UPE Tutoring:

CS 31 Final Review

Sign-in https://goo.gl/K6QsCt

Slides link available upon sign-in

Table of Contents

Libraries

- `#include` allows us to use a library
- `#include <iostream>` allows us to use things like:
  - `cin`
  - `cout`
  - `endl`
- *Note: `iostream` stands for input/output stream*

Modifying variables

- The type of the variable must be specified only once, at the time of declaration

```
int x = 5;
x = x + 5;
x -= 6; // equivalent to x = x - 6;

double z;
z = 53.234;
z *= 5; // equivalent to z = z * 5;
```

Modifying variables (cont.)

- Integer division truncates after the decimal point
- The % (modulus) operator returns the remainder of integer division

```
int x = 5;
int integerQuotient = x / 3; // integerQuotient equals 1
int remainder = x % 3;       // remainder equals 2
x %= 4;                      // same as x = x % 4, x now equals 1
```

Modifying variables (cont.)

- Double division
  - If <u>at least one</u> of the operands is a double, floating point division occurs.
  - If both values are integers, integer division occurs instead.

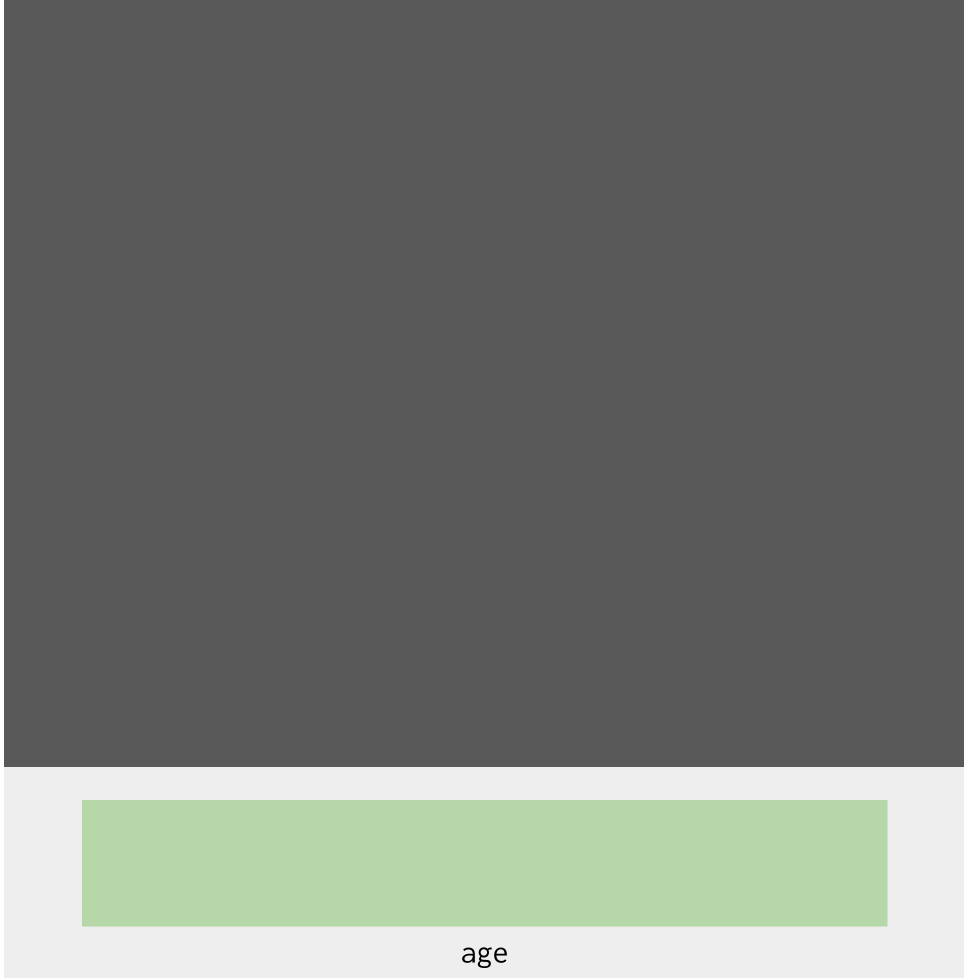
```
int x = 5;
double unexpectedQuotient = x / 2;     // equals 2.0
double expectedQuotient = x / 2.0;     // equals 2.5
```

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```



Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

```
> How old are you?
```

age

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

```
> How old are you? 20
```

20

age

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

```
> How old are you? 20
> You are 20 years old
```

20

age

Strings

- Used to store blocks of text
- Strings can be initialized from literals
  - `string x = “hello”;`
- Individual characters can be accessed with the `[ ]` operator.
  - `char c = x[0]; // c == ‘h’`

String operations

```
string x = "hello there";
```

- The `size()` method returns the number of characters in a string.
  - `int length = x.size(); // length equals 11`
- The `substr(startIndex, length)` method returns a substring *including* `startIndex` of length `length`.
  - `string sub = x.substr(3, 2); // sub equals "lo"`
- Concatenation adds strings together.
  - `string x = “hello”; x += “ there”; // x is now “hello there’`
- The getline(...) method reads in a string up to the first newline character and saves it to a string.
  - `string x; getline(cin, x); // chars up to a ‘\n’ saved into x`

Ignoring characters

- Undesirable characters are often left in the input buffer after using cin.
- `cin.ignore(int numChars, char delim)` can be used to “flush” out these undesired characters. It flushes up to the nearest `delim` or `numChar` characters, whichever comes first.
- `cin.ignore(...)` becomes necessary if after reading a number, the next thing you want to read is a string using `getline(...)`.

Ignoring characters

- Undesirable characters are often left in the input buffer after using cin.
- `cin.ignore(int numChars, char delim)` can be used to "flush" out these undesired characters. It flushes up to the nearest `delim` or `numChar` characters, whichever comes first.
- `cin.ignore(...)` becomes necessary if after reading a number, the next thing you want to read is a string using `getline(...)`.

- **Common question**: does `getline` consume `'\n'`?

Ignoring characters

- Undesirable characters are often left in the input buffer after using cin.
- `cin.ignore(int numChars, char delim)` can be used to “flush” out these undesired characters. It flushes up to the nearest `delim` or `numChar` characters, whichever comes first.
- `cin.ignore(10000, ‘\n’)` becomes necessary if after reading a number, the next thing you want to read is a string using `getline(...)`.

- **Common question**: does `getline` consume `'\n'`?

  If a newline is found, it is extracted and discarded (i.e. it is not stored and the next input operation will begin after it).

cin.ignore(...) example

- What will be stored in the string “a” in this example?
- Assume that input is newline terminated.

```
int x; string a;
cout << "Enter an integer" << endl;
cin >> x; // Assume the user enters “7”
cout << "Enter a string" << endl;
getline(cin, a); // Assume user enters “500”
```

cctype

- Useful shortcut methods for characters
- #include<cctype> gives you...
    - `isalpha('M')  // true, since 'M' is a letter`
    - `isupper('M')  // true, since 'M' is an uppercase letter`
    - `islower('r')  // true, since 'r' is a lowercase letter`
    - `isdigit('5')  // true, since '5' is a digit character`
    - `islower('M')  // false, since 'M' is not a lowercase letter`
    - `isalpha(' ')  // false, since ' ' is not a letter`
    - `isalpha('5')  // false, since '5' is not a letter`

If statements

- `if` statements only run code if the condition is true
- *Note: any non-zero expression is considered true*

```
int age;
cin >> age;
if (age < 13) {
  cout << "You are not yet a teenager!" << endl;
}
```

If statements

- Without curly braces, only the next statement is attached to the control statement.
- So, if you want multiple statements to be executed based on a condition, use curly braces.
- *Note: this also applies to `else` and `else-if` statements*

```
if (cond1) {
  statement1;
  statement2;
}
```

If statements (cont.)

```
int main() {
  int x = 3;
  if (x == 5)
    cout << "x is 5" << endl;
  cout << "In if" << endl; // Incorrect!
}
```

```
int main() {
  int x = 5;
  if (x == 5) {
    cout << "x is 5" << endl;
    cout << "In if" << endl;
  }
}
```

Else statements

- Performed when all `if` and `else if` conditions fail

```
int number;
cin >> number;
if (number % 2 == 0)
    cout << "You gave an even number" << endl;
else
    cout << "You gave an odd number" << endl;
```

Else-if statements

- Allows us to check for more than the `if` condition and its complement

```
if (cond1)
    statement1;
else if (cond2)   // executes if (!cond1 && cond2)
    statement2;
else if (cond3)   // executes if (!cond1 && !cond2 && cond3)
    statement3;
else
    statement4;
```

Comparison pitfalls

- **Equals-equals (==) vs. Equals (=)**
- These operators are very different!

```
(x == y) // Returns true if x and y are equal

(x = y)  // Assigns the value of y to x and returns the value
         // ASSIGNED to x.
```

Conditional confusion?

- Does this output anything?

```
int age = 17;
if (age) {
  cout << "You are not 0 years old!" << endl;
}
```

Conditional confusion?

- What does this output?

```
int age = 0;
if (age) {
  cout << "You are not 0 years old!" << endl;
} else {
  cout << “You are 0 years old!” << endl;
}
```

Switches

- Arguably a more compact alternative to long `if/else if/else` sequences
- The value tested must be an integral type or convertible to one
  - e.g. `int`, `char`, `short`, `long`, etc.
  - `string` is not a permitted type
- A `break` statement must be used to leave the switch. Otherwise execution will fall through to the next case.

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
    break;
}
```

Common question 1: is a break statement required for the default case?

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
    break;
}
```

Common question 1: is a `break` statement required for the `default` case?

No, if the default case is at the end. However, we recommend that you add one anyway. This allows the `default` case to appear in a different order, and not necessarily at the end of the switch statement.

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
    break;
}
```

Common question 1: is a `break` statement required for the `default` case?

No if the default case is at the end. However, we recommend that you put one anyways. This allows the `default` case to appear in a different order, and not necessarily at the end of the switch statement.

Common question 2: do I need a default statement?

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
    break;
}
```

Common question 1: is a `break` statement required for the `default` case?

No if the default case is at the end. However, we recommend that you put one anyways. This allows the `default` case to appear in a different order, and not necessarily at the end of the switch statement.

Common question 2: do I need a default statement?

No, but it is good to have to catch unexpected cases. You should leave a `//comment` if you don't have a `default` to explain why!

While loops

- `while` loops run code over and over until their condition is false

```
int count;
cin >> count;
while (count >= 0) {
  cout << "Countdown: " << count << endl;
  count--;
}
```

Do-while loops

- Same as while loops, except the first iteration always runs.

```
statement1;
do {
  statement2;
} while (cond1); // Don't forget the semicolon!

statement3;
```

For loops

- Declaration is run <u>once</u> before anything else
- Condition is evaluated <u>before</u> the code block is executed
- Action is run <u>after</u> the code block is executed

```
for (declaration; condition; action) {
  statement1;
  statement2;
}
```

Note: all of the three sections of the for loop are optional; all that is required is the semicolon. If `condition` is empty, it defaults to always `true`. Example: `for(int i = 0;;i++) { //infinite loop }`

Nested loops

```
for (int i = 1; i <= 10; i++) {
  for (int j = 1; j <= 10; j++)  {
    cout << (i * j) << "\t";
  }
  cout << endl;
}
```

Nested loops

```
for (int i = 1; i <= 10; i++) {
  for (int j = 1; j <= 10; j++)  {
    cout << (i * j) << "\t";
  }
  cout << endl;
}
```

```
// prints:
// 1   2   3   4   5    6    7    8    9    10
// 2   4   6   8   10   12   14   16   18   20
// ...
// 10  20  30  40  50   60   70   80   90  100
```

Quick Question - Breaking the Outer Loop

- What happens when you `break` inside nested loops?

Quick Question - Breaking the Outer Loop

- What happens when you `break` inside nested loops?
  - Only the loop that contains the `break` statement is broken out of.

Quick Question - Breaking the Outer Loop

- What happens when you break inside nested loops?
  - Only the loop that contains the break statement is broken out of.
- Solution: use a boolean variable (a *flag*) in your loop's statements and change the boolean from true to false when you want to break out of a nested loop.

```
bool keepLoopingI = true;
for (int i = 1; i <= 100; i++) {
     for (int j = 1; j <= 200; j++) {
          if (i+j > 250) keepLoopingI = false;
          cout << i << “,” << j << endl;
     }
     if (!keepLoopingI) break;
}
```

Scoping

- Variables only exist within the curly brackets or the implied curly brackets that they were written in.

```
if (cond1) {
  statement1;
}
```

Scoping (cont.)

```
if (cond1) {
  int x = 5;
  cout << x << endl; // No error
}

cout << x << endl; // Error!! x doesn't exist
                   // outside the if statement
```

Scoping (cont.)

```
int x = 1;
if (cond1) {
  x = 5;
  cout << x << endl; // No error
}

cout << x << endl; // No error here either!
```

Scoping (cont.)

```
string s1 = “bonjour”;
for (int i = 0; i < s1.size(); i++) {
  char lastChar = s1[i];
}

// Both i and lastChar don't exist here!
cout << i << " " << lastChar << endl; // Error!
```

Scoping (cont.)

```
string s1 = “bonjour”;
int i; char lastChar;
for (i = 0; i < s1.size(); i++) {
  lastChar = s1[i];
}

// Now both i and lastChar exist here
cout << i << " " << lastChar << endl;
```

Scoping: Switch Statements

```
int main() {
  int n;
  cin >> n;
  switch (n) {
    case 1:
      int x = 10;
      cout << "You entered 1! 1 times 10 is " << x << endl;
      break;
    default:
      int x = 5;
      cout << "You didn't enter 1" << endl;
  }
}
```

Scoping: Switch Statements

Warning: the cases of a `switch` statement share the same scope!

```
int main() {
  int n;
  cin >> n;
  switch (n) {
    case 1:
      int x = 10;
      cout << "You entered 1! 1 times 10 is " << x << endl;
      break;
    default:
      int x = 5;  // This is an error! Compiler says “error: redefinition of 'x'”
      cout << "You didn't enter 1" << endl;
  }
}
```

Scoping: Switch Statements

Warning: the cases of a `switch` statement share the same scope!

```
int main() {
  int n;
  cin >> n;
  switch (n) {
    case 1:
      int x = 10;
      cout << "You entered 1! 1 times 10 is " << x << endl;
      break;
    default:
      // Does x exist here? It's within the switch's curly braces, but "int x = 10" was never executed?!
      cout << "You didn't enter 1" << endl;
  }
}
```

Scoping: Switch Statements

Warning: the cases of a `switch` statement share the same scope!

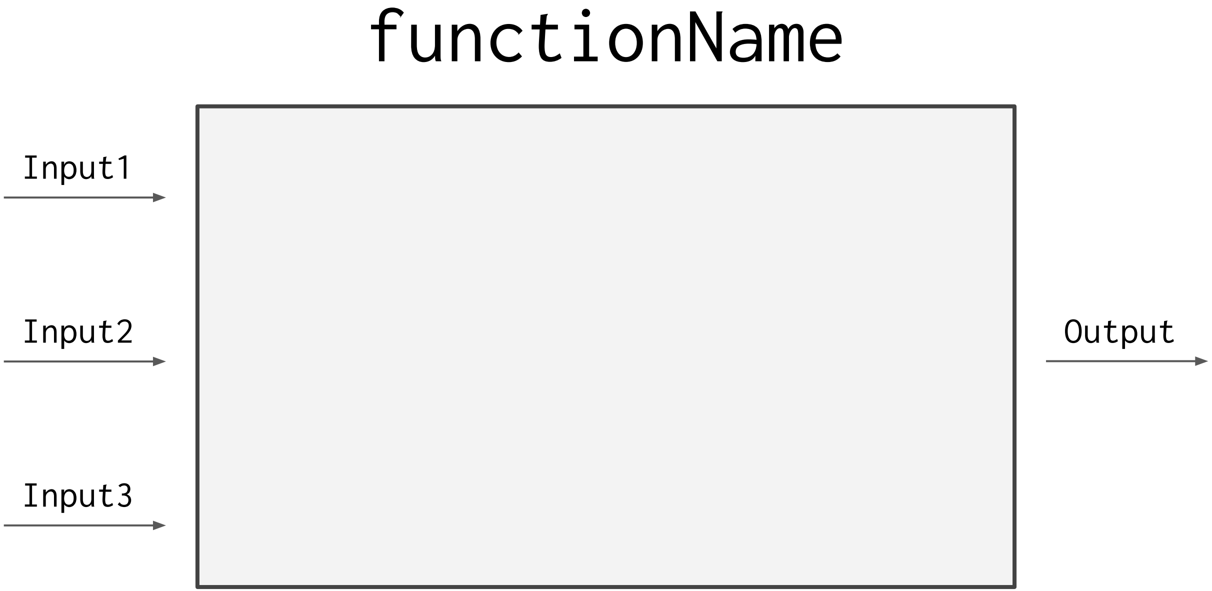
```
int main() {
  int n;
  cin >> n;
  switch (n) {
    case 1:
      int x = 10;
      cout << "You entered 1! 1 times 10 is " << x << endl;
      break;
    default:
      // So, this is also an error! Compiler says "note: jump bypasses variable initialization"
      cout << "You didn't enter 1" << endl;
  }
}
```

Scoping: Switch Statements

Warning: the cases of a `switch` statement share the same scope!

```
int main() {
  int n;
  cin >> n;
  switch (n) {
    case 1: {
      int x = 10; // Now x is only known to the scope of this case
      cout << "You entered 1! 1 times 10 is " << x << endl;
      break;
    }
    default:
      cout << "You didn't enter 1" << endl;
  }
}
```

Functions: The Box Model



Functions: Simple Example

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

int hypotenuse(int side1, int side2) {
     /* Function body */
     /* We don’t need to know how the function is implemented */
     ...
}

int main() {
  int x = hypotenuse(3,4);
}
```

Functions: Scoping

- Variables declared outside the function do not exist inside the function unless they are global variables

```
const int foo = 6;
string functionName(int a, int b) {
  cout << x << endl; // x does not exist here, so this is an ERROR.
  int y = 5 + foo;   // This is okay because foo is global.
  return y + a + b;
}
```

Functions: Scoping (cont.)

Variables declared inside the function do not exist outside the function

```
int main() {
  int x = 4;
  cout << functionName(4,5) << endl;
  cout << y << endl; // y does not exist here, so this is an error!
}
```

Functions: Parameters

- The types, modifiers, order, and number of parameters are all important in a function declaration
  - types: `string, int, bool`, etc.
  - modifiers: `&, const, *`, etc.
  - number: how many parameters are passed to a particular function?

- Function call **must** match the pattern of the declaration

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  string x = isEqual(s1,s2, position);
}
```

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  string x = isEqual(s1,s2, position);
}
```

No. The return type is boolean.

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(s1,s2);
}
```

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(s1,s2);
}
```

No. The number of arguments doesn't match.

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(position, s1, s2);
}
```

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(position, s1, s2);
}
```

No. The order of arguments doesn't match.

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(s1, s2, position);
}
```

Will this compile?

```
// Assume this function is defined later.
bool isEqual(string s1, string s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(s1, s2, position);
}
```

Yes.

Will this compile?

```
// Assume this function is defined later.
bool isEqual(const string& s1, const string& s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(s1, s2, position);
}
```

Will this compile?

```
// Assume this function is defined later.
bool isEqual(const string& s1, const string& s2, int position);

int main() {
  string s1 = "hello";
  string s2 = "there";
  int position = 1;
  bool x = isEqual(s1, s2, position);
}
```

Yes. Notice the argument passing syntax is identical for pass by reference.

Functions: Pass by Value

- By default, all parameters in C++ are pass by value.
- Every pass by value parameter is **copied** into the function

```
bool containsLowerCase(string s);

int main() {
  string s1 = "really long string";
  containsLowerCase(s1);     // a copy of s1 is made and
}                            // passed to containsLowerCase
```

Functions: Pass by Reference

- A **reference** to a variable is passed to the function instead of a copy of the variable
- Syntax: add an & between parameter type and name
  - `int& x, bool& b, string& s`
- If these variables are **changed inside** the function, then they will also be **changed outside.**

Functions: swap

- Does this function properly swap the two variables passed to it?

```
void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}
```

Functions: swap

- Does this function properly swap the two variables passed to it?

```
void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}
```

No, it only swaps local copies! We need to use pass by reference.

Functions: swap #2

- Does this function properly swap the two variables passed to it?

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}
```

Functions: swap #2

- Does this function properly swap the two variables passed to it?

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}
```

Yes, because we are using the & modifier on the parameters to pass by reference.

Functions: Const Variables

- A parameter with the `const` modifier **cannot be modified** within the function. For example, we cannot change the value of `num` from within the body of the function `isPrime`.

```
bool isPrime(const int& num) {
  // Cannot change value of num here.
  ...
}
```

Functions: Const Variables

- Why are they useful?
  - Gives assurance the the caller of a function that the argument they pass in won’t be modified
  - Makes convoluted functions easier to understand if we know a certain variable can’t be modified

- These are usually **passed by reference.**

Functions: Passing by Constant Reference

- The purpose of passing by reference is to save memory or allow modifications by the function.
- What if we want to avoid copying but don't want to allow functions to modify the variables we pass in?

Functions: Passing by Constant Reference

- If we pass by **const reference** we can:
  - avoid the cost of copying
  - prevent our variables from being modified by the function
- Essentially a free performance gain
- You'll run into const reference often in CS32

Arrays

- Valid declarations:

```
int arr[10];

bool list[5];

const int MAX_SIZE = 10;
string words[MAX_SIZE];

int arr[] = {1, 2, 3};
```

Arrays (cont.)

- Rules for specifying size:
  - **Must** be included in the brackets
  - **Cannot** involve a variable unless it is a constant known at compile time
  - The only time size can be left out is when a list of its contents is included
- Not allowed in C++:
  - `int arr[]; // Size not included.`
  - ```
    /****** Use of non-const variable. ******/
    int x;
    cin >> x;
    char buffer[x];
    ```
    ```

Passing Arrays to Functions

- Parameter Syntax
  - `(..., type name[], ...)`
- Arrays are default passed by reference
  - Any changes made to the array will be retained outside of the function scope

Passing Arrays to Functions (cont.)

- Size of array should be passed to the function
- Call to the function just passes in array name

```
// arr is the array itself, n is the size.
int firstOdd(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    if (arr[i] % 2 == 1)
      return i;
  }
  return n; // If no odd number found.
}
```

Printing Arrays

- To print an array, we need to use a loop to print each element.
- Printing the name will just print the starting address of the array

```
string arr[] = {“Smallberg”, “CS31”, “Midterm”};
for (int i = 0; i < 3; ++i) {
  cout << arr[i];
}
```

Out of Bounds Errors

- Occur anytime you can access memory past the end (or beginning) of an array
  - Only certain spaces in memory have useful data
  - Anything outside is essentially garbage
  - Hard to debug. C++ doesn't do bounds checking on array access so out of bounds accesses can often go unnoticed.

```
string array[3] = {“CS31”, “Smallberg”, “Midterm”};
cout << array[3] << endl; // Out of bounds error!
```

Out of Bounds Example

Do we have an out of bounds memory access here?

```
// Assume arr only contains n elements.
int countFives(int arr[], int n) {
  int count = 0;
  for (int i = 0; i <= n; ++i) {
    if (arr[i] == 5) {
      count++;
    }
  }
  return count;
}
```

Out of Bounds Example

Do we have an out of bounds memory access here?

```
// Assume arr only contains n elements
int countFives(int arr[], int n) {
  int count = 0;
  for (int i = 0; i <= n; ++i) {
    if (arr[i] == 5) {
      count++;
    }
  }
  return count;
}
```

Yes! The for loop will access the element at the **nth** index.

C Strings

- C does not have the string class *(or classes at all!)*
- In C, we cannot declare strings or use class methods:
  - `string x = “hello”;`
    - `x.size()` **`// This is okay in C++, but not in C.`**
- Instead, we represent strings using char arrays:
  - `char y[] = “hello”;`
  - Cannot use C++ string functions with it
    - `y.size(), y.substr(...), etc.` **`// Syntax errors.`**
  - **`#include <cstring>`** provides functions like **`strlen`**
    - **`strlen(x)`** returns 5

Ascii: Characters are actually integers

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Ascii (cont.)

```
int x = ‘G’; // x is now 71.
x -= 1;      // x is now 70.
char y = x;  // y is now F.
int z = ‘5’; // z is now 53.
```

C Strings (cont.)

- The end of a C string is marked by a null byte (**‘\0’**)
  - Null byte has ASCII value `0`
  - **`strlen`** simply looks for the null byte for you

```
char arr[] = “hello”;
for (int i = 0; arr[i] != ‘\0’; i++) // Standard for loop to iterate
                                      // through c-strings.
```

Note: `arr[i] != ‘\0’` and `arr[i] != 0` are the same, as ascii value of **‘\0’** is **0**.

C Strings (cont.)

```
// A null character is automatically
// put in index 5.
char x[50] = “hello”;

// Because we have more space in the array
// (50 total), we can add more characters.
x[5] = ‘s’;
x[6] = ‘\0’;
```

```
[‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’, ...]
```

x

C Strings (cont.)

```
// A null character is automatically
// put in index 5.
char x[50] = “hello”;

// Because we have more space in the array
// (50 total), we can add more characters.
x[5] = ‘s’;
x[6] = ‘\0’;
```

```
[‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘s’, ...]
```

x

C Strings (cont.)

```
// A null character is automatically
// put in index 5.
char x[50] = “hello”;

// Because we have more space in the array
// (50 total), we can add more characters.
x[5] = ‘s’;
x[6] = ‘\0’;
```

```
[‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘s’, ‘\0’, ...]
```

x

Pointers

- A **pointer** is the memory address of a variable.

- The **&** operator can be used to determine the **address** of a variable to be stored in the pointer.
- The ***** operator can be used to **dereference** a pointer and get the value stored in the variable that is being pointed to.

```
double d = 10.0;
double *dp;          // pointer that holds the address of a double
dp = &d;             // stores the address of d into dp
*dp = 20.0;          // changes the value of d from 10.0 to 20.0
```

Pointers

```
int var = 20;    // actual variable declaration
int *ip;         // pointer variable declaration

// store address of var in ip
ip = &var;

cout << "Value of var variable: ";
cout << var << endl;

// print the address stored in ip pointer
cout << "Address stored in ip variable: ";
cout << ip << endl;

// access the value at address stored in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;
```

```
> Value of var variable: 20
> Address stored in ip variable: 0xBFC601AC
> Value of *ip variable: 20
```

Pointer Arithmetic

```
int arr[] = {10, 20, 30, 40};

int *ptr = arr;               // arr == &arr[0]
cout << *ptr << endl;

ptr++;
cout << *ptr << endl;

ptr = ptr + 1;
cout << *ptr << endl;

ptr--;
cout << *(ptr + 2) << endl;
```

```
> 10
> 20
> 30
> 40
```

Pointers – new and delete

- The **`new`** operator can be used to create **dynamic** variables. These variables can be accessed using pointers.

```
string *p;
p = new string;
p = new string(“hello”);
```

- The **`delete`** operator eliminates dynamic variables.

```
delete p;
```

- Note: Pointer `p` is now a **dangling pointer**! Dereferencing it is dangerous and leads to undefined behavior. One way to avoid this is to set `p` to `NULL` after using `delete`.

Pointers – the Heap and the Stack

- As it turns out, there are **two** places where your variables live.
- The first is the **stack**, which is the place you're most familiar with. With **local variables**, the compiler is like a city planner who decides where each variable should live.

```
void foo() {
  int a[4];   // Stored at 100
  int k;      // Stored at 116
  string s;   // Stored at 120
}
```

When foo called →

120	string s
116	int k
100	int a[4]
0-100	Variables in the calling function.

If the size isn't specified at compile time, how would the compiler know where to put k or s?

Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The first is the **stack**, which is the place you're most familiar with. With **local variables**, the compiler is like a city planner who decides where each variable should live.

```
void foo() {
  int a[4];   // Stored at 100
  int k;      // Stored at 116
  string s;   // Stored at 120
}
```

When foo returns →

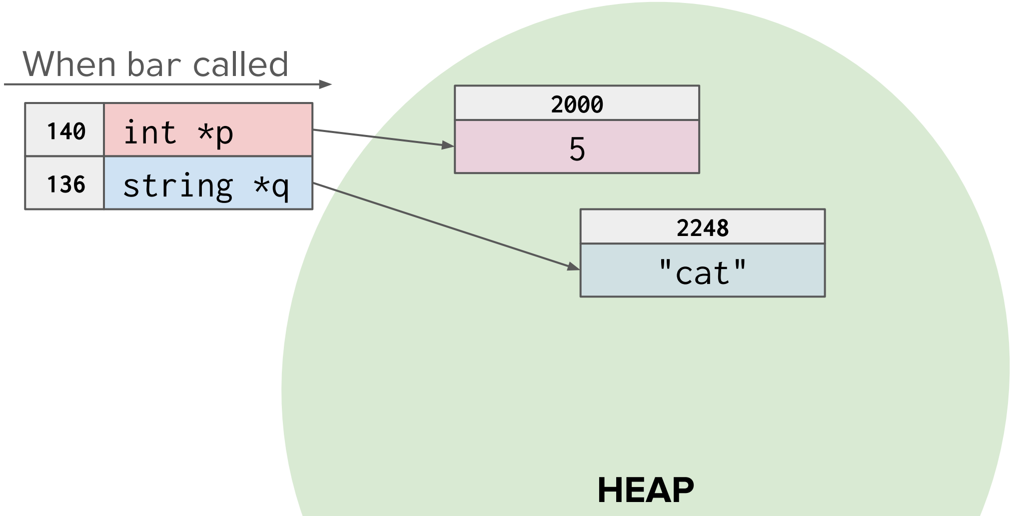
120	Vacant
116	Vacant
100	Vacant
0-100	Variables in the calling function.

When the function returns, the variables are evicted from their addresses.

Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The second is the **heap**, which is the place where dynamic variables live. Dynamic variables essentially lease some part of the heap to live in.

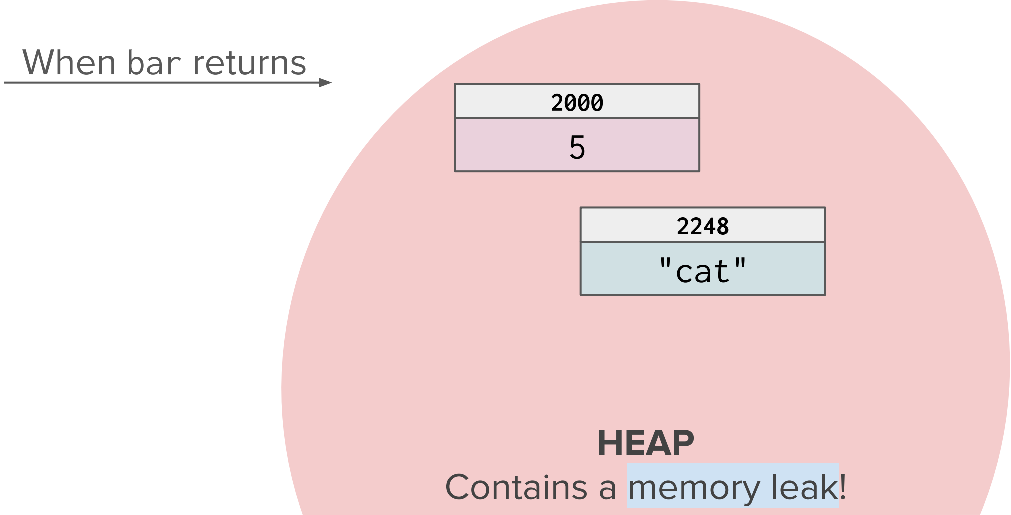
```
void bar() {
  int *p = new int;
  *p = 5;
  string *q = new string("Cat");
}
```



Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The second is the **heap**, which is the place where dynamic variables live. Dynamic variables essentially lease some part of the heap to live in.

```
void bar() {
  int *p = new int;
  *p = 5;
  string *q = new string("Cat");
}
```



Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The second is the **heap**, which is the place where dynamic variables live. Dynamic variables essentially lease some part of the heap to live in.

```
void bar() {
  int *p = new int;
  *p = 5;
  string *q = new string("Cat");
  delete p;
  delete q;
}
```

When bar returns

Practice Question: Array Traversal w/ Pointers

Write a function that sums the items of an array of `n` integers using only pointers to traverse the array.

Solution: Array Traversal w/ Pointers

Simple array traversal, but with pointers. To get to item `i`, add `i` to your head pointer then dereference. This works because the compiler knows the size of an item in your array in C++. Sum values as you go by adding them to a `total` value created outside of the for loop.

```
int sum(int *head, int n) {
  int total = 0;
  for (int i = 0; i < n; i++) {
    total += *(head + i);
  }
  return total;
}
```

```
sum(arr, 5);
```

Structs

- A **struct** is a collection of data that is treated as its own special data type. We use them to organize data that belongs together.

```
struct Person {
  int age;
  string name;
};   // Must end with semicolon!
```

- Structs can store any number of any other data type, accessed using `.` (the **dot operator**). These stored values are called **member variables**.

Structs

You can declare a struct outside of any functions and treat it as a normal variable, for the most part. A struct can even contain another struct!

```
struct Date {
  int day, month, year;
};

struct Person {
  Date birthday;
  string name;
  double money;
};
```

```
void doubleMoney(Person& guy) {
  guy.money *= 2;
}

int main() {
  Person p1;
  p1.name = "Smelborp";
  p1.money = 3.50;
  p1.birthday.day = p1.birthday.month = 1;
  Person p2 = p1;  // Perfectly legal
  doubleMoney(p2);
  cout << p2.money; // 7
  cout << p1.money; // 3.5
  Person p3 = { p1.birthday, "Jimbo", 3.5 };
  p2 += p1;  // ERROR! How do you add people?!
}
```

Structs – Tips

- Structs are a good way to keep code looking organized and readable
- When declaring a struct, member variables with primitive types will be left uninitialized. Classes will be constructed with their default constructor.
  - Long story short: you must assign them yourself!
- **Always remember the semicolon!** It's there so that you can declare struct variables at the same time that you define the struct.

```
struct Circle {
  int radius;
} ring, hoop;    // This creates two Circle structs named ring and hoop
```

Classes

- A **class** is like a struct, but with **member functions** as well as member variables. Classes are at the core of Object-oriented Programming (OOP).

```
class Person {
  int age;
  string name;          // Strings are actually objects from the string class!
  double money;
public:
  void doubleMoney();   // Member function, declared but not yet implemented.
};
```

- We call an instance of a class an **object**. In this way, OOP involves different types of objects interacting with each other.

Classes – Member Functions

Now, let's fill in the `doubleMoney` function of the `Person` class from the previous slide. Two ways:

```
// 1. Inside the class definition.
class Person {
  int age;
  string name;
  double money;
public:
  void doubleMoney() { money *= 2; }
};

// 2. Outside of the class definition.
void Person::doubleMoney() {
  money *= 2;
}
```

The **scope resolution operator** (e.g. `Person::`) tells the compiler that we are defining the `doubleMoney` function of the `Person` class.

Note: we don't need the dot operator to refer to `Person`'s `money` variable when we are in one of `Person`'s member functions!

Classes – Access Specifiers

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;
  bobby.name = "Bobby";
  bobby.money = 3.49;
  bobby.doubleMoney();
}
```

Classes – Access Specifiers (cont.)

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;   //ERROR!
  bobby.name = "Bobby";   //ERROR!
  bobby.money = 3.49;   //ERROR!
  bobby.doubleMoney();   //ERROR!
}
```

Classes – Access Specifiers (cont.)

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;   // Good to go :^)
  bobby.name = "Bobby";
  bobby.money = 3.49;
  bobby.doubleMoney();
}
```

The compiler doesn't let us access any of bobby's members! This is because class members have the **private** access specifier by default and cannot be accessed by an outside class or function. To fix this, we adjust our class:

```
class Person {
public:
  int age;
  string name;
  double money;
  void doubleMoney() { money *= 2; }
};
```

Classes – Access Specifiers (cont.)

- The reason we were able to access the members of a struct earlier is because they are **public** by default.
- One big problem with our `Person` class so far is that if we make its member variables `age`, `name`, and `money` private, we have no way to change them. If we make them public, they can be set to an invalid state by any code that instantiates a `Person` object!

```
int main() {
  Person p;
  p.age = -5;   // This doesn't make sense (unless we're Benjamin Button).
}
```

Classes – Encapsulation

To fix this, we make Person's member variables private and add public **accessor** (getter) and **mutator** (setter) functions that set rules on how to access and change them. This is called **encapsulation**!

```
class Person {
public:
  void setAge(int yrs);
  int getAge();
  void setName(string nm);
  string getName();
  void doubleMoney();
  double getMoney();
private:  // Same member variables!
};
```

```
void Person::setAge(int yrs) {
  if (yrs >= 0)
    age = yrs;
}
int Person::getAge() { return age; }

void Person::setName(string nm) {
  if (nm.length > 0)
    name = nm;
}
string Person::getName() { return name; }

void Person::doubleMoney() { money *= 2; }
double Person::getMoney() { return money; }
```

Classes – Encapsulation (cont.)

- Generally, we want to make our member variables private. Therefore, we make them accessible through public member functions that access or mutate them in ways that are reasonable towards our implementation.
  - This keeps objects in a valid state and hides the "nitty gritty" details of our implementation from anyone who wants to use our class
- Now, you just have to call `doubleMoney` on a `Person` and you know that their money will be doubled.

Encapsulation Example

```
int main() {
  string name;
  cout << "What is my name? " << endl;
  getline(cin, name);

  Person p;
  p.setName(name);
  p.setName("");
  cout << "I Am " <<
      p.getName() << "\n";
  p.setAge(49);
  p.setAge(-1);
  cout << "I am " << p.getAge() <<
      " years old.\n";
}
```

```
> What is my name? the Walrus
> I Am the Walrus
> I am 49 years old.
```

Classes vs. Structs

- Technically, the only difference between a class and a struct is the default access specifier.
- By convention, we use structs to represent simple collections of data and classes to represent complex objects.
  - This comes from C, which has only structs and no member functions.

Constructors

- There is yet another problem with our `Person` class: initializing its members one-by-one is annoying but if we don't do it, our `Person` starts in an invalid state!
- A **constructor** is a member function that has the same name as the class, no return type, and automatically performs initialization when we declare an object:

```
class Person {
public:
  Person();
  // Same stuff as before!
};
```

Constructors (cont.)

- Constructors can be defined inside or outside of a class, just like normal functions.
- Like normal functions, constructors can be (and usually are) overloaded with different numbers and types of parameters to suit different purposes.
- Unlike normal functions, they <u>cannot</u> be called with the dot operator.
- A **default constructor** is one with no arguments; the compiler generates an empty one by default.
- The "default default" constructor leaves primitive member variables (int, double, etc.) uninitialized and calls the default constructors of class members
  - Example: any string members will be created with the default string constructor

Constructors – Basic Syntax

Let's add constructors to our Person class!

```
class Person {
public:
  Person();                                     // 1
  Person(int yrs, string nm, double cash);      // 2
  // Same stuff as before!
};

Person::Person() {
  age = 0;
  name = "";
  money = 0.0;
}
```

```
Person::Person(int yrs, string nm, double cash) {
  setAge(yrs);
  setName(nm);
  money = cash;
}

int main() {
  Person p1;   // Constructor 1 is called.
  // Constructor 2 is called.
  Person p2(44, "Elon Musk", 13000000000.0);
  p1 = Person(19, "Freshman at UCLA", -100000.0);
  Person p3(); /* Constructor 1 NOT called:
                  The compiler thinks we're
                  defining a function!    */
  p1.Person(); // Illegal!
}
```

Destructors

- A **destructor** is a member function that is called automatically when an object of a class passes out of scope
  - The destructor should use `delete` to eliminate any dynamically allocated variables created by the object
- For example, suppose that our `Person` class creates a **dynamically allocated** `Pet` type object. This memory would need to be freed in the destructor!

Destructors (cont.)

Let’s add a destructor to our class!

```
class Pet { ... };

class Person {
public:
  Person();
  ~Person();
  // Same stuff as before ...
private:
  Pet* fluffy;
};
```

```
Person::Person() {
  // Same stuff as before ...
  fluffy = new Pet(“Steve”);
}

Person::~Person() {
  delete fluffy;
}
```

Pointers to Objects – Arrow Operator

Like the dot operator, the **arrow operator ->** can be used to access an object's member variables and functions. The arrow operator is used when we have a pointer to the object whose members we are trying to reference.

```
int main() {
  Person* p = new Person;
  p->setAge(20);
  p->setName("Bob");
  double money = p->getMoney();
  p->age = 10;        // ERROR: age is not a public variable!
}
```

Note: `p->setAge(20)` and `(*p).setAge(20)` are equivalent statements!

Pointers to Objects – The `this` Pointer

When defining member functions for a class, we sometimes want to refer to the calling object. The `this` pointer is a predefined pointer that points to the calling object.

```
int Person::getAge() {
  return age;
}
```

```
int Person::getAge() {
  return this->age;
}
```

Note: the above two definitions for the function `getAge()` are equivalent, but the left method is clearer and better stylistically.

Pointers to Objects – The `this` Pointer

- The `this` pointer allows us to access member variables even when they are shadowed by local variables.

```
Person::setAge(int age) {
    this->age = age;
}
```

- The `this` pointer also allows us to pass the current object into a function that takes an argument of its class.

```
void printPerson(Person *p);
Person::print() {
    printPerson(this);
}
p1.print()
```

Function Overloading

- You can have multiple definitions for the same function name in the same scope. However, the definition of the functions must differ from each other by the types and/or the number of arguments in the argument list.

```
class printData {
   public:
      void print(int i) {
         cout << "Printing int: " << i << endl;
      }

      void print(int i, double f) {
         cout << "Printing numbers: " << f << ‘ ‘ << i << endl;
      }

      void print(char* c) {
         cout << "Printing character: " << c << endl;
      }
};
```

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print numbers
   pd.print(42, 500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Practice Question: C String Reversal with Pointers

Implement the function `reverse`, which takes a C String as an argument and prints out the characters in reverse order. You are not allowed to use the `strlen` function, and you must use pointers in any traversal of the C String.

```
void reverse(const char s[]);

int main() {
    char str[] = “stressed”
    reverse(str);
    // OUTPUT: desserts
}
```

Solution: C String Reversal with Pointers

```
void reverse(const char s[]) {
    const char *p = s; // create a new pointer for our traversal
    while (*p != '\0') { // move the pointer to the end of the C String
        p++;
    }
    p--; // set p to point at the last char in the C String
    while (p >= s) { // print out chars as we traverse back to the beginning
        cout << *p;
        p--;
    }
    cout << endl;
}
```

Practice Question: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};


int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }
};
```

What is the output of this code?

Practice Question: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};


int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }
};
```

WAIT!! There’s a memory leak!!

How do we fix it?

Solution: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << "I am a cat named " << m_name << endl;
    }
};


int main() {
  Person p(20, "Pusheen");
  Cat c("Kitty");
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << "I am " << age << " years old" << endl;
      m_cat = new Cat(name);
      cout << "MEOW" << endl;
    }

    ~Person() {
      delete m_cat;
    }
};
```

Solution: Cat Construction

```
Output:

I am 20 years old
I am a cat named Pusheen
MEOW
I am a cat named Kitty
```

Practice Question: strcat

Implement the C string concatenation function. The function takes two C strings and copies the chars from the source C string to the end of the destination C string. The original null byte of the destination is overwritten when copying the source. Return the destination pointer at the end of the function. You do not know the size of the destination and source C strings (so you can't create a temporary C string to store all of the characters!)

```
char* strcat(char* destination, const char* source);
```

Solution: strcat

```
char* strcat(char* destination, const char* source) {
    char* d = destination;
    while (*d) // this loop sets d to point at the null byte of destination
        d++;
    const char* s = source;
    while (*s) { // this loop copies the source C string to where d is pointing
        *d = *s;
        d++;
        s++;
    }
    *d = '\0'
    return destination; }
```

Practice Question: Transpose

Implement a function that takes in a pointer to an nxn 2d array of ints and transposes it. That is, the rows should become the columns and vice versa.

```
void transpose(int** matrix, int n);
```

```
Example: 1 2 3 (n = 3) → 1 4 7
         4 5 6           2 5 8
         7 8 9           3 6 9
```

Solution: Transpose

```
void transpose(int** matrix, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      int temp = matrix[i][j];
      matrix[i][j] = matrix[j][i];
      matrix[j][i] = temp;
    }
  }
}
```

Practice Question: Resolve Merge Issues

```
// Assume arr1 and arr2 are ordered from least to
// greatest and have size n1 and n2, respectively.
// Also assume arr3 has size n1 + n2.
void merge(int arr1[], int n1, int arr2[], int n2,
           int arr3[]) {
  int i1 = 0, i2 = 0;
  for (int i3 = 0; i3 < n1 + n2; i3++) {
    if (arr1[i1] < arr2[i2]) {
      arr3[i3] = arr1[i1];
      i1++;
    } else if (arr2[i2] < arr1[i1]) {
      arr3[i3] = arr2[i2];
      i2++;
    }
  }
}
```

This function attempts to merge two arrays arr1 and arr2 that are ordered from least to greatest into a third array arr3, so that arr3 contains the contents of both arr1 and arr2 ordered from least to greatest.

Example: arr1 = {1, 2, 5}, arr2 = {2, 4, 6}
➜ arr3 = {1, 2, 2, 4, 5, 6}

Can you find and fix the bugs in this function so that it performs correctly?

Practice Question: Resolve Merge Issues

```
// Assume arr1 and arr2 are ordered, n1 and n2 are
// their correct sizes, and arr3 has size n1 + n2.
void merge(int arr1[], int n1, int arr2[], int n2,
           int arr3[]) {
  int i1 = 0, i2 = 0, i3 = 0;
  while (i3 < n1 + n2) {
    if (arr1[i1] < arr2[i2]) {  // what if i1>=n1
      arr3[i3] = arr1[i1];      // or i2 >= n2??
      i1++;
    } else if (arr2[i2] < arr1[i1]) {  // same!
      arr3[i3] = arr2[i2];
      i2++;
    }  // what do we do if arr1[i1] == arr2[i2]?
    i3++;
  }
}
```

This function attempts to merge two arrays arr1 and arr2 that are ordered from least to greatest into a third array arr3, so that arr3 contains the contents of both arr1 and arr2 ordered from least to greatest.

Example: arr1 = {1, 2, 5}, arr2 = {2, 4, 6}
→ arr3 = {1, 2, 2, 4, 5, 6}

Can you find and fix the bugs in this function so that it performs correctly?

Solution: Resolve Merge Issues

```
void merge(int arr1[], int n1, int arr2[], int n2,
           int arr3[]) {
  int i1 = 0, i2 = 0, i3 = 0;
  while (i1 < n1 && i2 < n2) {
    if (arr1[i1] < arr2[i2]) {
      arr3[i3] = arr1[i1];
      i1++;
    } else if (arr2[i2] <= arr1[i1]) {
      arr3[i3] = arr2[i2];
      i2++;
    }
    i3++;
  }
  // continued...
```

```
  while (i1 < n1) {  // only one of these will run
    arr3[i3] = arr1[i1];
    i1++;
    i3++;
  }
  while (i2 < n2) {
    arr3[i3] = arr2[i2];
    i2++;
    i3++;
  }
}
```

Good luck!

Sign-in https://goo.gl/K6QsCt
Slides https://goo.gl/BErYYW
Practice https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring (Not during 10th week):
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.