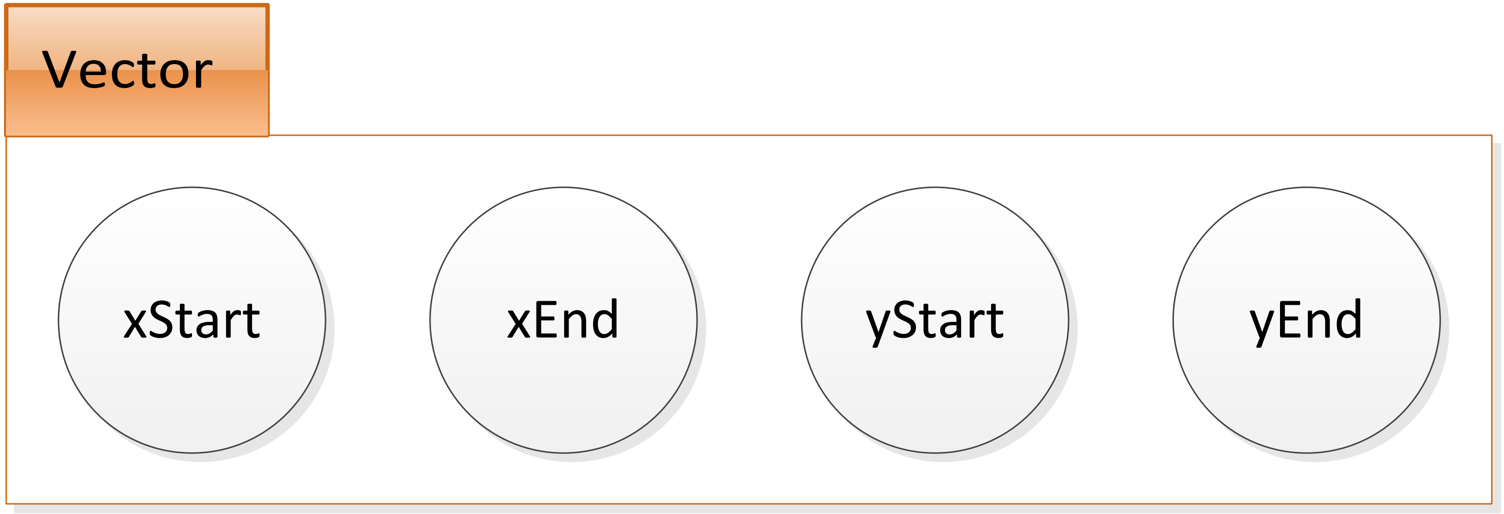
```
void printVector(double x0, double x1, double y0, double y1) {
  cout << "(" << x0 << "," << y0 << ") -> ("
       << x1 << "," << y1 << ")" << endl;
}

int main() {
  double xStart = 1.2;
  double xEnd = 2.0;
  double yStart = 0.4;
  double yEnd = 1.6;
  printVector(xStart, xEnd, yStart, yEnd);
  // (1.2,2.0) -> (0.4,1.6)
}
```

```
void offsetVector(double &x0, double &x1, double &y0, double &y1,
                  double offsetX, double offsetY) {
  x0 += offsetX;
  x1 += offsetX;
  y0 += offsetY;
  y1 += offsetY;
}

void printVector(double x0, double x1, double y0, double y1) {
  cout << "(" << x0 << "," << y0 << ") -> ("
       << x1 << "," << y1 << ")" << endl;
}

int main() {
  double xStart = 1.2;
  double xEnd = 2.0;
  double yStart = 0.4;
  double yEnd = 1.6;
  offsetVector(xStart, xEnd, yStart, yEnd, 1.0, 1.5);
  printVector(xStart, xEnd, yStart, yEnd);
  // (2.2,1.9) -> (3.8,4.3)
}
```

Many variables being passed to functions

class

- A user-defined datatype which groups together related pieces of information



class definition syntax

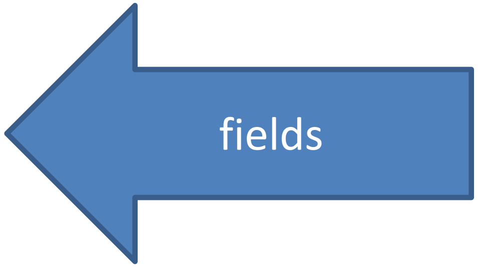
name

```
class Vector {
public:
  double xStart;
  double xEnd;
  double yStart;
  double yEnd;
};
```

- This indicates that the new datatype we’re defining is called Vector

class definition syntax

```
class Vector {
public:
  double xStart;
  double xEnd;
  double yStart;
  double yEnd;
};
```

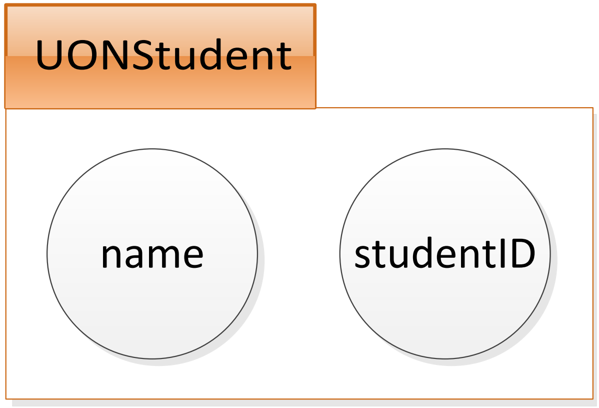


fields

- **Fields** indicate what related pieces of information our datatype consists of
  - Another word for field is **members**

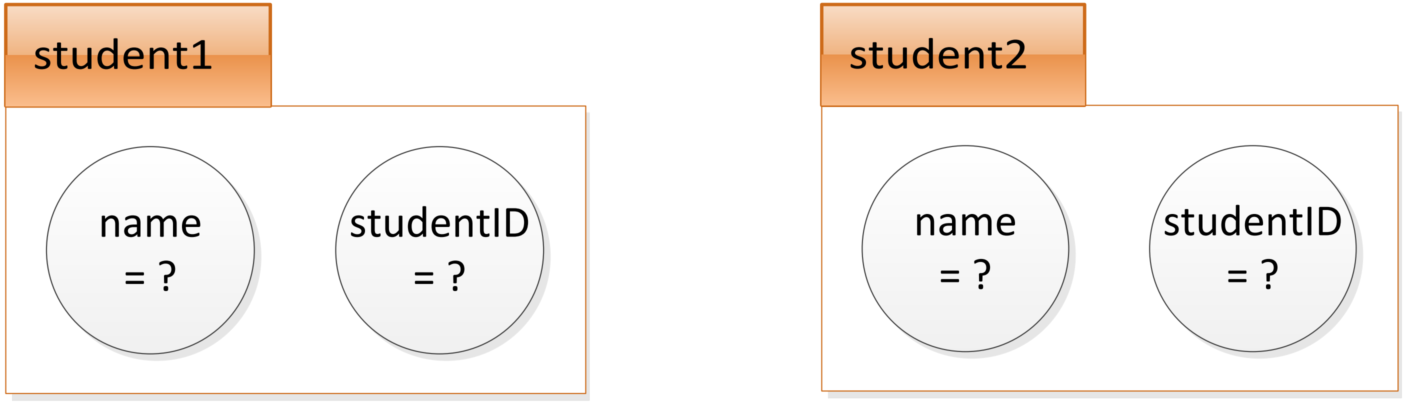
Fields can have different types

```
class UONStudent {
public:
  char *name;
  int studentID; };
```



Instances

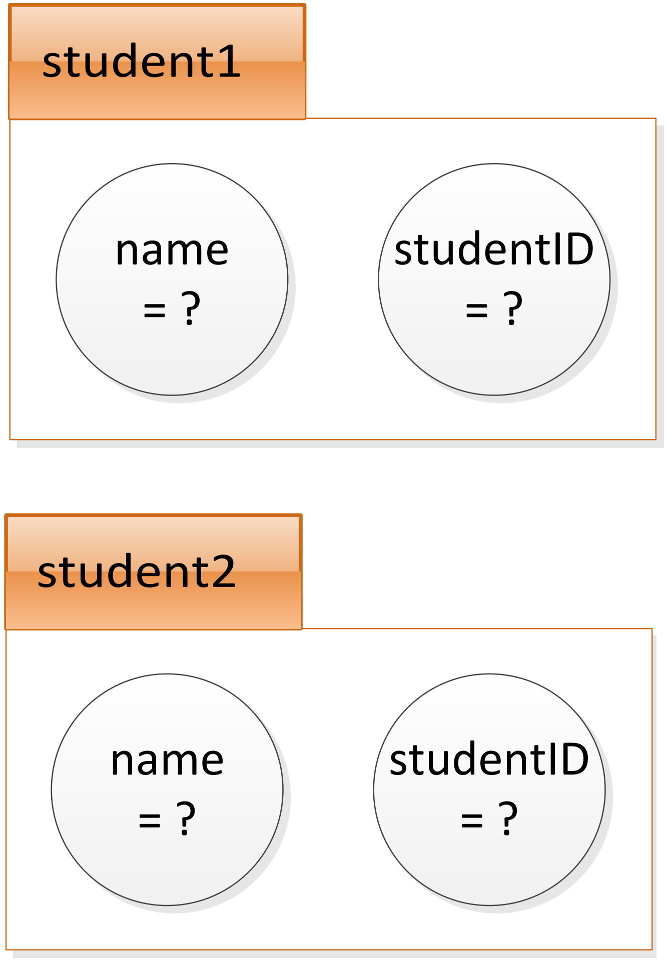
- An instance is an occurrence of a class. Different instances can have their own set of values in their fields.
- If you wanted to represent 2 different students (who can have different names and IDs), you would use 2 instances of UONStudent



Declaring an Instance

- Defines 2 instances of UONStudent: one called student1, the other called student2

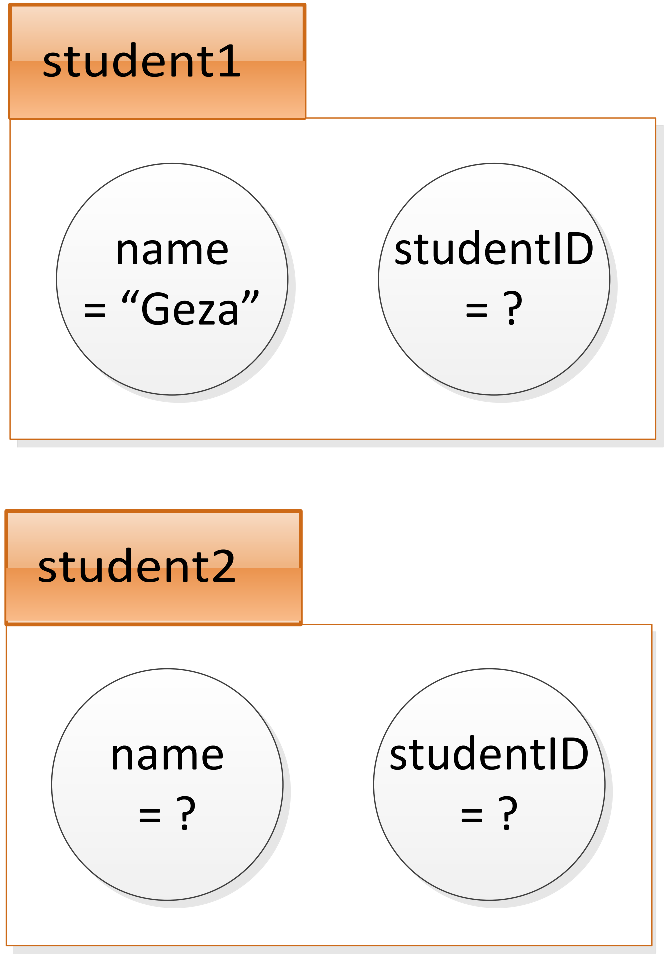
```
class UONStudent {
public:
   char *name;
   int studentID;
};

int main() {
   UONStudent student1;
   UONStudent student2;
 }
```



Accessing Fields

- To access fields of instances, use variable.fieldName

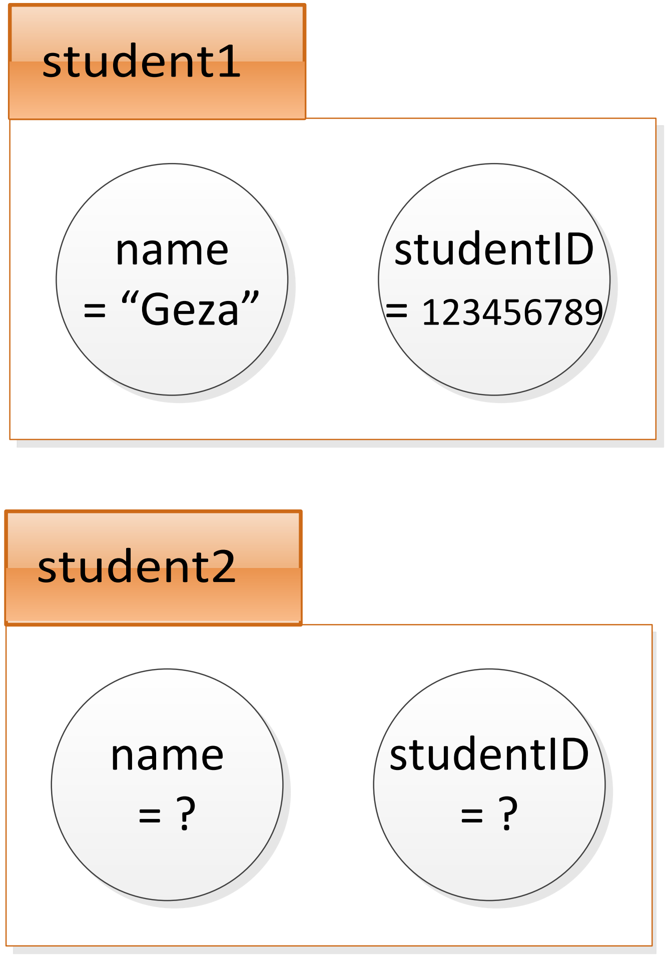
```
class UONStudent {
public:
  char *name;
  int studentID;
};

int main() {
  UONStudent student1;
  UONStudent student2;
  student1.name = "Geza";
}
```



Accessing Fields

- To access fields of instances, use variable.fieldName

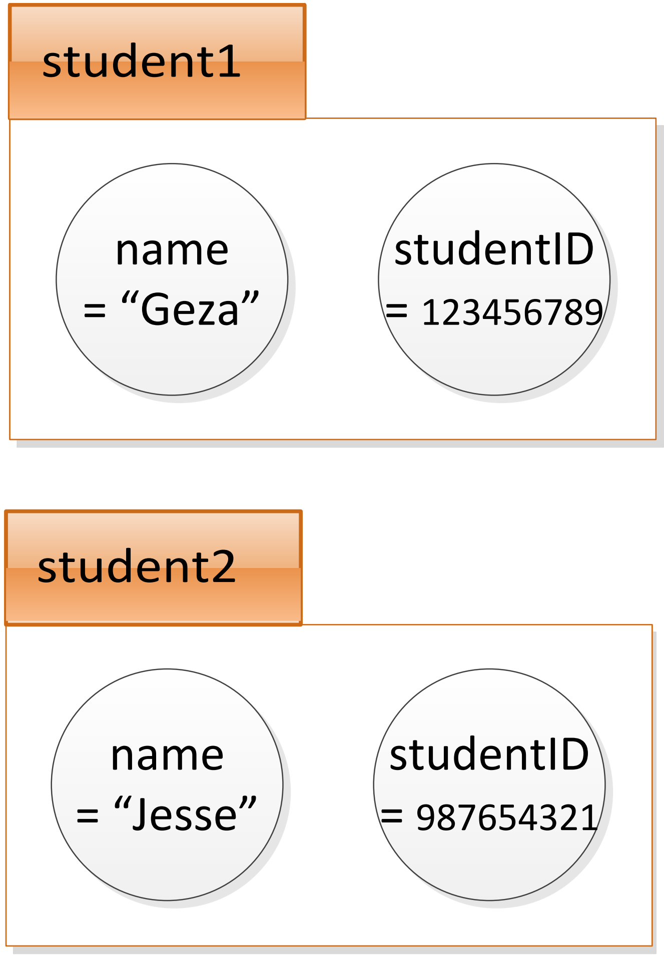
```
class UONStudent {
public:
  char *name;
  int studentID;
};

int main() {
  UONStudent student1;
  UONStudent student2;
  student1.name = "Geza";
  student1.studentID = 123456789;
}
```



Accessing Fields

- To access fields of instances, use variable.fieldName

```
class UONStudent {
public:
   char *name;
   int studentID;
};

int main() {
   UONStudent student1;
   UONStudent student2;
   student1.name = "Geza";
   student1.studentID = 123456789;
   student2.name = "Jesse";
   student2.studentID = 987654321;
}
```

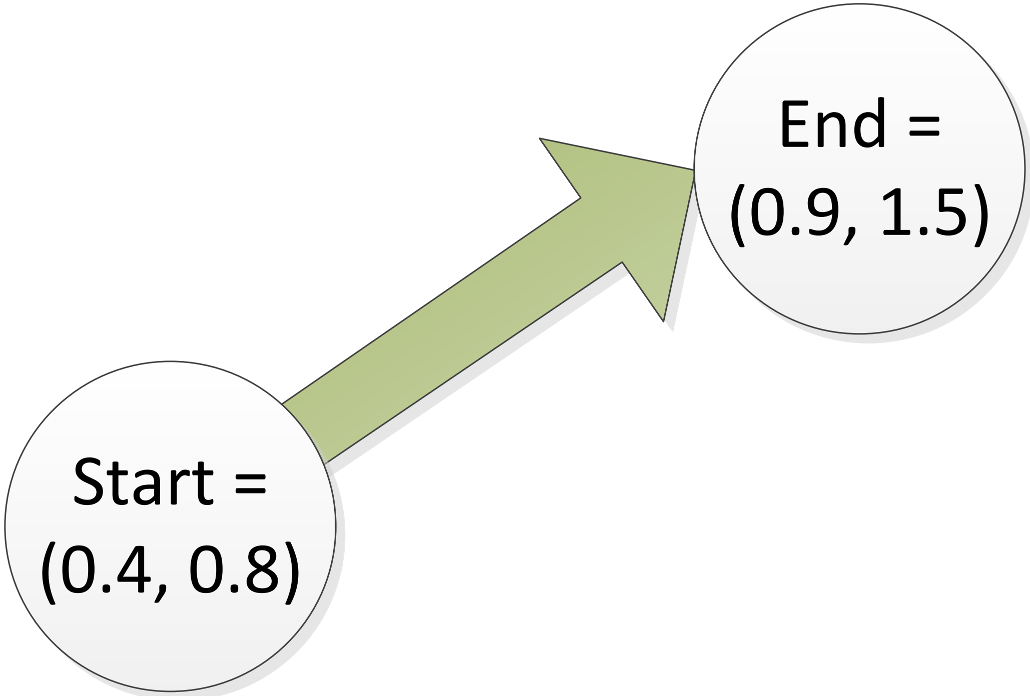


Accessing Fields

- To access fields of instances, use variable.fieldName

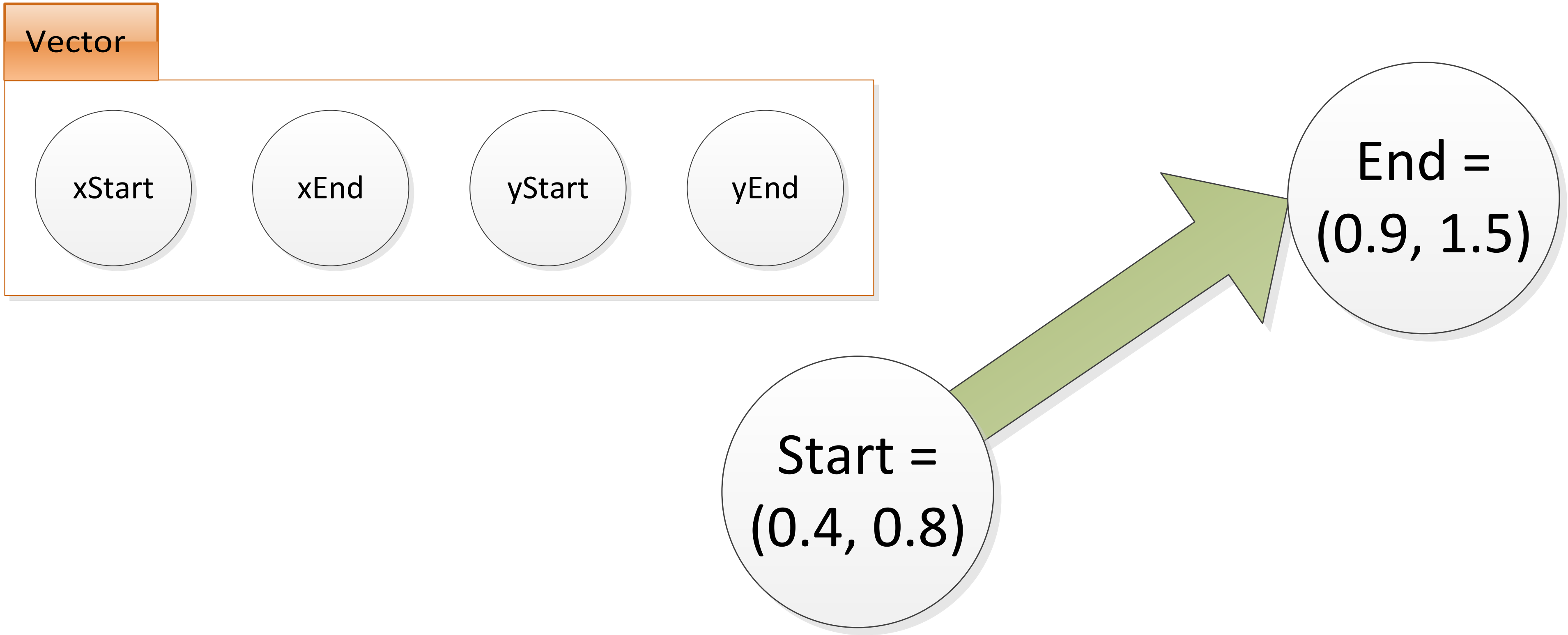
```
class UONStudent {
public:
  char *name;
  int studentID;
};

int main() {
  UONStudent student1;
  UONStudent student2;
  student1.name = "Geza";
  student1.studentID = 123456789;
  student2.name = "Jesse";
  student2.studentID = 987654321;
  cout << "student1 name is" << student1.name << endl;
  cout << "student1 id is" << student1.studentID << endl;
  cout << "student2 name is" << student2.name << endl;
  cout << "student2 id is" << student2.studentID << endl;
```

- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish



```
class Vector {
public:
  double xStart;
  double xEnd;
  double yStart;
  double yEnd;
};
```

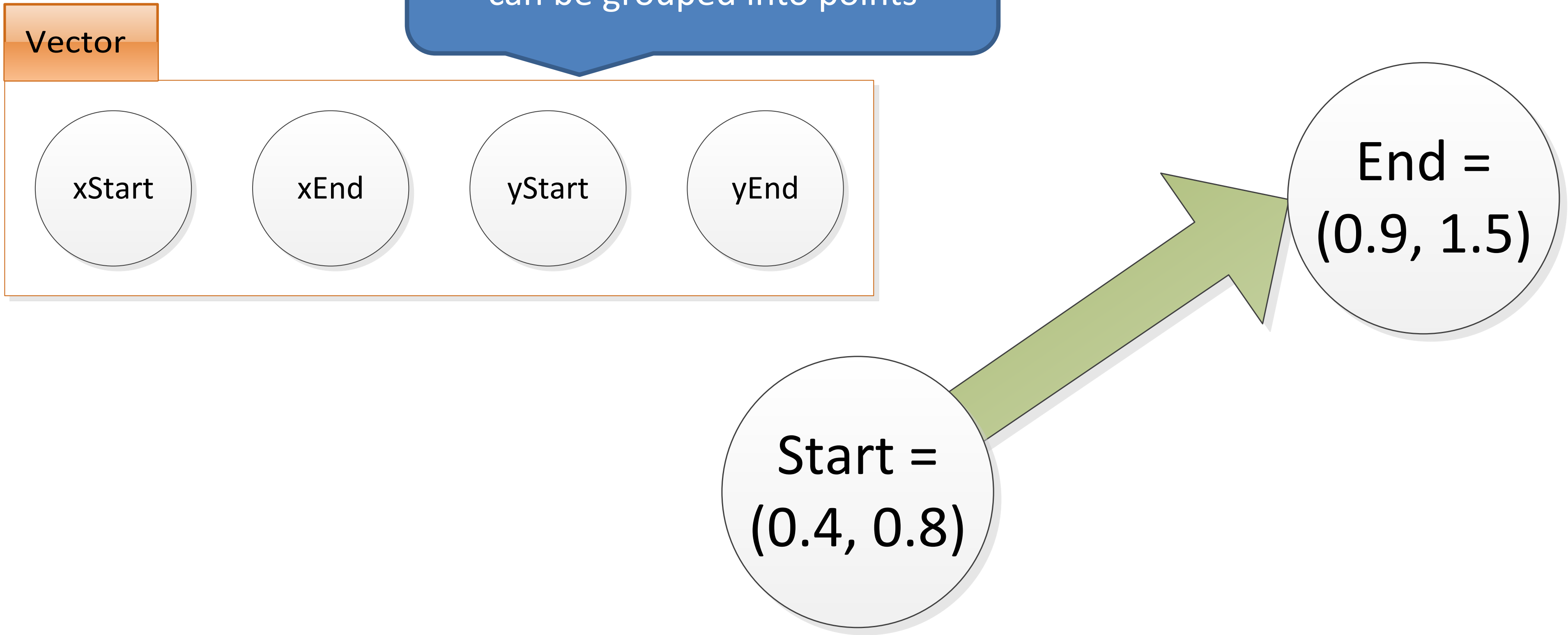
- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish



- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish

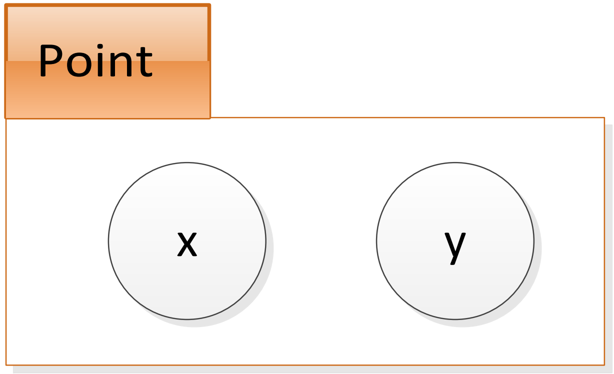
```
class Vector {
public:
  double xStart;
  double xEnd;
  double yStart;
  double yEnd;
};
```

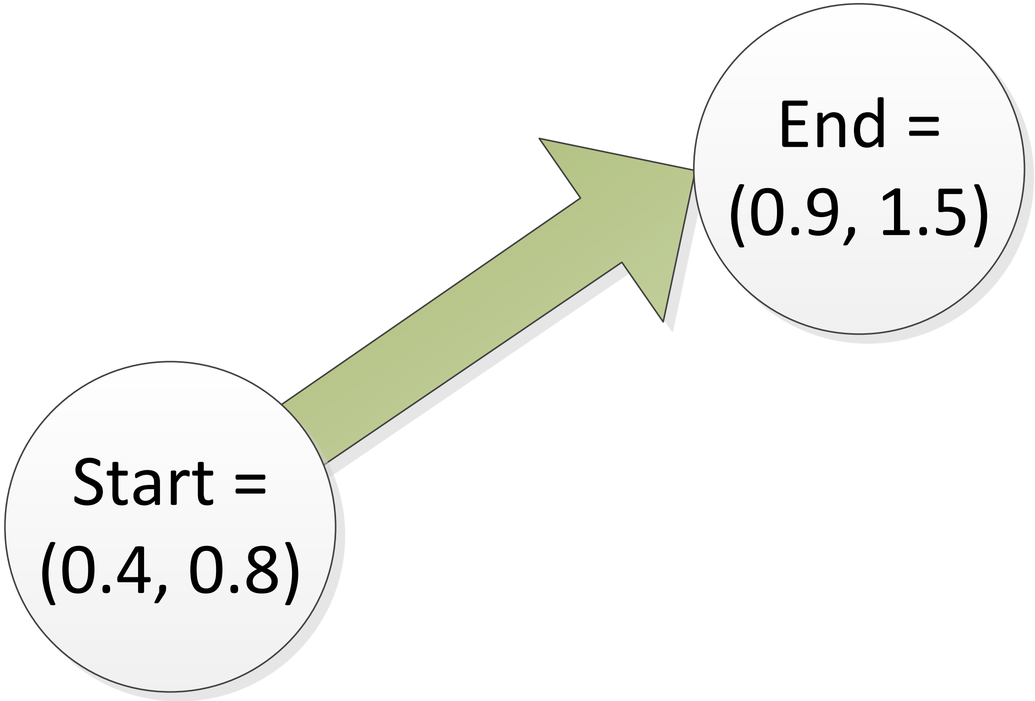
Doesn't show that coordinates can be grouped into points



- **A point consists of an x and y coordinate**
- A vector consists of 2 points: a start and a finish

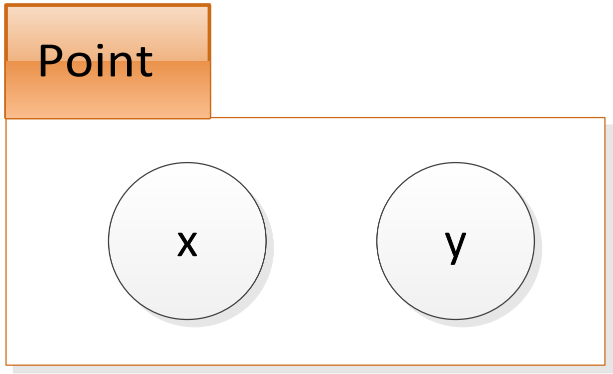
```
class Point {
public:
  double x;
  double y;
};
```

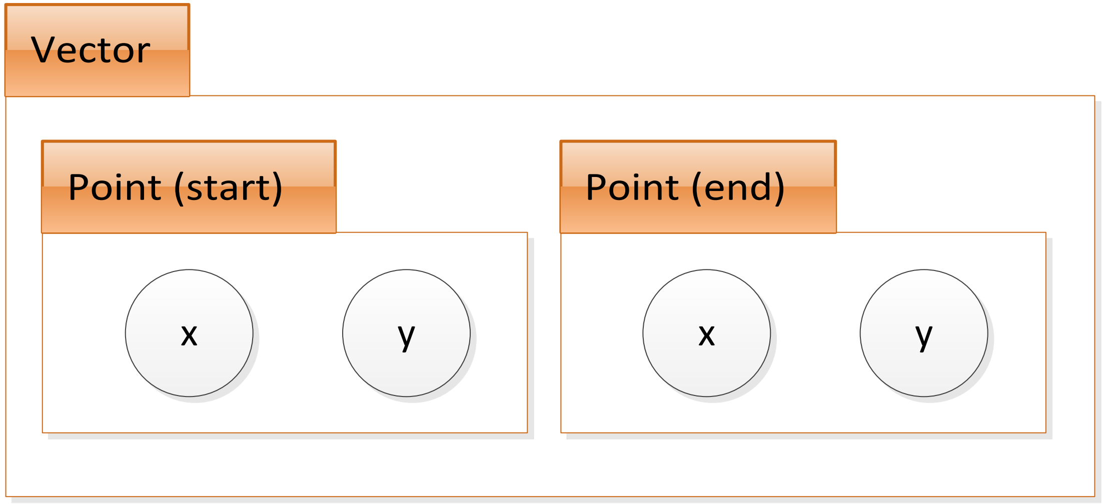




```
class Point {
public:
  double x;
  double y;
};
```

- A point consists of an x and y coordinate
- **A vector consists of 2 points: a start and a finish**

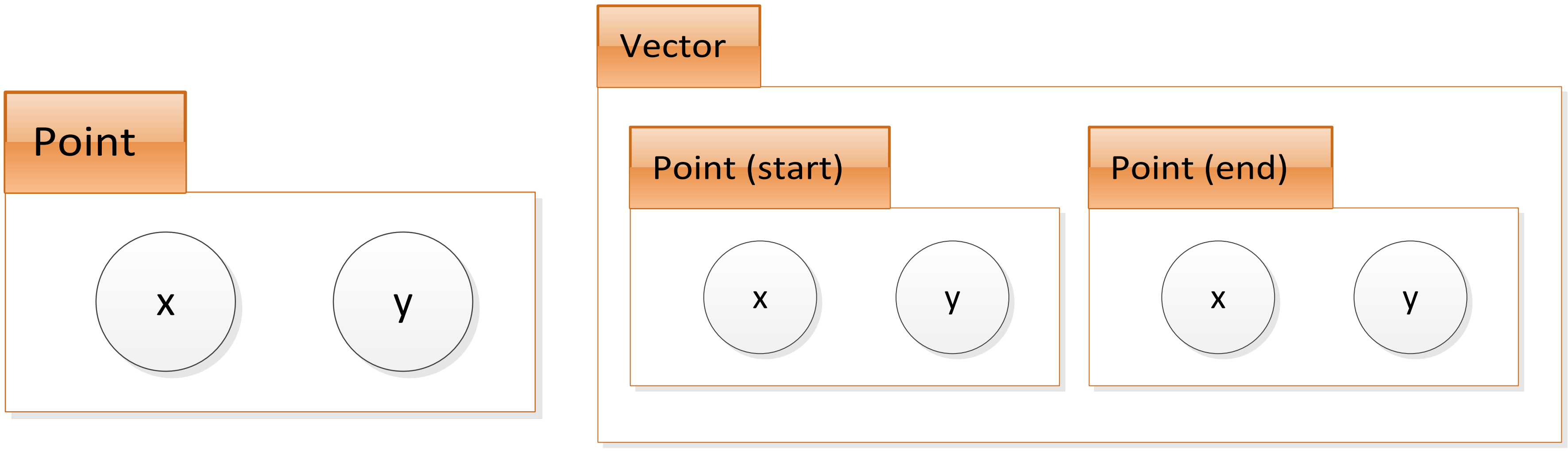




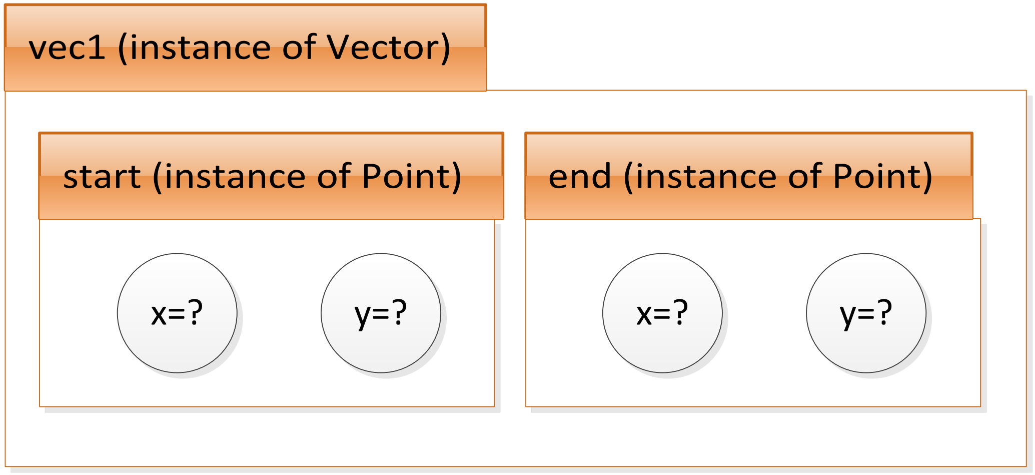
- A point consists of an x and y coordinate
- **A vector consists of 2 points: a start and a finish**

```
class Point {
public:
  double x;
  double y;
};

class Vector {
public:
  Point start;
  Point end;
};
```

Fields can be classes



```
class Point {
public:
  double x, y;
};

class Vector {
public:
  Point start, end;
};

int main() {
  Vector vec1;
}
```

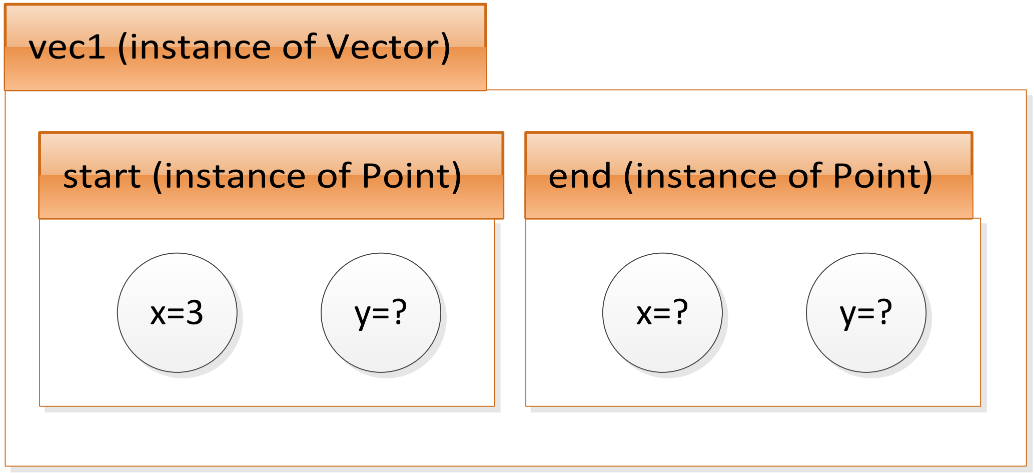


vec1 (instance of Vector)

start (instance of Point)

end (instance of Point)

```
class Point {
public:
  double x, y;
};

class Vector {
public:
  Point start, end;
};

int main() {
  Vector vec1;
  vec1.start.x = 3.0;
}
```



vec1 (instance of Vector)

start (instance of Point)

x=3

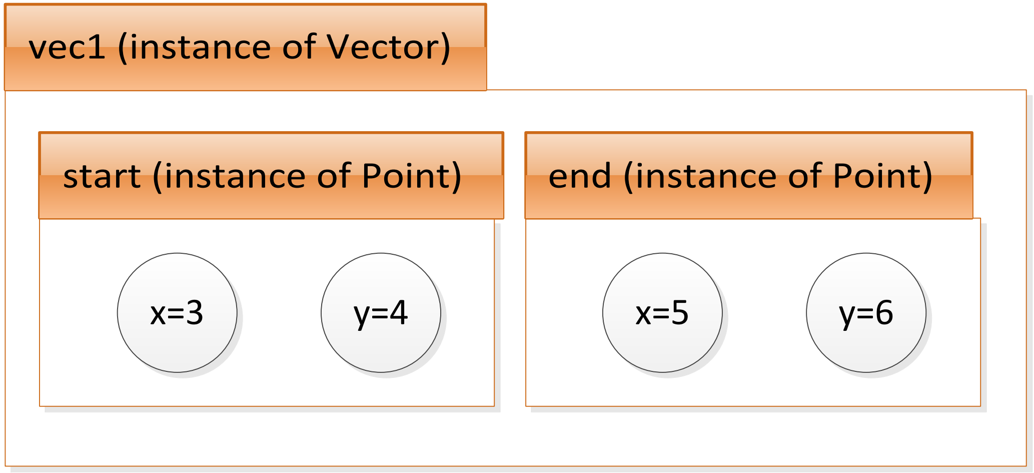
y=?

end (instance of Point)

x=?

y=?

```
class Point {
public:
  double x, y;
};

class Vector {
public:
  Point start, end;
};

int main() {
  Vector vec1;
  vec1.start.x = 3.0;
  vec1.start.y = 4.0;
  vec1.end.x = 5.0;
  vec1.end.y = 6.0;
}
```



vec1 (instance of Vector)

start (instance of Point)

x=3

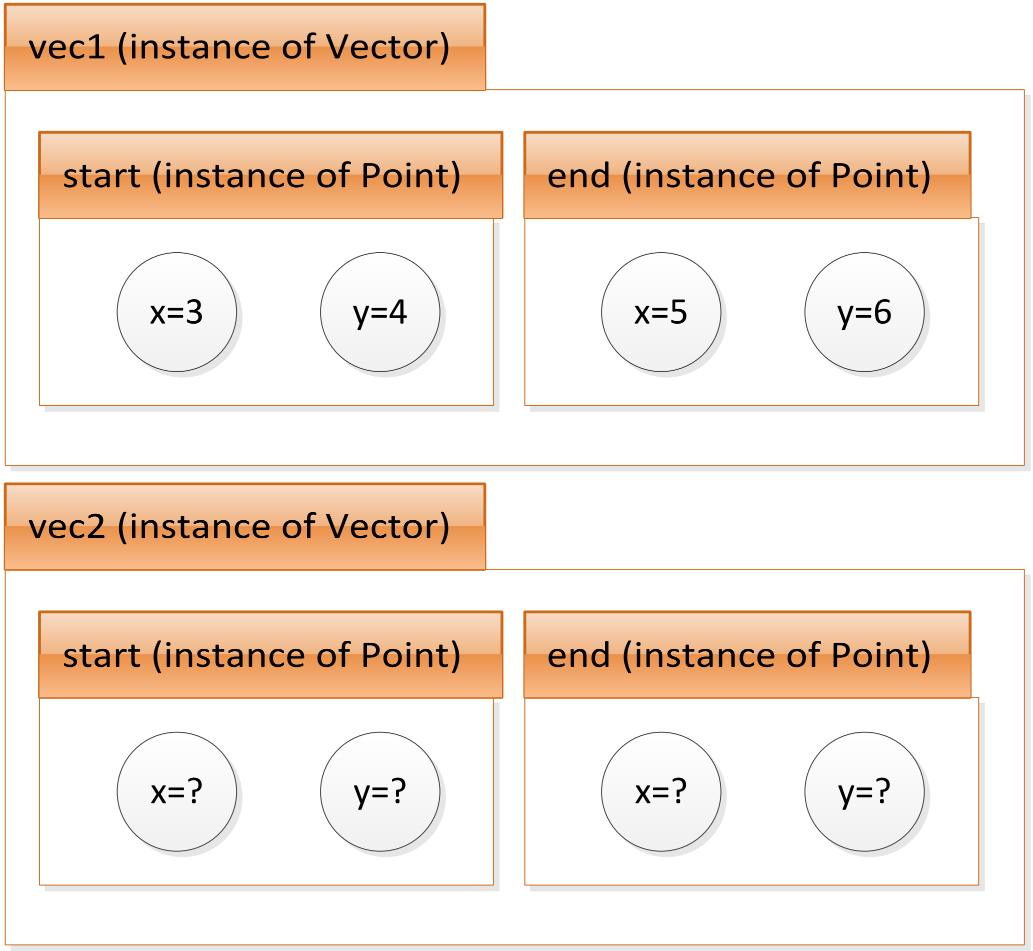
y=4

end (instance of Point)

x=5

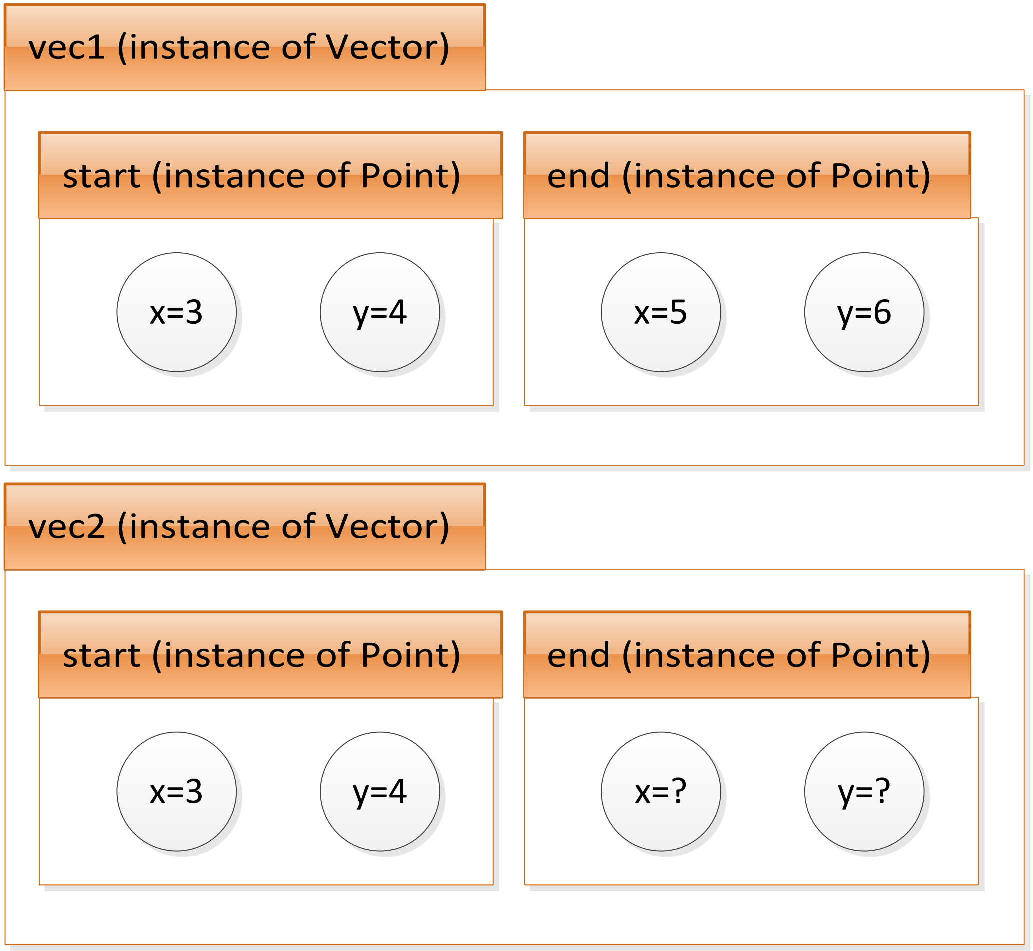
y=6

```
class Point {
public:
  double x, y;
};

class Vector {
public:
  Point start, end;
};

int main() {
  Vector vec1;
  vec1.start.x = 3.0;
  vec1.start.y = 4.0;
  vec1.end.x = 5.0;
  vec1.end.y = 6.0;
  Vector vec2;
}
```



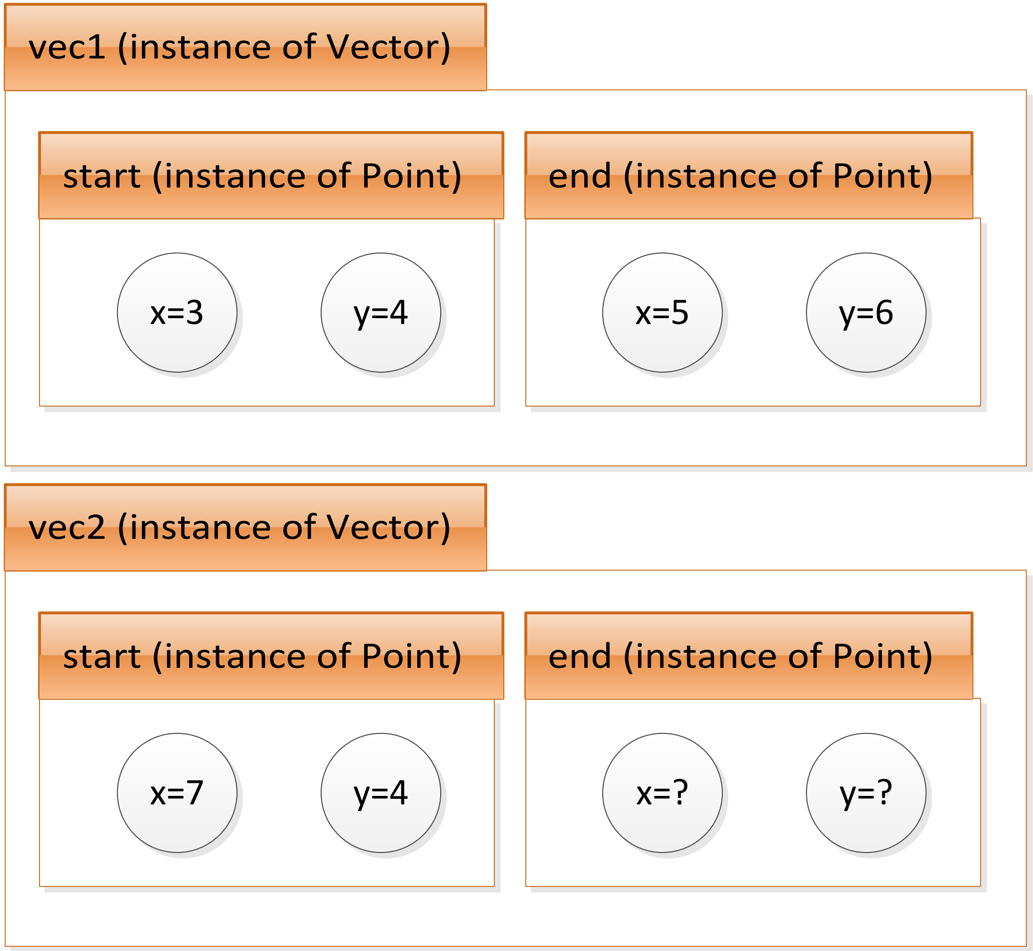
vec1 (instance of Vector)
start (instance of Point)
x=3
y=4
end (instance of Point)
x=5
y=6
vec2 (instance of Vector)
start (instance of Point)
x=?
y=?
end (instance of Point)
x=?
y=?

```
class Point {
public:
  double x, y;
};

class Vector {
public:
  Point start, end;
};

int main() {
  Vector vec1;
  vec1.start.x = 3.0;
  vec1.start.y = 4.0;
  vec1.end.x = 5.0;
  vec1.end.y = 6.0;
  Vector vec2;
  vec2.start = vec1.start;
}
```


vec1 (instance of Vector)
start (instance of Point)
x=3
y=4
end (instance of Point)
x=5
y=6
vec2 (instance of Vector)
start (instance of Point)
x=3
y=4
end (instance of Point)
x=?
y=?

- Assigning one instance to another copies all fields

```
class Point {
public:
  double x, y;
};

class Vector {
public:
  Point start, end;
};

int main() {
  Vector vec1;
  vec1.start.x = 3.0;
  vec1.start.y = 4.0;
  vec1.end.x = 5.0;
  vec1.end.y = 6.0;
  Vector vec2;
  vec2.start = vec1.start;
  vec2.start.x = 7.0;
}
```


vec1 (instance of Vector)
start (instance of Point)
x=3
y=4
end (instance of Point)
x=5
y=6
vec2 (instance of Vector)
start (instance of Point)
x=7
y=4
end (instance of Point)
x=?
y=?

- Assigning one instance to another copies all fields

Passing classes to functions

- Passing by value passes a copy of the class instance to the function; changes aren't preserved

```
class Point { public: double x, y; };

void offsetPoint(Point p, double x, double y) { // does nothing
  p.x += x;
  p.y += y;
}

int main() {
  Point p;
  p.x = 3.0;
  p.y = 4.0;
  offsetPoint(p, 1.0, 2.0); // does nothing
  cout << "(" << p.x << "," << p.y << ")"; // (3.0,4.0)
}
```

Passing classes to functions

- When a class instance is passed by reference, changes are reflected in the original

```
class Point { public: double x, y; };

void offsetPoint(Point &p, double x, double y) { // works
  p.x += x;
  p.y += y;
}
```

Passed by reference

```
int main() {
  Point p;
  p.x = 3.0;
  p.y = 4.0;
  offsetPoint(p, 1.0, 2.0); // works
  cout << "(" << p.x << "," << p.y << ")"; // (4.0,6.0)
}
```

```
class Point {
  public: double x, y;
};
```

Point class, with fields x and y

```
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

Fields can be classes

```
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

```
int main() {
  Vector vec;
}
```

vec is an instance of Vector

```cpp
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

```cpp
int main() {
  Vector vec;
  vec.start.x = 1.2;
}
```

Accessing fields

```
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

```
int main() {
  Vector vec;
  vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
}
```

```
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

```
void printVector(Vector v) {
  cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
"," << v.end.y << ")" << endl;
}

int main() {
  Vector vec;
  vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
  printVector(vec); // (1.2,0.4) -> (2.0,1.6)
}
```

classes can be passed to functions

```
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

Can pass to value if you don't need to modify the class

```
void printVector(Vector v) {
  cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
"," << v.end.y << ")" << endl;
}

int main() {
  Vector vec;
  vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
  printVector(vec); // (1.2,0.4) -> (2.0,1.6)
}
```

```
class Point {
  public: double x, y;
};
class Vector {
  public: Point start, end;
};
```

Pass classes by reference if they need to be modified

```
void offsetVector(Vector &v, double offsetX, double offsetY) {
  v.start.x += offsetX;
  v.end.x += offsetX;
  v.start.y += offsetY;
  v.end.y += offsetY;
}
void printVector(Vector v) {
  cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
"," << v.end.y << ")" << endl;
}

int main() {
  Vector vec;
  vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
  offsetVector(vec,  1.0, 1.5);
  printVector(vec); // (2.2,1.9) -> (3.8,4.3)
}
```

- Observe how some functions are closely associated with a particular class

```
void offsetVector(Vector &v, double offsetX, double offsetY);
void printVector(Vector v);
```

```
int main() {
  Vector vec;
  vec.start.x = 1.2; vec.end.x = 2.0;
  vec.start.y = 0.4; vec.end.y = 1.6;
  offsetVector(vec,  1.0, 1.5);
  printVector(vec);
}
```

- Observe how some functions are closely associated with a particular class
- **Methods**: functions which are part of a class

```
Vector vec;
vec.start.x = 1.2; vec.end.x = 2.0;
vec.start.y = 0.4; vec.end.y = 1.6;
vec.print();
```

Method name

- Observe how some functions are closely associated with a particular class
- **Methods**: functions which are part of a class
  - Implicitly pass the current instance

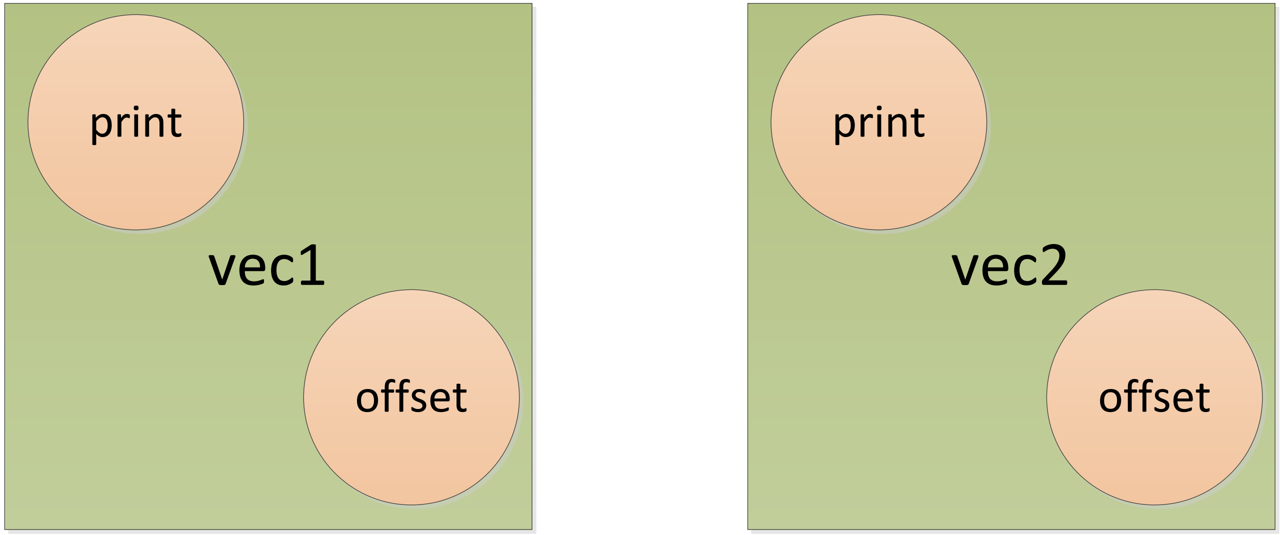
```
Vector vec;
vec.start.x = 1.2; vec.end.x = 2.0;
vec.start.y = 0.4; vec.end.y = 1.6;
vec.print();
```

Object instance

- Observe how some functions are closely associated with a particular class
- **Methods**: functions which are part of a class
  - Implicitly pass the current instance

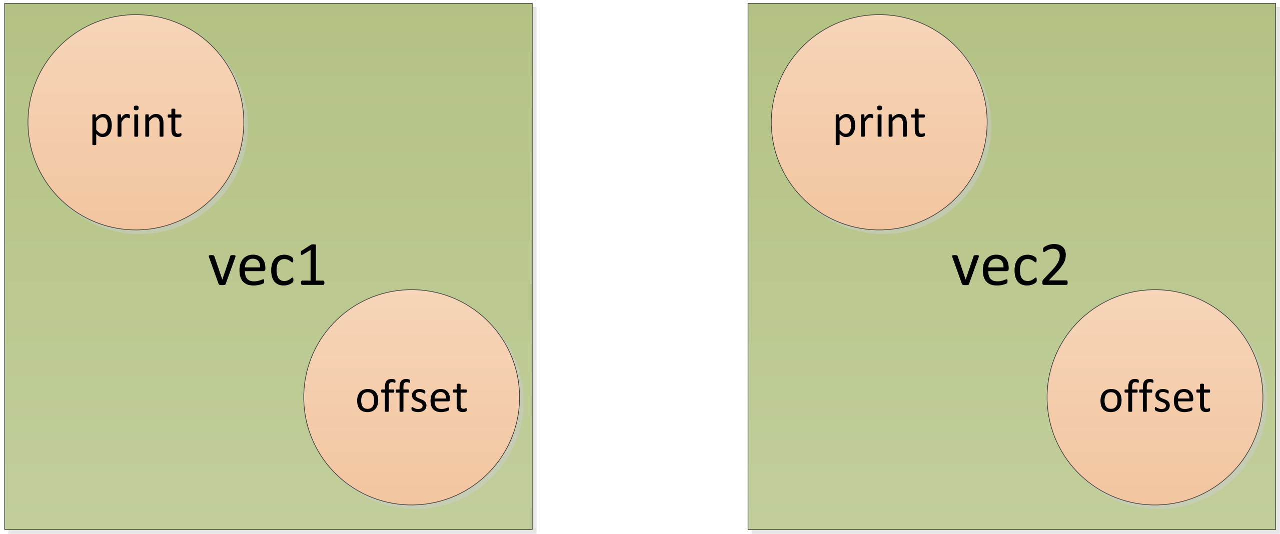
```
Vector vec;
vec.start.x = 1.2; vec.end.x = 2.0;
vec.start.y = 0.4; vec.end.y = 1.6;
vec.print();
vec.offset(1.0, 1.5);
```

Arguments can be passed to methods



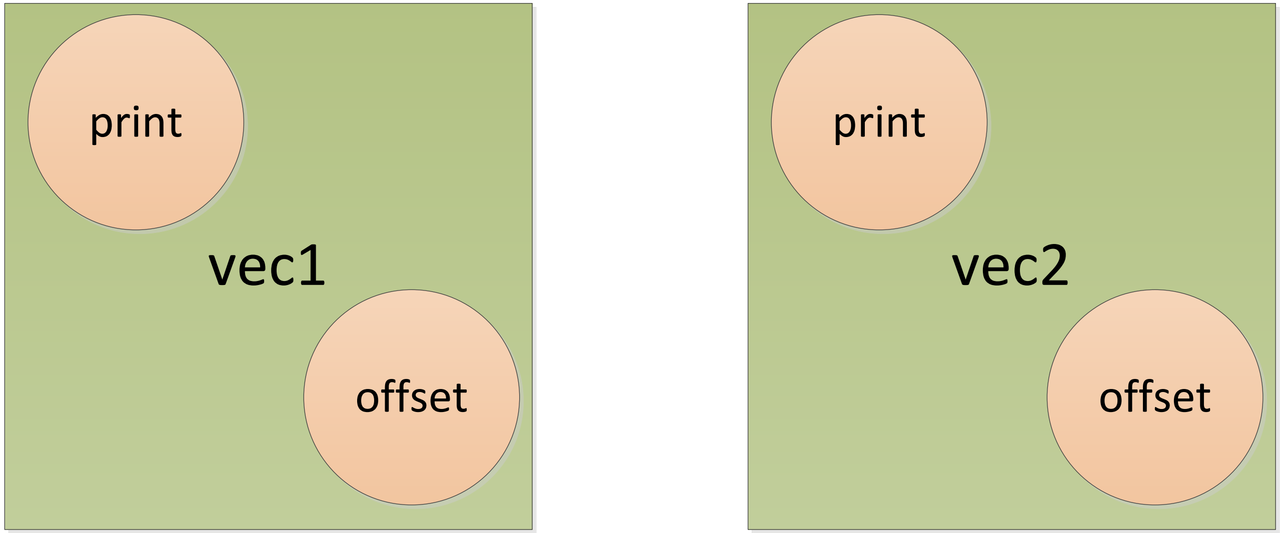
```
Vector vec1;
Vector vec2;
// initialize vec1 and vec2
vec1.print();
```

- Analogy: Methods are “buttons” on each box (instance), which do things when pressed



```
Vector vec1;
Vector vec2;
// initialize vec1 and vec2
vec1.print();
```

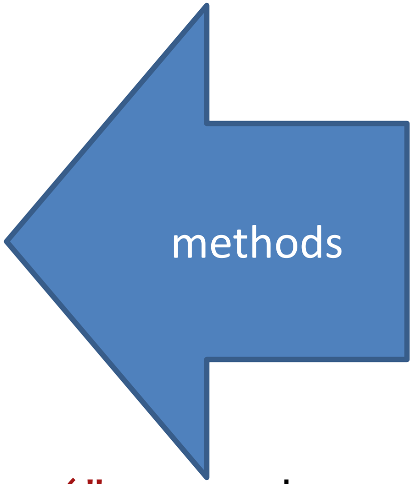
Which box's button was pressed?



```
Vector vec1;
Vector vec2;
// initialize vec1 and vec2
vec1.print();
```

Which button was pressed?

```
class Vector {
public:
  Point start;
  Point end;

  void offset(double offsetX, double offsetY) {
    start.x += offsetX;
    end.x += offsetX;
    start.y += offsetY;
    end.y += offsetY;
  }
  void print() {
    cout << "(" << start.x << "," << start.y << ") -> (" << end.x <<
"," << end.y << ")" << endl;
  }
};
```



methods

```
class Vector {
public:
  Point start;
  Point end;

  void offset(double offsetX, double offsetY) {
    start.x += offsetX;
    end.x += offsetX;
    start.y += offsetY;
    end.y += offsetY;
  }
  void print() {
    cout << "(" << start.x << "," << start.y << ") -> (" << end.x << "," << end.y << ")" << endl;
  }
};
```

Fields can be accessed in a method

```
class Vector {
public:
  Point start, end;

  void offset(double offsetX, double offsetY) {
    start.offset(offsetX, offsetY);
    end.offset(offsetX, offsetY);
  }
  void print() {
    start.print();
    cout << " -> ";
    end.print();
    cout << endl;
  }
};
```

methods of fields can be called

```
class Point {
public:
  double x, y;
  void offset(double offsetX, double offsetY) {
    x += offsetX; y += offsetY;
  }
  void print() {
    cout << "(" << x << "," << y << ")";
  }
};
```

Implementing Methods Separately

- Recall that function prototypes allowed us to declare that functions will be implemented later
- This can be done analogously for class methods

```
// vector.h - header file
class Point {
public:
  double x, y;
  void offset(double offsetX, double offsetY);
  void print();
};

class Vector {
public:
  Point start, end;
  void offset(double offsetX, double offsetY);
  void print();
};
```

```
#include "vector.h"
// vector.cpp - method implementation
void Point::offset(double offsetX, double offsetY) {
  x += offsetX; y += offsetY;
}
void Point::print() {
  cout << "(" << x << "," << y << ")";
}
void Vector::offset(double offsetX, double offsetY) {
  start.offset(offsetX, offsetY);
  end.offset(offsetX, offsetY);
}
void Vector::print() {
  start.print();
  cout << " -> ";
  end.print();
  cout << endl;
}
```

:: indicates which class’ method is being implemented

- Manually initializing your fields can get tedious
- Can we initialize them when we create an instance?

```
Vector vec;
vec.start.x = 0.0;
vec.start.y = 0.0;
vec.end.x = 0.0;
vec.end.y = 0.0;
```

```
Point p;
p.x = 0.0;
p.y = 0.0;
```

Constructors

- Method that is called when an instance is created

```
class Point {
public:
  double x, y;
  Point() {
    x = 0.0; y = 0.0; cout << "Point instance created" << endl;
  }
};

int main() {
  Point p; // Point instance created
  // p.x is 0.0, p.y is 0.0
}
```

Constructors

- Can accept parameters

```cpp
class Point {
public:
  double x, y;
  Point(double nx, double ny) {
    x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
};

int main() {
  Point p(2.0, 3.0); // 2-parameter constructor
  // p.x is 2.0, p.y is 3.0
}
```

Constructors

- Can have multiple constructors

```
class Point {
public:
  double x, y;
  Point() {
    x = 0.0; y = 0.0; cout << "default constructor" << endl;
  }
  Point(double nx, double ny) {
    x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
};

int main() {
  Point p; // default constructor
  // p.x is 0.0, p.y is 0.0)
  Point q(2.0, 3.0); // 2-parameter constructor
  // q.x is 2.0, q.y is 3.0)
}
```

- Recall that assigning one class instance to another copies all fields (default **copy constructor**)

```
class Point {
public:
  double x, y;
  Point() {
    x = 0.0; y = 0.0; cout << "default constructor" << endl;
  }
  Point(double nx, double ny) {
    x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
};

int main() {
  Point q(1.0, 2.0); // 2-parameter constructor
  Point r = q;
  // r.x is 1.0, r.y is 2.0)
}
```

Invoking the copy constructor

- You can define your own copy constructor

```
class Point {
public:
  double x, y;
  Point(double nx, double ny) {
    x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
  Point(Point &o) {
    x = o.x; y = o.y; cout << "custom copy constructor" << endl;
  }
};

int main() {
  Point q(1.0, 2.0); // 2-parameter constructor
  Point r = q; // custom copy constructor
  // r.x is 1, r.y is 2
}
```

- Why make a copy constructor? Assigning all fields (default copy constructor) may not be what you want

```
class UONStudent {
public:
  int studentID;
  char *name;
  UONStudent() {
    studentID = 0;
    name = "";
  }
};
```

```
int main() {
  UONStudent student1;
  student1.studentID = 98;
  char n[] = "foo";
  student1.name = n;
  UONStudent student2 = student1;
  student2.name[0] = 'b';
  cout << student1.name; // boo }
```

By changing student 2’s name, we changed student 1’s name as well

- Why make a copy constructor? Assigning all fields (default copy constructor) may not be what you want

```
class UONStudent {
public:
  int studentID;
  char *name;
  UONStudent() {
    studentID = 0;
    name = "";
  }
  UONStudent(UONStudent &o) {
    studentID = o.studentID;
    name = strdup(o.name);
  }
};
```

```
int main() {
  UONStudent student1;
  student1.studentID = 98;
  char n[] = "foo";
  student1.name = n;
  UONStudent student2 = student1;
  student2.name[0] = 'b';
  cout << student1.name; // foo }
```

Changing student 2's name doesn't effect student 1's name

Access Modifiers

- Define where your fields/methods can be accessed from

Access Modifier

```
class Point {
public:
  double x, y;

  Point(double nx, double ny) {
    x = nx; y = ny;
  }
};
```

Access Modifiers

- public: can be accessed from anywhere

```
class Point {
public:
  double x, y;

  Point(double nx, double ny) {
    x = nx; y = ny;
  }
};

int main() {
  Point p(2.0,3.0);
  p.x = 5.0; // allowed
}
```

Access Modifiers

- private: can only be accessed within the class

```
class Point {
private:
  double x, y;

public:
  Point(double nx, double ny) {
    x = nx; y = ny;
  }
};

int main() {
  Point p(2.0,3.0);
  p.x = 5.0; // not allowed
}
```

Access Modifiers

- Use getters to allow read-only access to private fields

```
class Point {
private:
  double x, y;

public:
  Point(double nx, double ny) {
    x = nx; y = ny;
  }
  double getX() { return x; }
  double getY() { return y; }
};

int main() {
  Point p(2.0,3.0);
  cout << p.getX() << endl; // allowed
}
```

Default Access Modifiers

- class: private by default

```
class Point {
  double x, y;
};
```

Equivalent to

```
class Point {
private:
  double x, y;
};
```

Structs

- Structs are a carry-over from the C; in C++, classes are generally used
- In C++, they're essentially the same as classes, except structs' default access modifier is public

```
class Point {
public:
  double x;
  double y;
};
```

```
struct Point {

  double x;
  double y;
};
```

Default Access Modifiers

- struct: public by default
- class: private by default

```
struct Point {
  double x, y;
};
```

Equivalent to

```
struct Point {
public:
  double x, y;
};
```

```
class Point {
  double x, y;
};
```

Equivalent to

```
class Point {
private:
  double x, y;
};
```

Lecture 7 Notes: Object-Oriented Programming (OOP) and Inheritance

We've already seen how to define composite datatypes using classes. Now we'll take a step back and consider the programming philosophy underlying classes, known as *object-oriented programming* (OOP).

1 The Basic Ideas of OOP

Classic "procedural" programming languages before C++ (such as C) often focused on the question "What should the program do next?" The way you structure a program in these languages is:

1. Split it up into a set of tasks and subtasks
2. Make functions for the tasks
3. Instruct the computer to perform them in sequence

With large amounts of data and/or large numbers of tasks, this makes for complex and unmaintainable programs.

Consider the task of modeling the operation of a car. Such a program would have lots of separate variables storing information on various car parts, and there'd be no way to group together all the code that relates to, say, the wheels. It's hard to keep all these variables and the connections between all the functions in mind.

To manage this complexity, it's nicer to package up self-sufficient, modular pieces of code. People think of the world in terms of interacting *objects*: we'd talk about interactions between the steering wheel, the pedals, the wheels, etc. OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

There are lots of definitions for OOP, but 3 primary features of it are:

- **Encapsulation:** grouping related data and functions together as objects and defining an *interface* to those objects
- **Inheritance:** allowing code to be reused between related types
- **Polymorphism:** allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Let's see how each of these plays out in C++.

2 Encapsulation

Encapsulation just refers to packaging related stuff together. We've already seen how to package up data and the operations it supports in C++: with classes.

If someone hands us a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data – its *interface*. This is often compared to operating a car: when you drive, you don't care how the steering wheel makes the wheels turn; you just care that the interface the car presents (the steering wheel) allows you to accomplish your goal. If you remember the analogy from Lecture 6 about objects being boxes with buttons you can push, you can also think of the interface of a class as the set of buttons each instance of that class makes available. Interfaces abstract away the details of how all the operations are actually performed, allowing the programmer to focus on how objects will use each other's interfaces – how they interact.

This is why C++ makes you specify `public` and `private` access specifiers: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is called "data hiding," or making your class a "black box."

One way to think about what happens in an object-oriented program is that we define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other's methods) to exchange information and tell each other what to do.

3 Inheritance

Inheritance allows us to define hierarchies of related classes.

Imagine we're writing an inventory program for vehicles, including cars and trucks. We could write one class for representing cars and an unrelated one for representing trucks, but we'd have to duplicate the functionality that all vehicles have in common. Instead, C++ allows us to specify the common code in a `Vehicle` class, and then specify that the `Car` and `Truck` classes share this code.

The `Vehicle` class will be much the same as what we've seen before:

```
class Vehicle {
protected:
    string license;
    int year;
```

```
 5
 6 public:
 7     Vehicle(const string &myLicense, const int myYear)
 8         : license(myLicense), year(myYear) {}
 9     const string getDesc() const
10         {return license + " from " + stringify(year);}
11     const string &getLicense() const {return license;}
12     const int getYear() const {return year;}
13 };
```

A few notes on this code, by line:

2. The standard `string` class is described in Section 1 of PS3; see there for details. Recall that `string`s can be appended to each other with the `+` operator.

3. `protected` is largely equivalent to `private`. We'll discuss the differences shortly.

8. This line demonstrates *member initializer syntax*. When defining a constructor, you sometimes want to initialize certain members, particularly `const` members, even before the constructor body. You simply put a colon before the function body, followed by a comma-separated list of items of the form `dataMember(initialValue)`.

10. This line assumes the existence of some function `stringify` for converting numbers to `string`s.

Now we want to specify that `Car` will inherit the `Vehicle` code, but with some additions. This is accomplished in line 1 below:

```
1 class Car : public Vehicle { // Makes Car inherit from Vehicle
2     string style;
3
4 public:
5     Car(const string &myLicense, const int myYear, const string
         &myStyle)
6         : Vehicle(myLicense, myYear), style(myStyle) {}
7     const string &getStyle() {return style;}
8 };
```

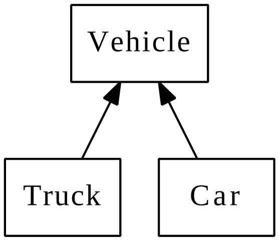
Now class `Car` has all the data members and methods of `Vehicle`, as well as a `style` data member and a `getStyle` method.

Class `Car` *inherits from* class `Vehicle`. This is equivalent to saying that `Car` is a *derived class*, while `Vehicle` is its *base class*. You may also hear the terms *subclass* and *superclass* instead.

Notes on the code:

1. Don't worry for now about why we stuck the `public` keyword in there.
6. Note how we use member initializer syntax to call the base-class constructor. We need to have a complete `Vehicle` object constructed before we construct the components added in the `Car`. If you do not explicitly call a base-class constructor using this syntax, the default base-class constructor will be called.

Similarly, we could make a `Truck` class that inherits from `Vehicle` and shares its code. This would give a *class hierarchy* like the following:



Class hierarchies are generally drawn with arrows pointing from derived classes to base classes.

3.1 Is-a vs. Has-a

There are two ways we could describe some class `A` as depending on some other class `B`:

1. Every `A` object *has a* `B` object. For instance, every `Vehicle` *has a* `string` object (called `license`).
2. Every instance of `A` *is a* `B` instance. For instance, every `Car` *is a* `Vehicle`, as well.

Inheritance allows us to define "is-a" relationships, but it should not be used to implement "has-a" relationships. It would be a design error to make `Vehicle` inherit from `string` because every `Vehicle` has a license; a `Vehicle` is not a `string`. "Has-a" relationships should be implemented by declaring data members, not by inheritance.

3.2 Overriding Methods

We might want to generate the description for `Car`s in a different way from generic `Vehicle`s. To accomplish this, we can simply redefine the `getDesc` method in `Car`, as below. Then, when we call `getDesc` on a `Car` object, it will use the redefined function. Redefining in this manner is called *overriding* the function.

```
class Car : public Vehicle { // Makes Car inherit from Vehicle
    string style;

public:
    Car(const string &myLicense, const int myYear, const string &myStyle)
        : Vehicle(myLicense, myYear), style(myStyle) {}
    const string getDesc() // Overriding this member function
        {return stringify(year) + ' ' + style + ": " + license;}
    const string &getStyle() {return style;}
};
```

3.2.1 Programming by Difference

In defining derived classes, we only need to specify what's different about them from their base classes. This powerful technique is called *programming by difference*.

Inheritance allows only overriding methods and adding new members and methods. We cannot remove functionality that was present in the base class.

3.3 Access Modifiers and Inheritance

If we'd declared `year` and `license` as `private` in `Vehicle`, we wouldn't be able to access them even from a derived class like `Car`. To allow derived classes but not outside code to access data members and member functions, we must declare them as `protected`.

The `public` keyword used in specifying a base class (e.g., `class Car : public Vehicle {...}`) gives a limit for the visibility of the inherited methods in the derived class. Normally you should just use `public` here, which means that inherited methods declared as `public` are still `public` in the derived class. Specifying `protected` would make inherited methods, even those declared `public`, have at most `protected` visibility. For a full table of the effects of different inheritance access specifiers, see
http://en.wikibooks.org/wiki/C++_Programming/Classes/Inheritance.

4 Polymorphism

Polymorphism means "many shapes." It refers to the ability of one object to have many types. If we have a function that expects a `Vehicle` object, we can safely pass it a `Car` object, because every `Car` is also a `Vehicle`. Likewise for references and pointers: anywhere you can use a `Vehicle *`, you can use a `Car *`.

4.1 virtual Functions

There is still a problem. Take the following example:

```
1 Car c("VANITY", 2003);
2 Vehicle *vPtr = &c;
3 cout << vPtr->getDesc();
```

(The `->` notation on line 3 just dereferences and gets a member. `ptr->member` is equivalent to `(*ptr).member`.)

Because `vPtr` is declared as a `Vehicle *`, this will call the `Vehicle` version of `getDesc`, even though the object pointed to is actually a `Car`. Usually we'd want the program to select the correct function at runtime based on which kind of object is pointed to. We can get this behavior by adding the keyword `virtual` before the method definition:

```
class Vehicle {
    ...
    virtual const string getDesc() {...}
};
```

With this definition, the code above would correctly select the `Car` version of `getDesc`.

Selecting the correct function at runtime is called *dynamic dispatch*. This matches the whole OOP idea – we're sending a message to the object and letting it figure out for itself what actions that message actually means it should take.

Because references are implicitly using pointers, the same issues apply to references:

```
Car c("VANITY", 2003);
Vehicle &v = c;
cout << v.getDesc();
```

This will only call the `Car` version of `getDesc` if `getDesc` is declared as `virtual`.

Once a method is declared `virtual` in some class `C`, it is virtual in every derived class of `C`, even if not explicitly declared as such. However, it is a good idea to declare it as `virtual` in the derived classes anyway for clarity.

4.2 Pure virtual Functions

Arguably, there is no reasonable way to define `getDesc` for a generic `Vehicle` – only derived classes really need a definition of it, since there is no such thing as a generic vehicle that isn't also a car, truck, or the like. Still, we do want to require every derived class of `Vehicle` to have this function.

We can omit the definition of getDesc from Vehicle by making the function *pure virtual* via the following odd syntax:

```
class Vehicle {
    ...
    virtual const string getDesc() = 0; // Pure virtual
};
```

The = 0 indicates that no definition will be given. This implies that one can no longer create an instance of Vehicle; one can only create instances of Cars, Trucks, and other derived classes which do implement the getDesc method. Vehicle is then an *abstract class* – one which defines only an interface, but doesn't actually implement it, and therefore cannot be instantiated.

5 Multiple Inheritance

Unlike many object-oriented languages, C++ allows a class to have multiple base classes:

```
class Car : public Vehicle, public InsuredItem {
    ...
};
```

This specifies that Car should have all the members of both the Vehicle and the InsuredItem classes.

Multiple inheritance is tricky and potentially dangerous:

- If both Vehicle and InsuredItem define a member x, you must remember to disambiguate which one you're referring to by saying Vehicle::x or InsuredItem::x.
- If both Vehicle and InsuredItem inherited from the same base class, you'd end up with two instances of the base class within each Car (a "dreaded diamond" class hierarchy). There are ways to solve this problem, but it can get messy.

In general, avoid multiple inheritance unless you know exactly what you're doing.

Lecture 8:
Memory Management

Clean up after your pet program

Review: Constructors

- Method that is called when an instance is created

```
class Integer {
public:
  int val;
  Integer() {
    val = 0; cout << "default constructor" << endl;
  }
};

int main() {
  Integer i;
}
```

Output:

```
default constructor
```

- When making an array of objects, default constructor is invoked on each

```
class Integer {
public:
  int val;
  Integer() {
    val = 0; cout << "default constructor" << endl;
  }
};

int main() {
  Integer arr[3];
}
```

Output:

```
default constructor
default constructor
default constructor
```

- When making a class instance, the default constructor of its fields are invoked

```cpp
class Integer {
public:
  int val;
  Integer() {
    val = 0; cout << "Integer default constructor" << endl;
  }
};
class IntegerWrapper {
public:
  Integer val;
  IntegerWrapper() {
    cout << "IntegerWrapper default constructor" << endl;
  }
};

int main() {
  IntegerWrapper q;
}
```

Output:

```
Integer default constructor
IntegerWrapper default constructor
```

- Constructors can accept parameters

```
class Integer {
public:
  int val;
  Integer(int v) {
    val = v; cout << "constructor with arg " << v << endl;
  }
};

int main() {
  Integer i(3);
}
```

Output:

```
constructor with arg 3
```

- Constructors can accept parameters
  - Can invoke single-parameter constructor via assignment to the appropriate type

```
class Integer {
public:
  int val;
  Integer(int v) {
    val = v; cout << "constructor with arg " << v << endl;
  }
};

int main() {
  Integer i(3);
  Integer j = 5;
}
```

Output:

```
constructor with arg 3
constructor with arg 5
```

- If a constructor with parameters is defined, the default constructor is no longer available

```
class Integer {
public:
  int val;
  Integer(int v) {
    val = v;
  }
};

int main() {
  Integer i(3); // ok
  Integer j;
}
```

Error: No default constructor available for Integer

- If a constructor with parameters is defined, the default constructor is no longer available
  - Without a default constructor, can’t declare arrays without initializing

```
class Integer {
public:
  int val;
  Integer(int v) {
    val = v;
  }
};

int main() {
  Integer a[] = { Integer(2), Integer(5) }; // ok
  Integer b[2];
}
```

Error: No default constructor available for Integer

- If a constructor with parameters is defined, the default constructor is no longer available
  - Can create a separate 0-argument constructor

```
class Integer {
public:
  int val;
  Integer() {
    val = 0;
  }
  Integer(int v) {
    val = v;
  }
};

int main() {
  Integer i; // ok
  Integer j(3); // ok
}
```

- If a constructor with parameters is defined, the default constructor is no longer available
  - Can create a separate 0-argument constructor
  - Or, use default arguments

```
class Integer {
public:
  int val;
  Integer(int v = 0) {
    val = v;
  }
};

int main() {
  Integer i; // ok
  Integer j(3); // ok
}
```

- How do I refer to a field when a method argument has the same name?
- **this**: a pointer to the current instance

```
class Integer {
public:
  int val;
  Integer(int val = 0) {
    this->val = val;
  }
};
```

this->val is a shorthand for (*this).val

- How do I refer to a field when a method argument has the same name?
- **this**: a pointer to the current instance

```
class Integer {
public:
  int val;
  Integer(int val = 0) {
    this->val = val;
  }
  void setVal(int val) {
    this->val = val;
  }
};
```

Scoping and Memory

- Whenever we declare a new variable (int x), memory is allocated
- When can this memory be freed up (so it can be used to store other variables)?
  - When the variable goes out of scope

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {
  if (true) {
    int x = 5;
  }
  // x now out of scope, memory it used to occupy can be reused
}
```

Scoping and Memory

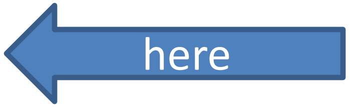
- When a variable goes out of scope, that memory is no longer guaranteed to store the variable’s value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  cout << *p << endl; // ???
}
```

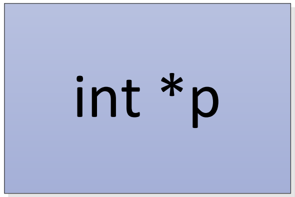
Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable’s value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  cout << *p << endl; // ???
}
```



here



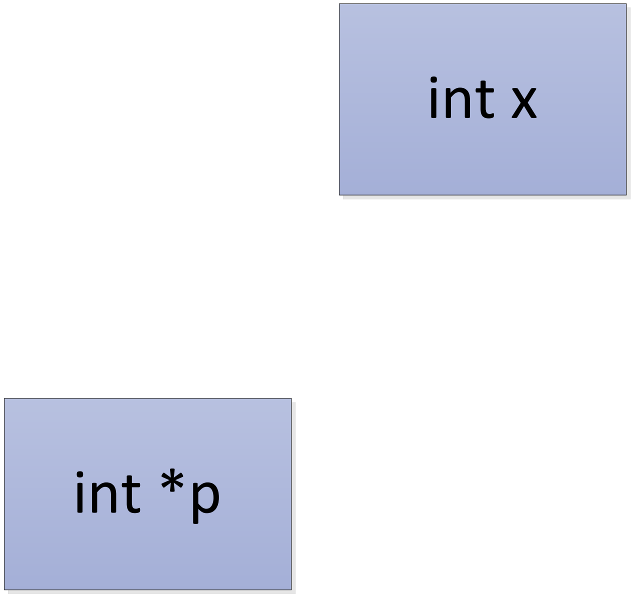
int *p

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  cout << *p << endl; // ???
}
```

here

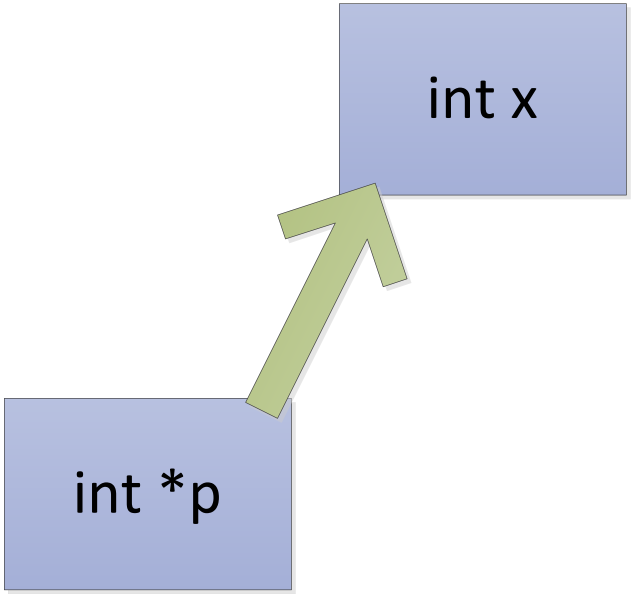


Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  cout << *p << endl; // ???
}
```

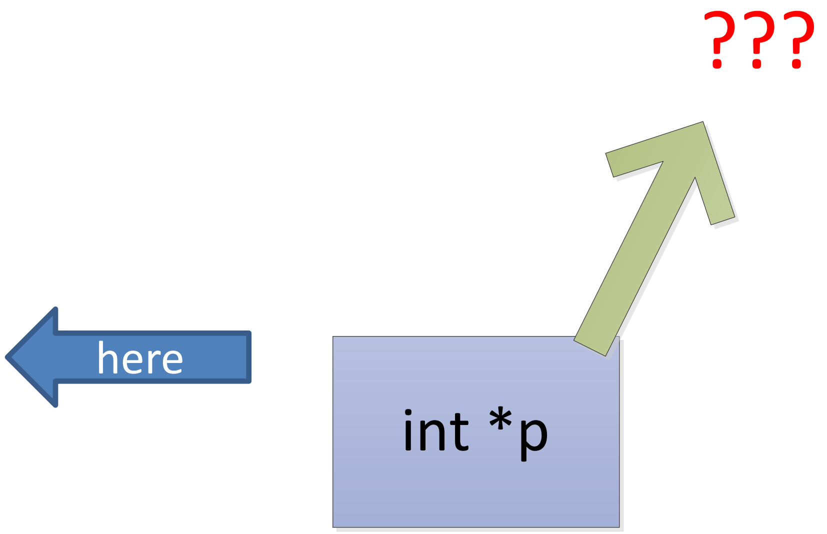
here



Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value
  - Here, p has become a **dangling pointer** (points to memory whose contents are undefined)

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  cout << *p << endl; // ???
}
```



A Problematic Task

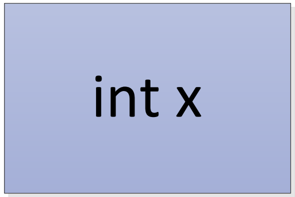
- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:

```
int* getPtrToFive() {
  int x = 5;
  return &v;
}
```

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:
  - x is declared in the function scope

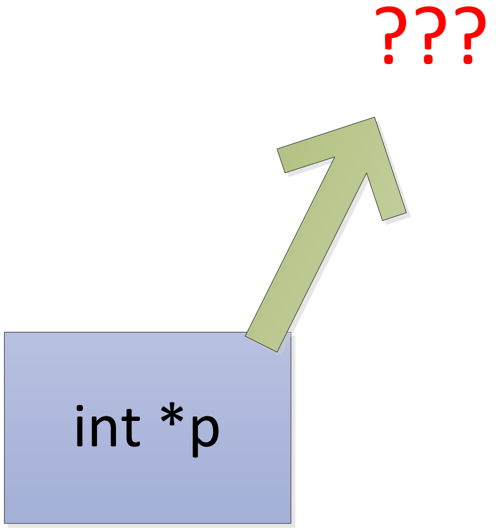
```
int* getPtrToFive() {
  int x = 5;
  return &x;
}

int main() {
  int *p = getPtrToFive();
  cout << *p << endl; // ???
}
```

here



- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:
  - x is declared in the function scope
  - As getPtrToFive() returns, x goes out of scope. So a dangling pointer is returned

```
int* getPtrToFive() {
  int x = 5;
  return &x;
}

int main() {
  int *p = getPtrToFive();
  cout << *p << endl; // ???
}
```



The **new** operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory

```
int *x = new int;
```

The **new** operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory

```
int *x = new int;
```

Type parameter needed to determine how much memory to allocate

The **new** operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory
- Terminology note:
  - If using **int x;** the allocation occurs on a region of memory called **the stack**
  - If using **new int;** the allocation occurs on a region of memory called **the heap**

The **delete** operator

- De-allocates memory that was previously allocated using **new**
- Takes a pointer to the memory location

```
int *x = new int;
// use memory allocated by new
delete x;
```

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate memory using **new** to ensure it remains allocated

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}
```

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate memory using **new** to ensure it remains allocated.
  - When done, de-allocate the memory using **delete**

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p = getPtrToFive();
  cout << *p << endl; // 5
  delete p;
}
```

Delete Memory When Done Using It

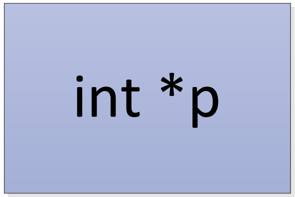
- If you don’t use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```

incorrect

- If you don’t use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```

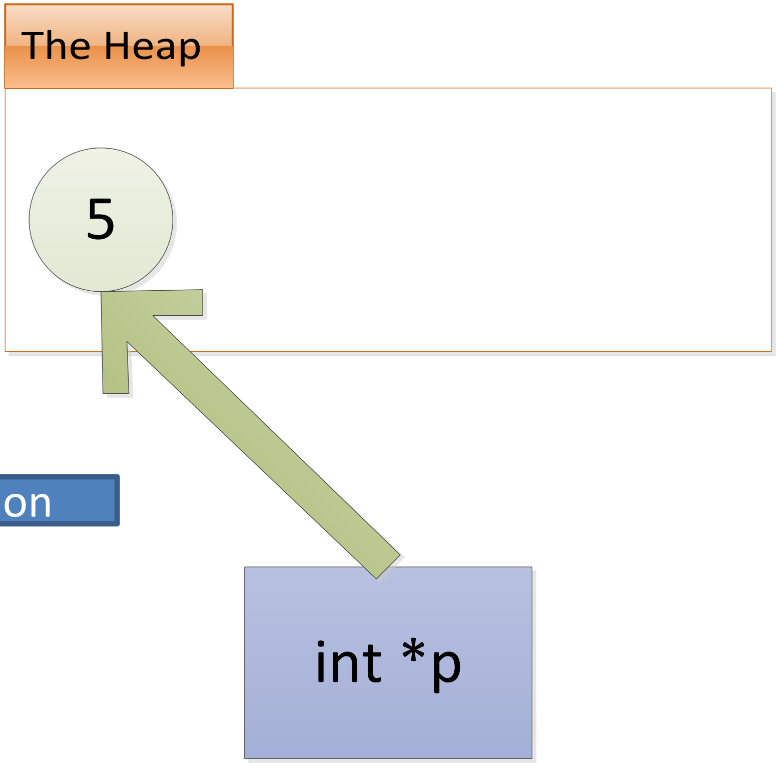


int *p

- If you don’t use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```

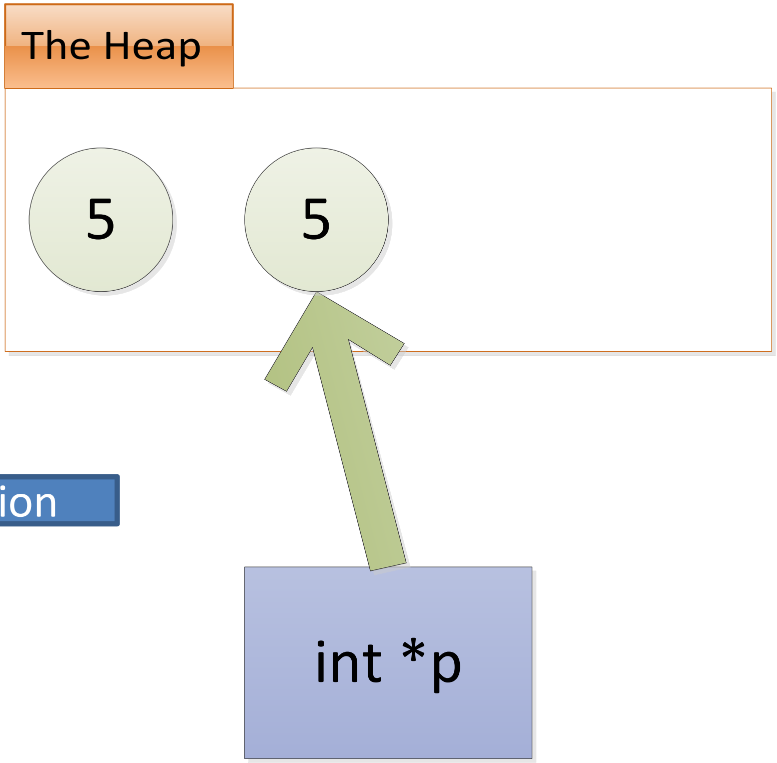
1st iteration



- If you don’t use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```

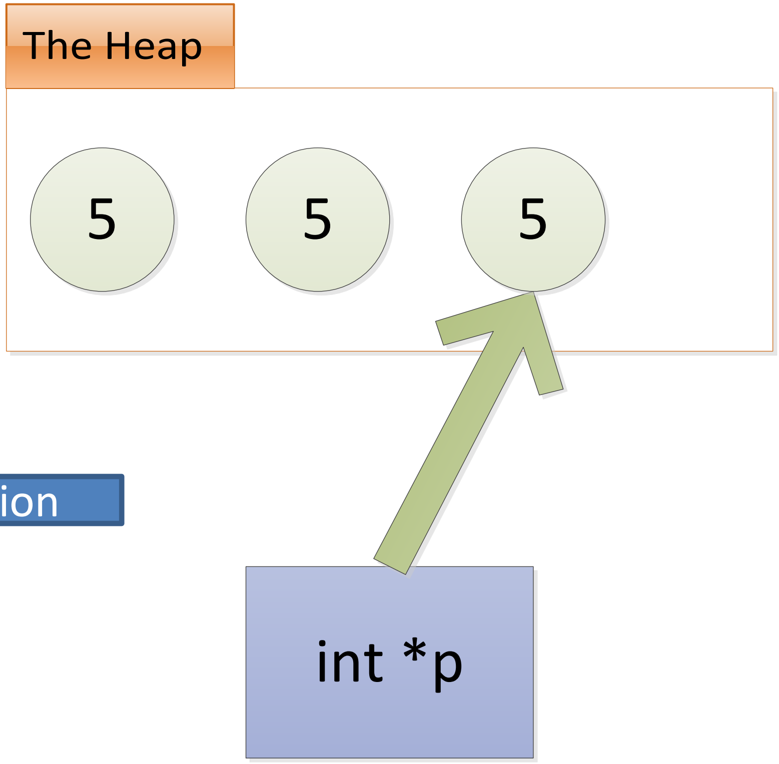
2nd iteration



- If you don’t use de-allocate memory using **delete**, your application will waste memory
- When your program allocates memory but is unable to de-allocate it, this is a **memory leak**

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```

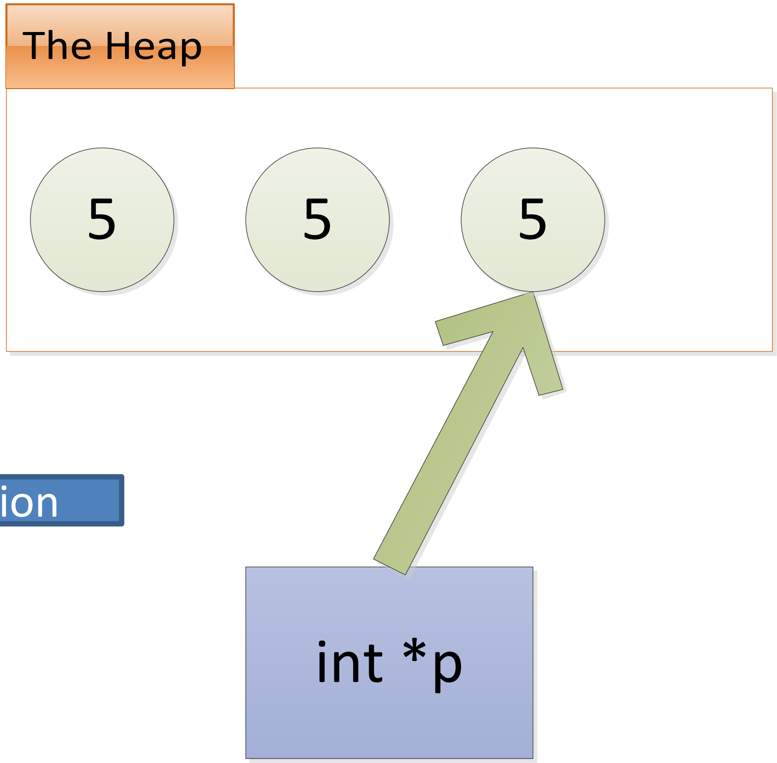
3rd iteration



- Does adding a delete after the loop fix this memory leak?

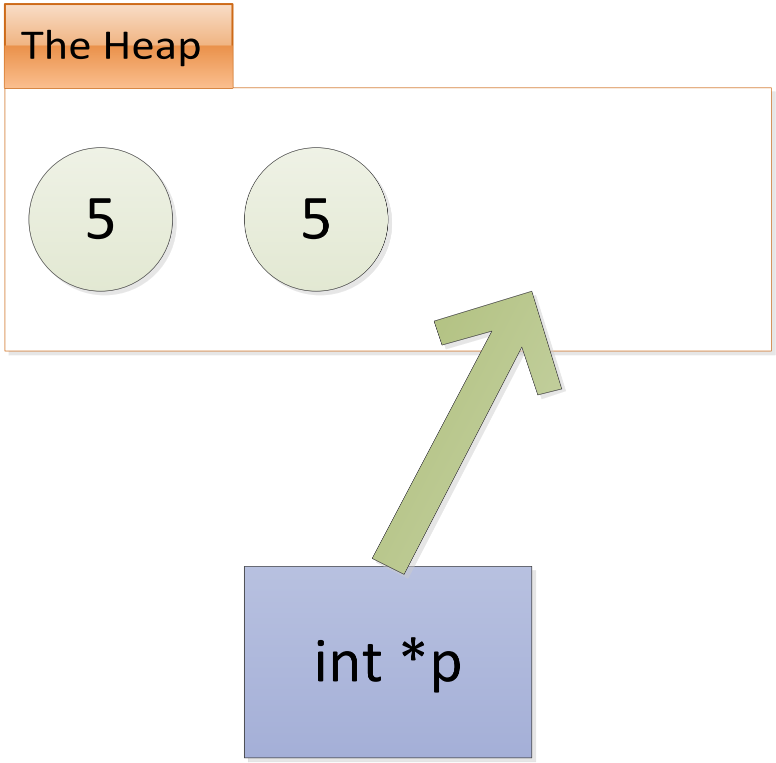
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
  delete p;
}
```

3rd iteration



- Does adding a delete after the loop fix this memory leak?
  - No; only the memory that was allocated on the last iteration gets de-allocated

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
  delete p;
}
```

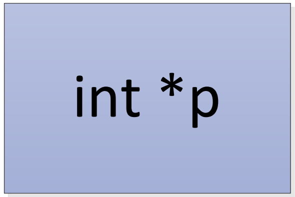


- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

- To fix the memory leak, de-allocate memory within the loop

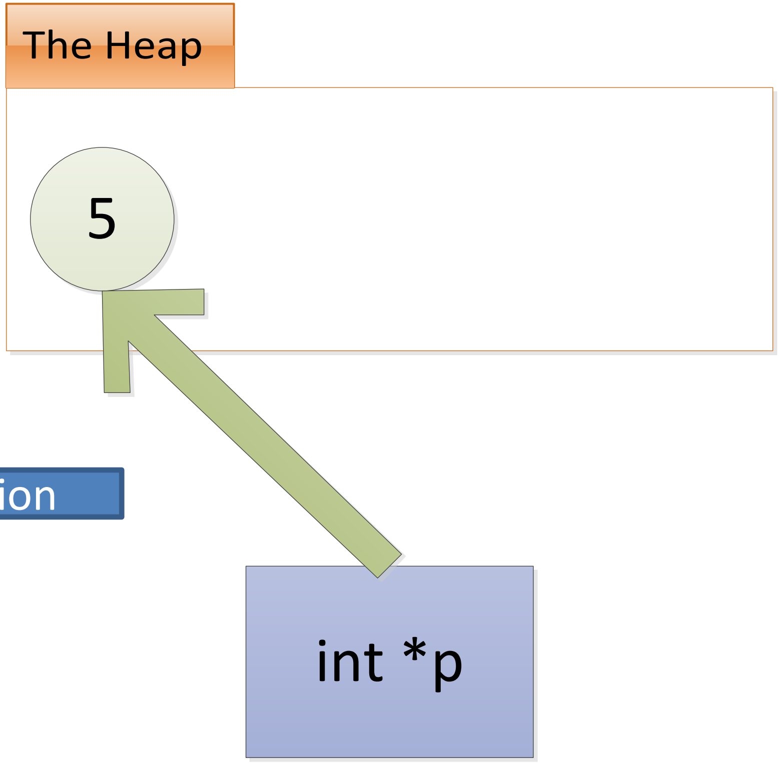
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

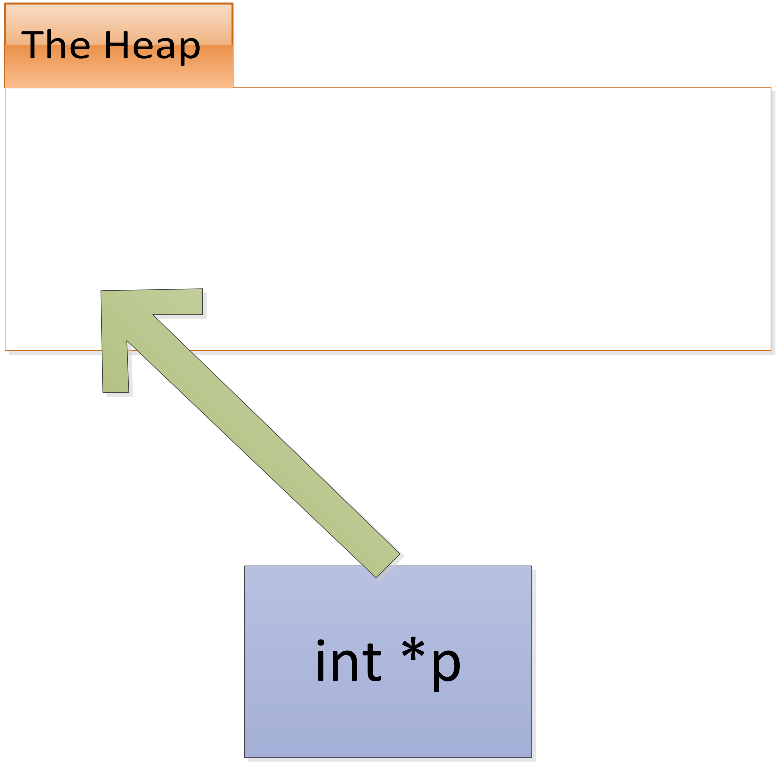
1st iteration



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

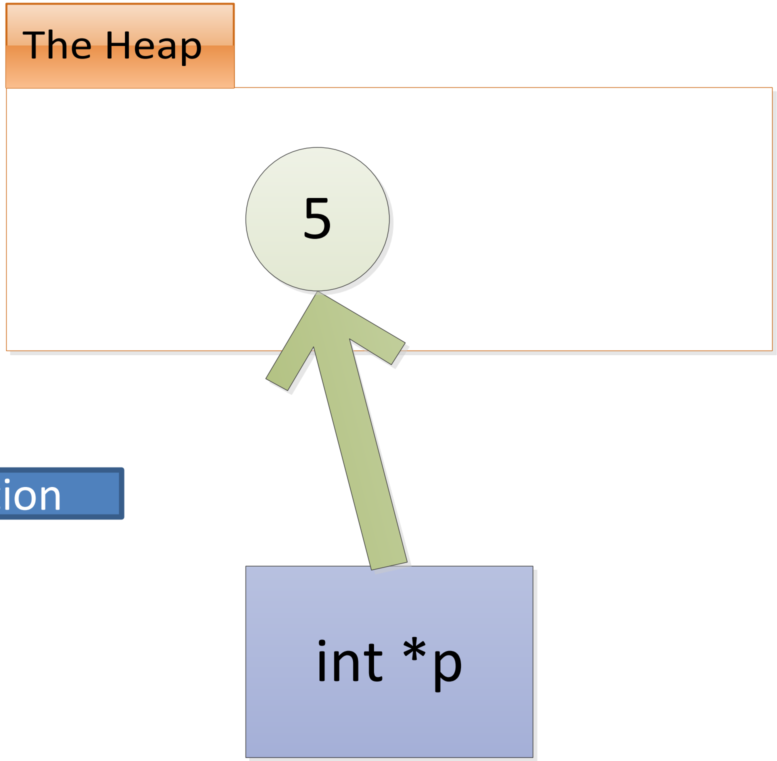
1st iteration



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

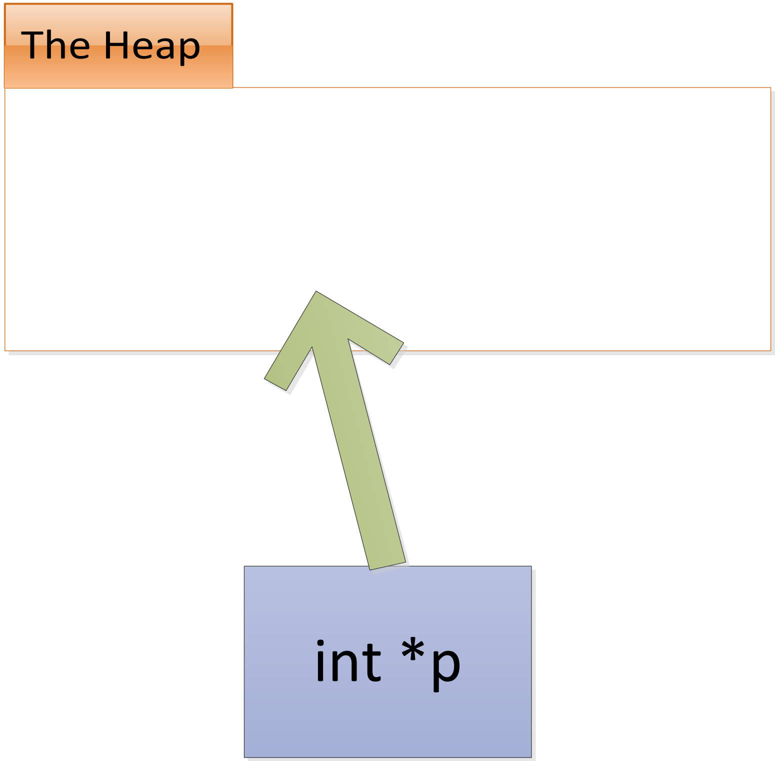
2nd iteration



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

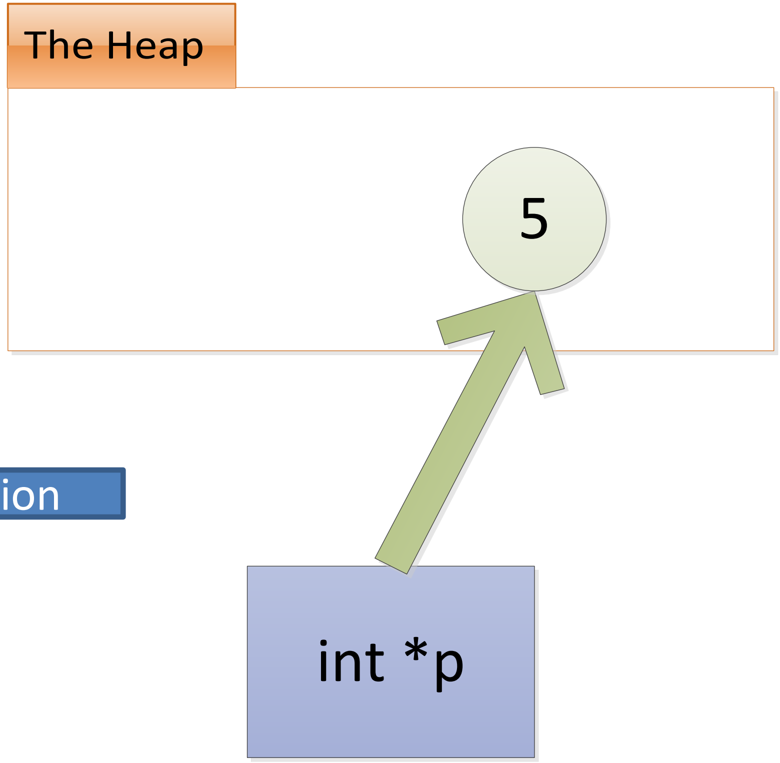
2^{nd} iteration



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

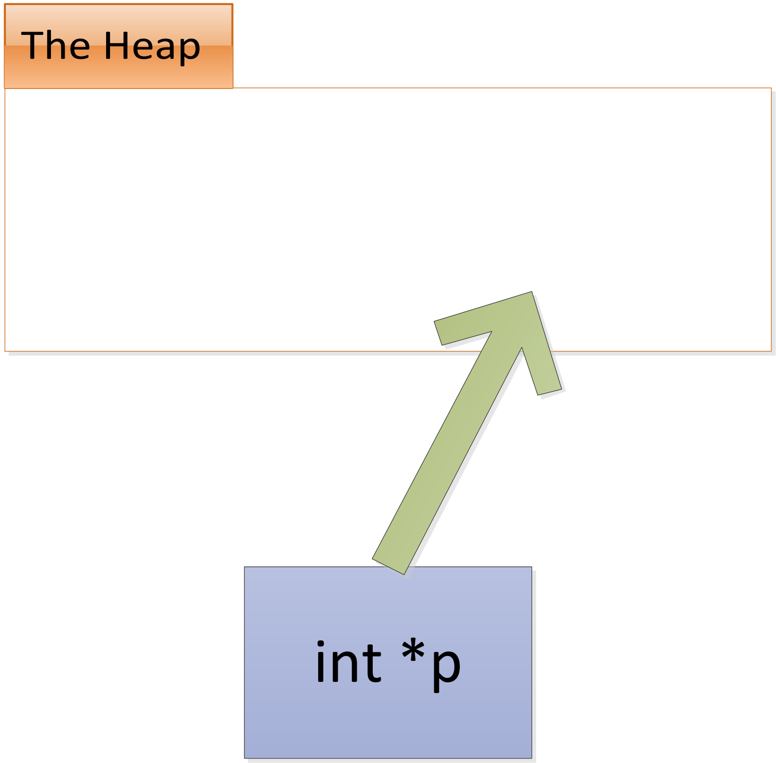
3rd iteration



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
    delete p;
  }
}
```

3rd iteration



Don’t Use Memory After Deletion

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  delete x;
  cout << *x << endl; // ???
}
```

Don’t Use Memory After Deletion

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  delete x;
  cout << *x << endl; // ???
}
```

correct

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
}
```

Don’t delete memory twice

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
  delete x;
}
```

Don’t delete memory twice

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
  delete x;
}
```

correct

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
}
```

Only **delete** if memory was allocated by **new**

incorrect

```
int main() {
  int x = 5;
  int *xPtr = &x;
  cout << *xPtr << endl;
  delete xPtr;
}
```

Only **delete** if memory was allocated by **new**

incorrect

```
int main() {
  int x = 5;
  int *xPtr = &x;
  cout << *xPtr << endl;
  delete xPtr;
}
```

correct

```
int main() {
  int x = 5;
  int *xPtr = &x;
  cout << *xPtr << endl;
}
```

Allocating Arrays

- When allocating arrays on the stack (using “int arr[SIZE]”), size must be a constant

```
int numItems;
cout << "how many items?";
cin >> numItems;
int arr[numItems]; // not allowed
```

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size

```
int numItems;
cout << "how many items?";
cin >> numItems;
int *arr = new int[numItems];
```

Type of items in array

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size

```
int numItems;
cout << "how many items?";
cin >> numItems;
int *arr = new int[numItems];
```

Number of items to allocate

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size
- De-allocate arrays with **delete[]**

```
int numItems;
cout << "how many items?";
cin >> numItems;
int *arr = new int[numItems];
delete[] arr;
```

Ex: Storing values input by the user

```
int main() {
  int numItems;
  cout << "how many items? ";
  cin >> numItems;
  int *arr = new int[numItems];
  for (int i = 0; i < numItems; ++i) {
    cout << "enter item " << i << ": ";
    cin >> arr[i];
  }
  for (int i = 0; i < numItems; ++i) {
    cout << arr[i] << endl;
  }
  delete[] arr;
}
```

```
how many items? 3
enter item 0: 7
enter item 1: 4
enter item 2: 9
7
4
9
```

Allocating Class Instances using **new**

- **new** can also be used to allocate a class instance

```
class Point {
public:
  int x, y;
};

int main() {
  Point *p = new Point;
  delete p;
}
```

Allocating Class Instances using **new**

- **new** can also be used to allocate a class instance
- The appropriate constructor will be invoked

```
class Point {
public:
  int x, y;
  Point() {
    x = 0; y = 0; cout << "default constructor" << endl;
  }
};

int main() {
  Point *p = new Point;
  delete p;
}
```

Output:

```
default constructor
```

Allocating Class Instances using **new**

- **new** can also be used to allocate a class instance
- The appropriate constructor will be invoked

```
class Point {
public:
  int x, y;
  Point(int nx, int ny) {
    x = ny; x = ny; cout << "2-arg constructor" << endl;
  }
};

int main() {
  Point *p = new Point(2, 4);
  delete p;
}
```

Output:

```
2-arg constructor
```

Destructor

- Destructor is called when the class instance gets de-allocated

```
class Point {
public:
  int x, y;
  Point() {
    cout << "constructor invoked" << endl;
  }
  ~Point() {
    cout << "destructor invoked" << endl;
  }
}
```

- Destructor is called when the class instance gets de-allocated
  - If allocated with **new**, when **delete** is called

```
class Point {
public:
  int x, y;
  Point() {
    cout << "constructor invoked" << endl;
  }
  ~Point() {
    cout << "destructor invoked" << endl;
  }
};
int main() {
  Point *p = new Point;
  delete p;
}
```

Output:

```
constructor invoked
destructor invoked
```

- Destructor is called when the class instance gets de-allocated
  - If allocated with **new**, when **delete** is called
  - If stack-allocated, when it goes out of scope

```
class Point {
public:
  int x, y;
  Point() {
    cout << "constructor invoked" << endl;
  }
  ~Point() {
    cout << "destructor invoked" << endl;
  }
};
int main() {
  if (true) {
    Point p;
  }
  cout << "p out of scope" << endl;
}
```

Output:

```
constructor invoked
destructor invoked
p out of scope
```

Representing an Array of Integers

- When representing an array, often pass around both the pointer to the first element and the number of elements
  - Let’s make them fields in a class

```
class IntegerArray {
public:
  int *data;
  int size;
};
```

Pointer to the first element

Representing an Array of Integers

- When representing an array, often pass around both the pointer to the first element and the number of elements
  - Let’s make them fields in a class

```
class IntegerArray {
public:
  int *data;
  int size;
};
```

Number of elements in the array

```
class IntegerArray {
public:
  int *data;
  int size;
};

int main() {
  IntegerArray arr;
  arr.size = 2;
  arr.data = new int[arr.size];
  arr.data[0] = 4; arr.data[1] = 5;
  delete[] a.data;
}
```

```
class IntegerArray {
public:
  int *data;
  int size;
};

int main() {
  IntegerArray arr;
  arr.size = 2;
  arr.data = new int[arr.size];
  arr.data[0] = 4; arr.data[1] = 5;
  delete[] a.data;
}
```

Can move this into a constructor

```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
};

int main() {
  IntegerArray arr(2);
  arr.data[0] = 4; arr.data[1] = 5;
  delete[] arr.data;
}
```

```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
};

int main() {
  IntegerArray arr(2);
  arr.data[0] = 4; arr.data[1] = 5;
  delete[] arr.data;
}
```

Can move this into a destructor

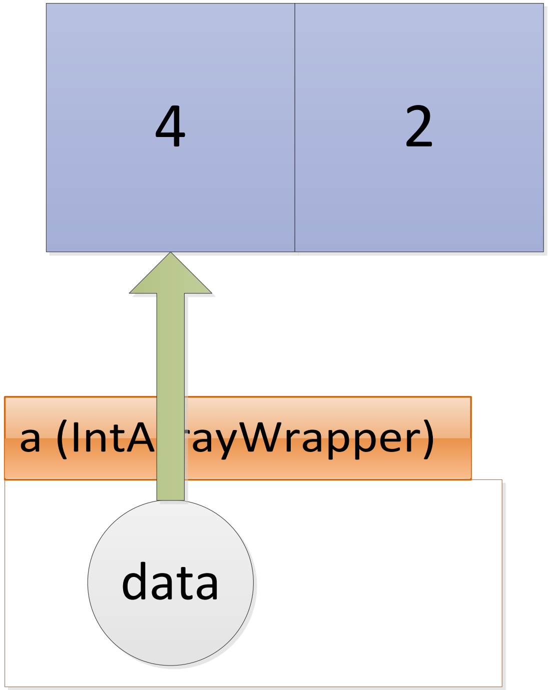
```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  ~IntegerArray () {
    delete[] data;
  }
};

int main() {
  IntegerArray arr(2);
  arr.data[0] = 4; arr.data[1] = 5;
}
```

De-allocate memory used by fields in destructor incorrect

```cpp
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  ~IntegerArray() {
    delete[] data;
  }
};

int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // not 4!
}
```

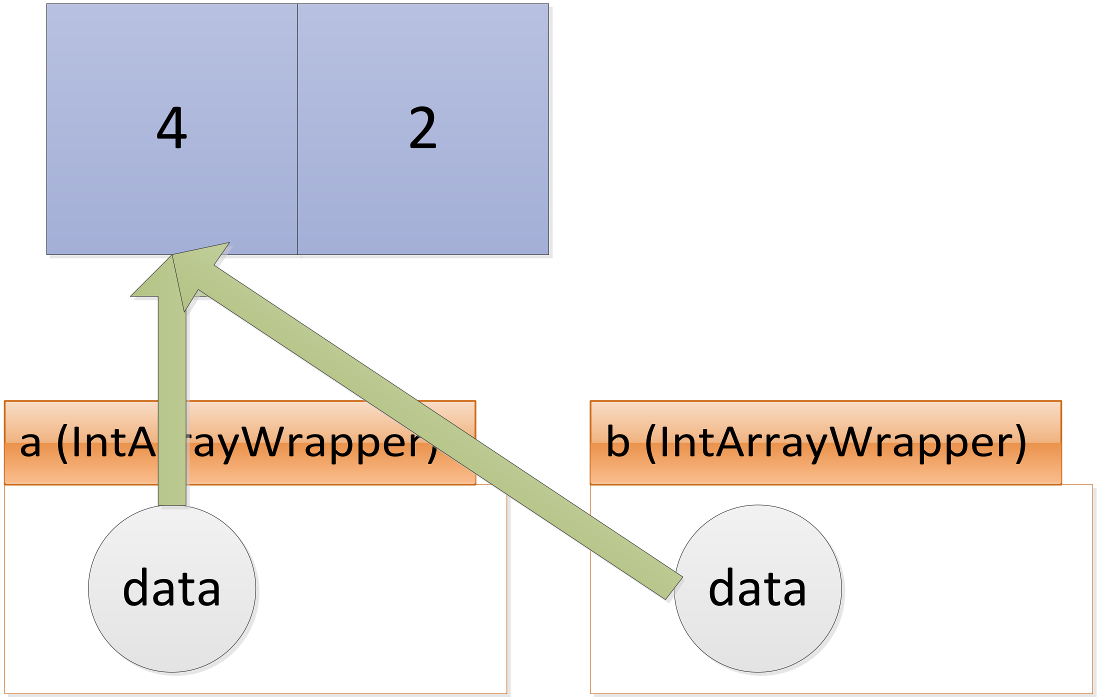
```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  ~IntegerArray() {
    delete[] data;
  }
};

int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // not 4!
}
```

here



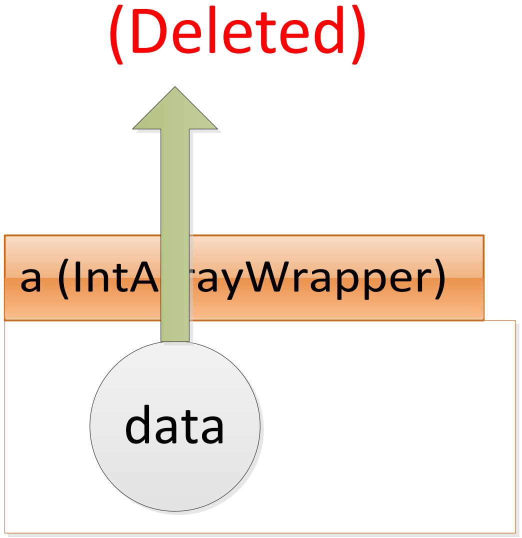
- Default copy constructor copies fields

```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  ~IntegerArray() {
    delete[] data;
  }
};

int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // not 4!
}
```



- When b goes out of scope, destructor is called (de-allocates array), a.data now a dangling pointer

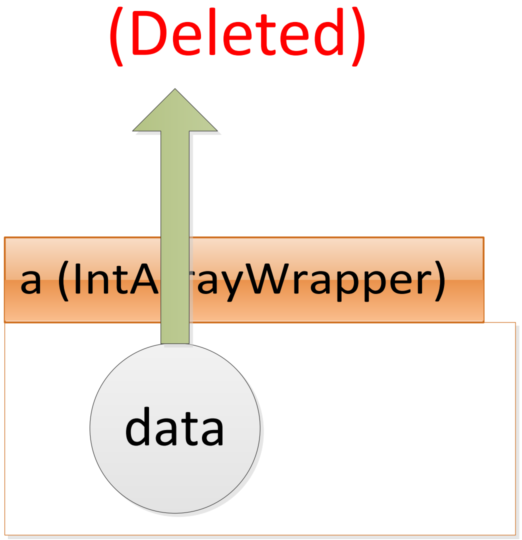
```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  ~IntegerArray() {
    delete[] data;
  }
};

int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // not 4!
}
```



here

- 2nd bug: when a goes out of scope, its destructor tries to delete the (already-deleted) array

```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  ~IntegerArray() {
    delete[] data;
  }
};

int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // not 4!
}
```



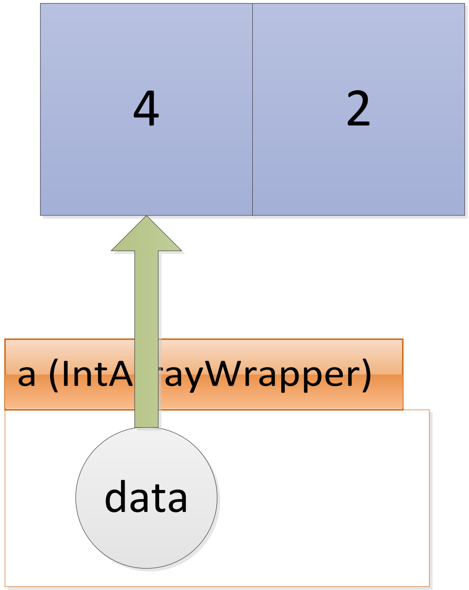
Program crashes as it terminates

- Write your own a copy constructor to fix these bugs

```
class IntegerArray {
public:
  int *data;
  int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  IntegerArray(IntegerArray &o) {
    data = new int[o.size];
    size = o.size;
    for (int i = 0; i < size; ++i)
      data[i] = o.data[i];
  }
  ~IntegerArray() {
    delete[] data;
  }
};
```

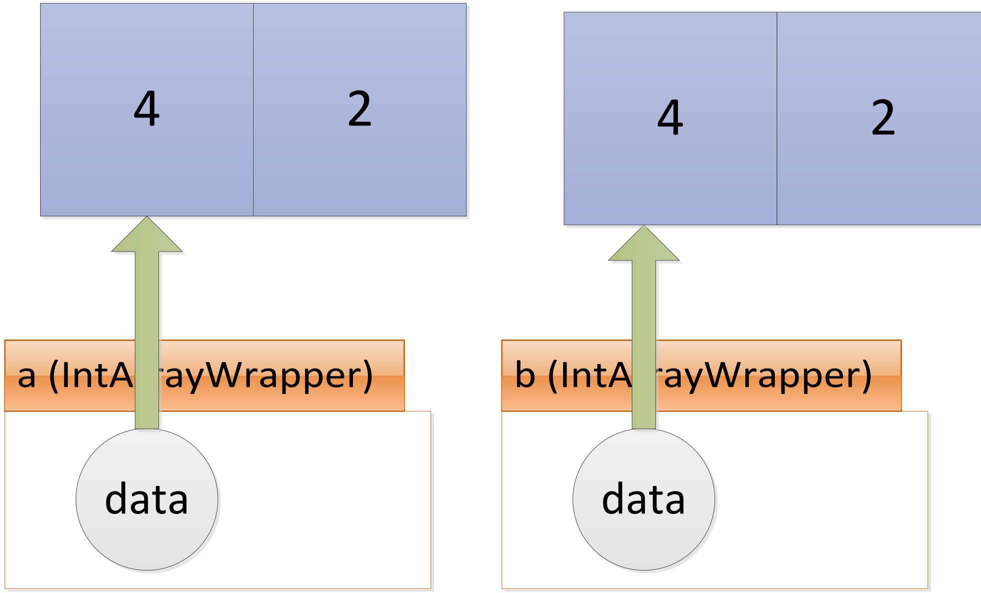
```
class IntegerArray {
public:
  int *data; int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  IntegerArray(IntegerArray &o) {
    data = new int[o.size];
    size = o.size;
    for (int i = 0; i < size; ++i)
      data[i] = o.data[i];
  }
  ~IntegerArray() {
    delete[] data;
  }
};
int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // 4
}
```

here



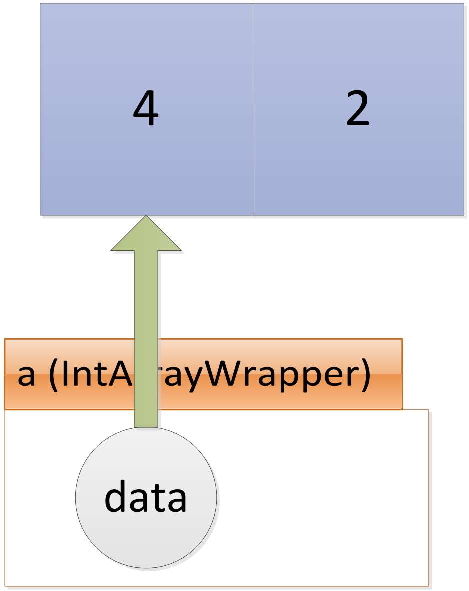
```
class IntegerArray {
public:
  int *data; int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  IntegerArray(IntegerArray &o) {
    data = new int[o.size];
    size = o.size;
    for (int i = 0; i < size; ++i)
      data[i] = o.data[i];
  }
  ~IntegerArray() {
    delete[] data;
  }
};
int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // 4
}
```

Copy constructor invoked



```
class IntegerArray {
public:
  int *data; int size;
  IntegerArray(int size) {
    data = new int[size];
    this->size = size;
  }
  IntegerArray(IntegerArray &o) {
    data = new int[o.size];
    size = o.size;
    for (int i = 0; i < size; ++i)
      data[i] = o.data[i];
  }
  ~IntegerArray() {
    delete[] data;
  }
};
int main() {
  IntegerArray a(2);
  a.data[0] = 4; a.data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  cout << a.data[0] << endl; // 4
}
```

here



Lecture 9 Notes: Advanced Topics I

1 Templates

We have seen that functions can take arguments of specific types and have a specific return type. We now consider *templates,* which allow us to work with *generic types.* Through templates, rather than repeating function code for each new type we wish to accommodate, we can create functions that are capable of using the same code for different types. For example:

```
int sum(const int x, const int y) {
    return x + y;
}
```

For this function to work with doubles, it must be modified to the following:

```
double sum (const double x, const double y) {
    return x + y;
}
```

For a simple function such as this, it may be a small matter to just make the change as shown, but if the code were much more complicated, copying the entire function for each new type can quickly become problematic. To overcome this we rewrite `sum` as a function template.

The format for declaring a function template is:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

Both forms are equivalent to one another, regardless of what type *identifier* ends up being. We can then use *identifier* to replace all occurrences of the type we wish to generalize.

So, we rewrite our `sum` function:

```
template <typename T>
T sum(const T a, const T b) {
    return a + b;
}
```

Now, when `sum` is called, it is called with a particular type, which will replace all `T`s in the code. To invoke a function template, we use:

```
function_name <type> (parameters);
```

Here is an example `main` function using the above `sum` function template:

```
int main() {
    cout << sum<int>(1, 2) << endl;
    cout << sum<float>(1.21, 2.43) << endl;
    return 0;
}
```

This program prints out `3` and `3.64` on separate lines.

The *identifier* can be used in any way inside the function template, as long as the code makes sense after *identifier* is replaced with some type.

It is also possible to invoke a function template without giving an explicit type, in cases where the generic type *identifier* is used as the type for a parameter for the function. In the above example, the following would also have been valid:

```
int main() {
    cout << sum(1, 2) << endl;
    cout << sum(1.21, 2.43) << endl;
    return 0;
}
```

Templates can also specify more than one type parameter. For example:

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

template <typename T, typename U>
U sum(const T a, const U b) {
   return a + b;
}

int main() {
   cout << sum<int, float>(1, 2.5) << endl;
   return 0;
}
```

This program prints out `3.5`. In this case we can also call sum by writing `sum(1, 2.5)`.

Class templates are also possible, in much the same way we have written function templates:

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

template <typename T>
class Point {
   private:
         T x, y;
   public:
         Point(const T u, const T v) : x(u), y(v) {}
         T getX() { return x; }
         T getY() { return y; }
};

int main() {
   Point<float> fpoint(2.5, 3.5);
   cout << fpoint.getX() << ", " << fpoint.getY() << endl;
   return 0;
}
```

The program prints out `2.5, 3.5`.

To declare member functions externally, we use the following syntax:

```
template <typename T>
T classname<T>::function_name()
```

So, for example, getX could have been declared in the following way:

```
template <typename T>
T Point<T>::getX() { return x; }
```

assuming a prototype of `T getX();` inside the class definition.

We can also define different implementations for a single template by using *template specialization.* Consider the following example:

```
1  #include <iostream>
2  #include <cctype>
3  using namespace std;
4
5  template <typename T>
6  class Container {
7     private:
8           T elt;
9     public:
10          Container(const T arg) : elt(arg) {}
11          T inc() { return elt+1; }
12 };
13
14 template <>
15 class Container <char> {
16    private:
17          char elt;
18    public:
19          Container(const char arg) : elt(arg) {}
20          char uppercase() { return toupper(elt); }
21 };
22
23 int main() {
24    Container<int> icont(5);
25    Container<char> ccont('r');
26    cout << icont.inc() << endl;
27    cout << ccont.uppercase() << endl;
28    return 0;
29 }
```

This program prints out `6` and `R` on separate lines. Here, the class `Container` is given two implementations: a generic one and one specifically tailored to the `char` type. Notice the syntax at lines 14 and 15 when declaring a specialization.

Finally, it is possible to parametrize templates on regular types:

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

template <typename T, int N>
class ArrayContainer {
   private:
         T elts[N];
   public:
         T set(const int i, const T val) { elts[i] = val; }
         T get(const int i) { return elts[i]; }
};

int main() {
   ArrayContainer <int, 5> intac;
   ArrayContainer <float, 10> floatac;
   intac.set(2, 3);
   floatac.set(3, 3.5);
   cout << intac.get(2) << endl;
   cout << floatac.get(3) << endl;
```

```
  return 0;
}
```

This program prints out `3` and `3.5` on separate lines. Here, one instance of the ArrayContainer class works on a 5-element array of `int`s whereas the other instance works on a 10-element array of `float`s.

Default values can be set for template parameters. For example, the previous template definition could have been:

```
template <typename T=int, int N=5> class ArrayContainer { ... }
```

and we could have created an `ArrayContainer` using the default parameters by writing:

```
ArrayContainer<> identifier;
```

2 Standard Template Library

Part of the C++ Standard Library, the *Standard Template Library* (STL) contains many useful container classes and algorithms. As you might imagine, these various parts of the library are written using templates and so are generic in type. The containers found in the STL are lists, maps, queues, sets, stacks, and vectors. The algorithms include sequence operations, sorts, searches, merges, heap operations, and min/max operations. We will explore how to use some of these through example here:

```
1   #include <iostream>
2   #include <set>
3   #include <algorithm>
4   using namespace std;
5
6   int main() {
7     set<int> iset;
8     iset.insert(5);
9     iset.insert(9);
10    iset.insert(1);
11    iset.insert(8);
12    iset.insert(3);
13
14    cout << "iset contains:";
15    set<int>::iterator it;
16    for(it=iset.begin(); it != iset.end(); it++)
17          cout << " " << *it;
18    cout << endl;
19
20    int searchFor;
21    cin >> searchFor;
22    if(binary_search(iset.begin(), iset.end(), searchFor))
23          cout << "Found " << searchFor << endl;
24    else
25          cout << "Did not find " << searchFor << endl;
26
27    return 0;
28 }
```

In this example, we create an integer set and insert several integers into it. We then create an iterator corresponding to the set at lines 14 and 15. An iterator is basically a pointer that provides a view of the set. (Most of the other containers also provide iterators.) By using this iterator, we display all the elements in the set and print out `iset contains: 1 3 5 8 9`. Note that the set automatically sorts its own items. Finally, we ask the user for an integer, search for that integer in the set, and display the result.

Here is another example:

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

void printArray(const int arr[], const int len) {
   for(int i=0; i < len; i++)
         cout << " " << arr[i];
   cout << endl;
}

int main() {
   int a[] = {5, 7, 2, 1, 4, 3, 6};

   sort(a, a+7);
   printArray(a, 7);
   rotate(a,a+3,a+7);
   printArray(a, 7);
   reverse(a, a+7);
   printArray(a, 7);

   return 0;
}
```

This program prints out:

```
 1 2 3 4 5 6 7
 4 5 6 7 1 2 3
 3 2 1 7 6 5 4
```

The STL has many, many more containers and algorithms that you can use. Read more at http://www.cplusplus.com/reference/stl and http://www.cplusplus.com/reference/algorithm/.

3 Operator Overloading

We have been using operators on primitives, but sometimes it makes sense to use them on user-defined datatypes. For instance, consider the following struct:

```
struct USCurrency {
      int dollars;
      int cents;
};
```

Perhaps we would like to add two `USCurrency` objects together and get a new one as a result, just like in normal addition:

```
USCurrency a = {2, 50};
USCurrency b = {1, 75};
USCurrency c = a + b;
```

This of course gives a compiler error, but we can define behavior that our datatype should have when used with the addition operator by overloading the addition operator. This can be done either inside the class as part of its definition (the addition from the point of view of the object on the left side of the `+`):

```
USCurrency operator+(const USCurrency o) {
   USCurrency tmp = {0, 0};
   tmp.cents = cents + o.cents;
   tmp.dollars = dollars + o.dollars;
```

```

    if(tmp.cents >= 100) {
        tmp.dollars += 1;
        tmp.cents -= 100;
    }

    return tmp;
}
```

or outside the class as a function independent of the class (the addition from the point of view of the +):

```
USCurrency operator+(const USCurrency m, const USCurrency o) {
    USCurrency tmp = {0, 0};
    tmp.cents = m.cents + o.cents;
    tmp.dollars = m.dollars + o.dollars;

    if(tmp.cents >= 100) {
        tmp.dollars += 1;
        tmp.cents -= 100;
    }

    return tmp;
}
```

Similarly, we can overload the << operator to display the result:

```
ostream& operator<<(ostream &output, const USCurrency &o)
{
    output << "$" << o.dollars << "." << o.cents;
    return output;
}
```

Assuming the above definitions, we can run the following program:

```
int main() {
   USCurrency a = {2, 50};
   USCurrency b = {1, 75};
   USCurrency c = a + b;
   cout << c << endl;
   return 0;
}
```

and get the printout `$4.25`.

The list of overloadable operators:

+	-	*	/	+=	-=	*=	/=	%	%=	++	--
=	==	<	>	<=	>=	!	!=	&&	||		
<<	>>	<<=	>>=	&	^	|	&=	^=	|=	~	
[]	()	,	->*	->	new	new[]	delete		delete[]		

Lecture 10 Notes: Advanced Topics II

1 Stuff You May Want to Use in Your Project

1.1 File handling

File handling in C++ works almost identically to terminal input/output. To use files, you write `#include <fstream>` at the top of your source file. Then you can access two classes from the `std` namespace:

- `ifstream` – allows reading input from files
- `ofstream` – allows outputting to files

Each open file is represented by a separate `ifstream` or an `ofstream` object. You can use `ifstream` objects in excatly the same way as `cin` and `ofstream` objects in the same way as `cout`, except that you need to declare new objects and specify what files to open.

For example:

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

int main() {
  ifstream source("source-file.txt");
  ofstream destination("dest-file.txt");
  int x;
  source >> x;  // Reads one int from source-file.txt
  source.close(); // close file as soon as we're done using it
  destination << x; // Writes x to dest-file.txt
  return 0;
} // close() called on destination by its destructor
```

As an alternative to passing the filename to the constructor, you can use an existing `ifstream` or `ofstream` object to open a file by calling the `open` method on it: `source.open("other-file.txt");`.

Close your files using the `close()` method when you're done using them. This is automatically done for you in the object's destructor, but you often want to close the file ASAP, without waiting for the destructor.

You can specify a second argument to the constructor or the `open` method to specify what "mode" you want to access the file in – read-only, overwrite, write by appending, etc. Check documentation online for details.

1.2 Reading Strings

You'll likely find that you want to read some text input from the user. We've so far seen only how to do `int`s, `char`s, etc.

It's usually easiest to manage text using the C++ `string` class. You can read in a string from `cin` like other variables:

```
string mobileCarrier;
cin >> mobileCarrier;
```

However, this method only reads up to the first whitespace; it stops at any tab, space, newline, etc. If you want to read multiple words, you can use the `getline` function, which reads everything up until the user presses enter:

```
string sentence;
getline(cin, sentence);
```

1.3 enum

In many cases you'll find that you'll want to have a variable that represents one of a discrete set of values. For instance, you might be writing a card game and want to store the suit of a card, which can only be one of clubs, diamonds, hearts, and spades. One way you might do this is declaring a bunch of `const int`s, each of which is an ID for a particular suit. If you wanted to print the suit name for a particular ID, you might write this:

```
const int CLUBS = 0, DIAMONDS = 1, HEARTS = 2, SPADES = 3;
void print_suit(const int suit) {
    const char *names[] = {"Clubs", "Diamonds",
                           "Hearts", "Spades"};
    return names[suit];
}
```

The problem with this is that `suit` could be integer, not just one of the set of values we know it should be restricted to. We'd have to check in our function whether `suit` is too big. Also, there's no indication in the code that these `const int`s are related.

Instead, C++ allows us to use `enum`s. An `enum` just provides a set of named integer values which are the only legal values for some new type. For instance:

```
1 enum suit_t {CLUBS, DIAMONDS, HEARTS, SPADES};
2 void print_suit(const suit_t suit) {
3     const char *names[] = {"Clubs", "Diamonds",
4                            "Hearts", "Spades"};
5     return names[suit];
6 }
```

Now, it is illegal to pass anything but CLUBS, DIAMODNS, HEARTS, or SPADES into `print_suit`. However, internally the `suit_t` values are still just integers, and we can use them as such (as in line 5).

You can specify which integers you want them to be:

```
enum suit_t {CLUBS=18, DIAMONDS=91, HEARTS=241, SPADES=13};
```

The following rules are used by default to determine the values of the `enum` constants:

- The first item defaults to 0.
- Every other item defaults to the previous item plus 1.

Just like any other type, an `enum` type such as `suit_t` can be used for any arguments, variables, return types, etc.

1.4 Structuring Your Project

Many object-oriented programs like those you're writing for your projects share an overall structure you will likely want to use. They have some kind of managing class (e.g., `Game`, `Directory`, etc.) that maintains all the other objects that interact in the program. For instance, in a board game, you might have a `Game` class that is responsible for maintaining `Player` objects and the `Board` object. Often this class will have to maintain some collection of objects, such as a list of people or a deck of cards; it can do so by having a field that is an STL container. `main` creates a single instance of this managing class, handles the interaction with the user (i.e. asking the user what to do next), and calls methods on the manager object to perform the appropriate actions based on user input.

2 Review

Some of the new concepts we'll cover require familiarity with concepts we've touched on previously. These concepts will also be useful in your projects.

2.1 References

References are perfectly valid types, just like pointers. For instance, just like `int *` is the "pointer to an integer" type, `int &` is the "reference to an integer" type. References can be passed as arguments to functions, returned from functions, and otherwise manipulated just like any other type.

References are just pointers internally; when you declare a reference variable, a pointer to the value being referenced is created, and it's just dereferenced each time the reference variable is used.

The syntax for setting a reference variable to become an alias for another variable is just like regular assignment:

```
int &x = y; // x and y are now two names for the same variable
```

Similarly, when we want to pass arguments to a function using references, we just call the function with the arguments as usual, and put the `&` in the function definiton, where the argument variables are being set to the arguments actually passed:

```
void sq(int &x) { // & is part of the type of x
                  // - x is an int reference
  x *= x;
}
sq(y);
```

Note that on the last line, where we specify what variable `x` will be a reference to, we just write the name of that variable; we don't need to take an address with `&` here.

References can also be returned from functions, as in this contrived example:

```
int g; // Global variable
int &getG() { // Return type is int reference
  return g; // As before, the value we're making a
            // reference *to* doesn't get an & in front of it
}

    // ...Somewhere in main
    int &gRef = getG(); // gRef is now an alias for g
    gRef = 7; // Modifies g
```

If you're writing a class method that needs to return some internal object, it's often best to return it by reference, since that avoids copying over the entire object. You could also then use your method to do something like:

```
vector<Card> &cardList
```

```
  = deck.getList(); // getList declared to return a reference
cardList.pop_back();
```

The second line here modifies the original list in deck, because cardList was declared as a reference and getList returns a reference.

2.2 const

2.2.1 Converting between const and non-const

You can always provide a non-const value where a const one was expected. For instance, you can pass non-const variables to a function that takes a const argument. The const-ness of the argument just means the function promises not to change it, whether or not you require that promise. The other direction can be a problem: you cannot provide a const reference or pointer where a non-const one was expected. Setting a non-const pointer/reference to a const one would violate the latter's requirement that it not be changeable. The following, for instance, does not work:

```
int g; // Global variable
const int &getG() { return g; }

    // ...Somewhere in main
    int &gRef = getG();
```

This fails because gRef is a non-const reference, yet we are trying to set it to a const reference (the reference returned by getG).

In short, the compiler will not let you convert a const value into a non-const value unless you're just making a copy (which leaves the original const value safe).

2.2.2 const functions

For simple values like ints, the concept of const variables is simple: a const int can't be modified. It gets a little more complicated when we start talking about const objects. Clearly, no fields on a const object should be modifiable, but what methods should be available? It turns out that the compiler can't always tell for itself which methods are safe to call on const objects, so it assumes by default that none are. To signal that a method is safe to call on a const object, you must put the const keyword at the end of its signature, e.g. int getX() const;. const methods that return pointers/references to internal class data should always return const pointers/references.

3 Exceptions

Sometimes functions encounter errors that make it impossible to continue normally. For instance, a `getFirst` function that was called on an empty `Array` object would have no reasonable course of action, since there is no first element to return.

A functions can signal such an error to its caller by *throwing* an *exception*. This causes the function to exit immediately with no return value. The calling function has the opportunity to *catch* the exception – to specify how it should be handled. If it does not do so, it exits immediately as well, the exception passes up to the next function, and so on up the *call stack* (the chain of function calls that got us to the exception).

An example:

```
1  const int DIV_BY_0 = 0;
2  int divide(const int x, const int y) {
3      if(y == 0)
4          throw DIV_BY_0;
5      return x / y;
6  }
7
8  void f(int x, int **arrPtr) {
9    try {
10     *arrPtr = new int[divide(5, x)];
11   } catch(int error) {
12     // cerr is like cout but for error messages
13     cerr << "Caught error: " << error;
14   }
15   // ... Some other code...
16 }
```

The code in `main` is executing a function (`divide`) that might throw an exception. In anticipation, the potentially problematic code is wrapped in a `try` block. If an exception is thrown from `divide`, `divide` immediately exits, passing control back to `main`. Next, the exception's type is checked against the type specified in the `catch` block (line 11). If it matches (as it does in this case), the code in the `catch` block is executed; otherwise, `f` will exit as well as though it had thrown the exception. The exception will then be passed up to `f`'s caller to see if it has a `catch` block defined for the exception's type.

You can have an arbitrary number of `catch` blocks after a `try` block:

```
int divide(const int x, const int y) {
    if(y == 0)
        throw std::runtime_exception("Divide by 0!");
    return x / y;
```

```
 5 }
 6
 7 void f(int x, int **arrPtr) {
 8   try {
 9     *arrPtr = new int[divide(5, x)];
10   }
11   catch(bad_alloc &error) {//new throws exceptions of this type
12     cerr << "new failed to allocate memory";
13   }
14   catch(runtime_exception &error) {
15     // cerr is like cout but for error messages
16     cerr << "Caught error: " << error.what();
17   }
18   // ...
19 }
```

In such a case, the exception's type is checked against each of the catch blocks' argument types in the order specified. If line 2 causes an exception, the program will first check whether the exception is a `bad_alloc` object. Failing that, it checks whether it was a `runtime_exception` object. If the exception is neither, the function exits and the exception continues propagating up the call stack.

The destructors of all local variables in a function are called before the function exits due to an exception.

Exception usage notes:

- Though C++ allows us to throw values of any type, typically we throw exception objects. Most exception classes inherit from class `std::exception` in header file `<stdexcept>`.
- The standard exception classes all have a constructor taking a string that describes the problem. That description can be accessed by calling the `what` method on an exception object.
- You should always use references when specifying the type a `catch` block should match (as in lines 11 and 14). This prevents excessive copying and allows virtual functions to be executed properly on the exception object.

4 `friend` Functions/Classes

Occasionally you'll want to allow a function that is not a member of a given class to access the private fields/methods of that class. (This is particularly common in operator overloading.)

We can specify that a given external function gets full access rights by placing the signature of the function inside the class, preceded by the word `friend`. For example, if we want to make the fields of the `USCurrency` type from the previous lecture private, we can still have our *stream insertion operator* (the output operator, `<<`) overloaded:

```
class USCurrency {
    friend ostream &operator<<(ostream &o, const USCurrency &c);
    int dollars, cents;
public:
    USCurrency(const int d, const int c) : dollars(d), cents(c) {}
};

ostream &operator<<(ostream &o, const USCurrency &c) {
    o << '$' << c.dollars << '.' << c.cents;
    return o;
}
```

Now the `operator<<` function has full access to all members of `USCurrency` objects.

We can do the same with classes. To say that all member functions of class `A` should be fully available to class `B`, we'd write:

```
class A {
  friend class B;
  // More code...
};
```

5 Preprocessor Macros

We've seen how to define constants using the preprocessor command `#define`. We can also define *macros*, small snippets of code that depend on arguments. For instance, we can write:

```
#define sum(x, y) (x + y)
```

Now, every time `sum(a, b)` appears in the code, for any arguments `a` and `b`, it will be replaced with `(a + b)`.

Macros are like small functions that are not type-checked; they are implemented by simple textual substitution. Because they are not type-checked, they are considered less robust than functions.

6 Casting

Casting is the process of converting a value between types. We've already seen *C-style casts* – e.g. `1/(double)4`. Such casts are not recommended in C++; C++ provides a number of more robust means of casting that allow you to specify more precisely what you want.

All C++-style casts are of the form `cast_type<type>(value)`, where `type` is the type you're casting to. The possible cast types to replace `cast_type` with are:

- `static_cast` – This is by far the most commonly used cast. It creates a simple copy of the value of the specified type. Example: `static_cast<float>(x)`, where `x` is an `int`, gives a `float` copy of `x`.
- `dynamic_cast` – Allows converting between pointer/reference types within an inheritance hierarchy. `dynamic_cast` checks whether `value` is actually of type `type`. For instance, we could cast a `Vehicle *` called `v` to a `Car *` by writing `dynamic_cast<Car *>(v)`. If `v` is in fact a pointer to a `Car`, not a `Vehicle` of some other type such as `Truck`, this returns a valid pointer of type `Car *`. If `v` does not point to a `Car`, it returns null.

  `dynamic_cast` can also be used with references: if `v` is a `Vehicle &` variable, `dynamic_cast <Car &>(v)` will return a valid reference if `v` is actually a reference to a `Car`, and will throw a `bad_cast` exception otherwise.
- `reinterpret_cast` – Does no conversion; just treats the memory containing `value` as though it were of type `type`
- `const_cast` – Used for changing `const` modifiers of a value. You can use this to tell the compiler that you really do know what you're doing and should be allowed to modify a const variable. You could also use it to add a `const` modifier to an object so you can force use of the `const` version of a member function.

7 That's All!

This is the end of the 6.096 syllabus, but there are lots of really neat things you can do with C++ that we haven't even touched on. Just to give you a taste:

- Unions – group multiple data types together; unlike classes/structs, though, the fields of a union are mutually exclusive – only one of them is well-defined at any time
- Namespaces – allow you to wrap up all your code, classes, etc. into a "directory" of names, like `std`
- Advanced STL manipulation – allows you to do all sorts of wacky things with STL containers and iterators

- `void` pointers – pointers to data of an unknown type
- `virtual` inheritance – the solution to the “dreaded diamond” problem described in Lecture 8
- String streams – allow you to input from and output to string objects as though they were streams like `cin` and `cout`
- Run-time type information (RTTI) – allows you to get information on the type of a variable at runtime
- vtables – how the magic of virtual functions actually works

If you are interested in learning more about these subjects or anything we’ve discussed, we encourage you to look through online tutorials, perhaps even to buy a C++ book – and most importantly, to just play around with C++ on your own!