CS 106B
Lecture 14: Pointers
Monday, July 22, 2017

Programming Abstractions
Summer 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 11
Today's Topics

• Logistics
  • Midterm — good job! Regrade requests due by Tuesday.
  • Chris: extra office hours this week: Monday 4-6pm, Tuesday 4-6pm

• Introduction to Pointers
  • What are pointers?
  • Pointer Syntax
  • Pointer Tips
  • Pointer Practice
  • Binky

• Back to classes
  • The copy constructor and the assignment overload
• Challenge #1: Write a swap function:

```cpp
? swap( ? ) {
    ...
}
```

```cpp
int main() {
    // swap the two variables below, using a swap function
    int a = 5;
    int b = 12;
    swap( ? );
    // at this point, a should equal 12 and b should equal 5
}
```
C++ Challenge

- Can we write a swap function in C++ using what we know already? Yes, we can!

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

int main() {
    // swap the two variables below, using a swap function
    int a = 5;
    int b = 12;
    swap (a, b);
    // at this point, a equals 12 and b equals 5
}
```
So it turns out that references are a nice C++ feature, but they abstract away some of the lower-level details that we might want to know about.

In order for our swap function to work, we must have access to the original elements.

This starts to fall under the category of "memory management".

As a close relative to C, C++ gives us access to all of C's low-level functionality.
Introduction to Pointers

• The next major topic is about the idea of a pointer in C++. We need to use pointers when we create data structures like Vectors and Linked Lists (which we will do later this week!)

• Pointers are used heavily in the C language, and also in C++, though we haven't needed them yet.

• Pointers delve under the hood of C++ to the memory system, and so we must start to become familiar with how memory works in a computer.
Introduction to Pointers

• The memory in a computer can be thought of simply as a long row of boxes, with each box having a value in it, and an index associated with it.
• If this sounds like an array, it's because it is!
• Computer memory (particularly, Random Access Memory, or RAM) is just a giant array. The "boxes" can hold different types, but the numbers associated with each box is just a number, one after the other:

<table>
<thead>
<tr>
<th>values (ints):</th>
<th>7</th>
<th>2</th>
<th>8</th>
<th>3</th>
<th>14</th>
<th>99</th>
<th>-6</th>
<th>3</th>
<th>45</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>associated index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>values (strings):</th>
<th>cat</th>
<th>dog</th>
<th>apple</th>
<th>tree</th>
<th>shoe</th>
<th>hand</th>
<th>chair</th>
<th>light</th>
<th>cup</th>
<th>toe</th>
</tr>
</thead>
<tbody>
<tr>
<td>associated index:</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>
Introduction to Pointers

- In C++, we just call those boxes variables, and we call the associated indices addresses, because they can tell us where the variable is located (like a house address).

<table>
<thead>
<tr>
<th>variable:</th>
<th>cat</th>
<th>dog</th>
<th>apple</th>
<th>tree</th>
<th>shoe</th>
<th>hand</th>
<th>chair</th>
<th>light</th>
<th>cup</th>
<th>toe</th>
</tr>
</thead>
<tbody>
<tr>
<td>address:</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>

```cpp
string pet = "cat";
```

- What is the address of the `pet` variable?

10

- The operating system determines the address, not you! In this case it is 10, but it could be any other address in memory.
Introduction to Pointers

- Guess what? If we store that **memory address** in a different variable, it is called a **pointer**.

```cpp
string pet = "cat";  // Some other variable:
```

<table>
<thead>
<tr>
<th>pet</th>
<th>petPointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1234</td>
</tr>
</tbody>
</table>

So, what is a pointer?

**A memory address!**
Introduction to Pointers

What is a pointer??

a memory address!
Introduction to Pointers

- We really don't care about the actual memory address numbers themselves, and most often we will simply use a visual "pointer" to show that a variable points to another variable:

```cpp
string pet = "cat";
```

![Diagram showing the relationship between variables and their pointers](image)
Introduction to Pointers

• We really don't care about the actual memory address numbers themselves, and most often we will simply use a visual "pointer" to show that a variable points to another variable:

```cpp
string pet = "cat";
```

 petPointer variable
Introduction to Pointers

What you need to know about pointers:

• Every location in memory, and therefore every variable, has an address.
• Every address corresponds to a unique location in memory.
• The computer knows the address of every variable in your program.
• Given a memory address, the computer can find out what value is stored at that location.
• While addresses are just numbers, C++ treats them as a separate type. This allows the compiler to catch cases where you accidentally assign a pointer to a numeric variable and vice versa (which is almost always an error).
Pointer syntax can get tricky. We will not go too deep -- you'll get that when you take cs107!

**Pointer Syntax #1:** To declare a pointer of a particular type, use the "*" (asterisk) symbol:

```c
string *petPtr; // declare a pointer (which will hold a memory address) to a string
int *agePtr; // declare a pointer to an int
char *letterPtr; // declare a pointer to a char
```

The type for petPtr is a "string *" and not a `string`. This is important! A pointer type is distinct from the pointee type.
Pointer Syntax #2: To get the address of another variable, use the "&" (ampersand) character:
Pointer Syntax #2: To get the address of another variable, use the "&" (ampersand) character:

```c
string *petPtr; // declare a pointer (which will hold a // memory address) to a string
```

(pointer pointing to memory address 1234)

(dead squirrel!)
**Pointer Syntax #2:** To *get the address of another variable*, use the "&" (ampersand) character:

```c
string *petPtr; // declare a pointer (which will hold a
                // memory address) to a string

string pet = "cat"; // a string variable
```

```
  pet     petPtr
   cat    ?
    10    1234
```

*(dead squirrel!)*
**Pointer Syntax #2:** To *get the address of another variable*, use the "&" (ampersand) character:

```cpp
string *petPtr; // declare a pointer (which will hold a memory address) to a string
string pet = "cat"; // a string variable

petPtr = &pet; // petPtr now holds the address of pet
```

![Diagram showing the relationship between `pet`, `petPtr`, `10`, and `1234`]
Pointer Syntax #2: To get the address of another variable, use the "&" (ampersand) character:

```c
string *petPtr; // declare a pointer (which will hold a memory address) to a string
string pet = "cat"; // a string variable

petPtr = &pet; // petPtr now holds the address of pet
```

you almost never need to do this in 106B!!!
Pointer Syntax #3: To get value of the variable a pointer points to, use the "*" (asterisk) character (in a different way than before!):

```cpp
string *petPtr; // declare a pointer to a string
string pet = "cat"; // a string variable
petPtr = &pet; // petPtr now holds the address of pet
cout << *petPtr << endl; // prints out "cat"
```

This is called "dereferencing" the pointer: the asterisk says, "go to where the pointer is pointing, and return the value stored there"
**Pointer Tip #1**: To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to `NULL`, which means "no valid address" (it actually is just 0 in C++).

Instead of this:

```c++
string *petPtr; // declare a pointer to
// a string with a dead squirrel
```

```c++
petPtr
```

(1234)

(dead squirrel!)
**Pointer Tip #1**: To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to `NULL`, which means "no valid address" (it actually is just 0 in C++).

Do this:

```cpp
string *petPtr = NULL; // declare a pointer to a string that points to NULL
```

![Diagram showing a pointer set to NULL](image)
**Pointer Tip #2**: If you are unsure if your pointer holds a valid address, you should check for **NULL**

Do this:

```cpp
void printPetName(string *petPtr) {
    if (petPtr != NULL) {
        cout << *petPtr << endl; // prints out the value
        // pointed to by petPtr
        // if it is not NULL
    }
}
```
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
```

What type does this pointer point to?
What should we draw?
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
```

What type does this pointer point to? **an int**

What should we draw?
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
```

What type does this pointer point to? **an int**

What should we draw?

```plaintext
NULL

0

7224
```

nPtr
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
```

What type does this pointer point to? **an int**

What should we draw?

![Diagram](image)

We don't care what this number is, just that it tells us where `nPtr` is in memory.
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
int n = 16;

What should we draw?
```
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
int n = 16;
```
These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
int n = 16;
nPtr = &n;
```

What should we draw and fill in?

```
16
42
```

```
0
7224
```

```c
nPtr
```
Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```c
int *nPtr = NULL;
int n = 16;
nPtr = &n;
```

We now say that nPtr points to n.
What is a pointer??

a memory address!
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
```cpp
string *sPtr = NULL;
string s = "hello"
sPtr = &s;
cout << *sPtr << endl;
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;

Output: hello
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;

Output?
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;

Output? Seg Fault! (crash!)
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;

Output? Seg Fault! (crash!)

Be careful when dereferencing pointers!
You can also use the dereferencing operator to set the value of the "pointee" (the variable being pointed to):

```cpp
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```
• You can also use the dereferencing operator to set the value of the "pointee" (the variable being pointed to):

```c++
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```
Pointer Practice

- You can also use the dereferencing operator to set the value of the "pointee" (the variable being pointed to):

```cpp
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```
Pointer Practice

• You can also use the dereferencing operator to set the value of the "pointee" (the variable being pointed to):

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

int main() {
    string *sPtr = NULL;
    string s = "hello";
    sPtr = &s;
    *sPtr = "goodbye";
    cout << s << endl;
    return 0;
}
```
• You can also use the dereferencing operator to set the value of the "pointee" (the variable being pointed to):

```cpp
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```
• You can also use the dereferencing operator to set the value of the "pointee" (the variable being pointed to):

```cpp
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```

Output: `goodbye`
**Pointer Practice**

- If you set one pointer equal to another pointer, they *both point to the same variable!*

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```
If you set one pointer equal to another pointer, they *both point to the same variable!*
If you set one pointer equal to another pointer, they both point to the same variable!

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```
Pointer Practice

• If you set one pointer equal to another pointer, they both point to the same variable!

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;
sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```

Output: `hello`
If you set one pointer equal to another pointer, they both point to the same variable!

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```
• If you set one pointer equal to another pointer, they both point to the same variable!

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```

Output: `hello`
Pointer Practice

• If you dereference and assign a different value, both pointers will now print the same value!

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;

*sPtr1 = "goodbye";
cout << *sPtr1 << endl;
cout << *sPtr2 << endl;
```

If you dereference and assign a different value, both pointers will now print the same value!
• If you dereference and assign a different value, both pointers will now print the same value!

```cpp
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;

*sPtr1 = "goodbye";
cout << *sPtr1 << endl;
cout << *sPtr2 << endl;
```

Output: goodbye
goodbye
What is a pointer??

a memory address!
More information about addresses:

Addresses are just numbers, as we have seen. However, you will often see an address listed like this:

0x7fff3889b4b4
or this: 0x602a10

This is a base-16, or "hexadecimal" representation. The 0x just means "the following number is in hexadecimal."

The letters are used because base 16 needs 16 digits:

0 1 2 3 4 5 6 7 8 9 a b c d e f
So, you might see the following -- remember, we don't actually care about the address values, just that they are memory locations.
Our very own Nick Parlante (another CS Lecturer) put this video together many years ago:

https://www.youtube.com/watch?v=6pmWojisM_E
Let's write our swap function with pointers!
Let's write our swap function with pointers!

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```
Okay, now we know about classes and pointers. We will learn about the "new" operator on Monday, but for now just know that it gives us space to hold variables that doesn't get lost when a function ends.

Let's take a look at a simple Rectangle class (almost certainly not the way we would really write this class)
#pragma once

class Rectangle {
public:
    Rectangle(double width = 1, double height = 1); // constructor
    ~Rectangle(); // destructor (more on this later)
    double area();
    double perimeter();
    double getHeight();
    double getWidth();

private:
    double *height; // pointer to a double
    double *width;  // pointer to a double
};
Back to Classes: the Copy Constructor

rectangle.cpp:

```cpp
#include "rectangle.h"

Rectangle::Rectangle(double width, double height) { // constructor
    this->width = new double;
    this->height = new double;
    *(this->width) = width;
    *(this->height) = height;
}

Rectangle::~Rectangle() { // destructor
    delete height;
    delete width;
}

double Rectangle::area() {
    return *width * *height;
}

double Rectangle::perimeter() {
    return 2 * *width + 2 * *height;
}

double Rectangle::getHeight() {
    return *height;
}

double Rectangle::getWidth() {
    return *width;
}
```
rectangle.cpp:

```cpp
int main() {
    Rectangle r(3,4);
    cout << "Width: " << r.getWidth() << ", ";
    cout << "Height: " << r.getHeight() << endl;
    cout << "Area: " << r.area() << endl;
    cout << "Perimeter: " << r.perimeter() << endl;
    // let's make a copy:
    Rectangle r2 = r;
    return 0;
}
```

no problem...

crash!
What happened?

```c
int main() {
    Rectangle r(3,4);
    Rectangle r2 = r;
}
```
The default is to copy the values...

```c
int main() {
    Rectangle r(3, 4);
    Rectangle r2 = r;
}
```
The default is to copy the values...

```c
int main() {
    Rectangle r(3, 4);
    Rectangle r2 = r;
}
```
Problem! Now both r and r2 point to the same ints!

```c
int main() {
    Rectangle r(3, 4);
    Rectangle r2 = r;
}
```
What to do? Define a "copy constructor"

The copy constructor tells the compiler how to copy your class. It is important to do this so you don't end up with the situation on the previous slides.

```cpp
class Rectangle {
    public:
        Rectangle(double height = 1, double width = 1); // constructor
        Rectangle(const Rectangle &src); // copy constructor
        ...
};
```

```cpp
Rectangle::Rectangle(const Rectangle &src) { // copy constructor
    width = new double; // request new memory
    height = new double;

    // copy the values
    *width = *src.width;
    *height = *src.height;
}
```
• Pointers
  • A pointer is just a memory address that refers to the address of another variable
  • Pointers must point to a particular type (int *, char *, string *, etc.)
  • To declare a pointer, use * (e.g., \texttt{string *stPtr})
  • To get the address of a variable to store in a pointer, use \\
  • To access the value pointed to by a pointer, use the *
  • Watch out for NULL pointers!
  • Two pointers can point to the same variable.
For Next Time

Dynamic Memory Allocation!

new  
delete  
arrays  
assignment overload (similar in principle to the copy constructor)
References and Advanced Reading

• References:
  • More on C++ classes: https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
  • C++ Pointers: https://www.tutorialspoint.com/cplusplus/cpp_pointers.htm

• Advanced Reading:
  • Fun video on pointers: https://www.youtube.com/watch?v=B7IVHq-cgeU
  • Hexadecimal numbers: http://www.binaryhexconverter.com/hex-to-decimal-converter
  • Pointer arithmetic: https://www.tutorialspoint.com/cplusplus/cpp_pointer_arithmatic.htm
  • More on pointers: https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html