# C++ at Velocity, Part 2

CS 002 - WI 2016
January 6, 2016

# General remarks

- Course website is up on Moodle
  - Enrollment key: 'CS2EnrollME'
- Assignment 1 is on Moodle
  - 12 points of fairly basic C++
  - 3 points of tic-tac-toe
  - up to 5 points of "other stuff"
  - Email questions to cs2tas-2016@caltech.edu
  - Office hours start on Friday

# Today

Coding style
Git introduction
Pointers
gdb

# Coding style guidelines

- We don't really enforce a particular style in this course.
- Only requirements:
  - Clean
  - Readable
  - Consistent
- The point: someone else eventually needs to maintain your code!
  - ...you, after you've forgotten the details
  - ...your coworkers

# Coding style suggestions

- Spacing between operands
  - What's easier to read?

    `int y=a+b;` vs. `int y = a + b;`
- Control statement spacing
  - What's easier to read?

    `if(a==b)` vs. `if (a == b)`
- Block indentation
  - use spaces, not tabs
  - keep indentation consistent (as if Python)
- Variable and function naming
  - try to pick names that convey semantic meaning
    - common exception: loop variables `i, j, k`

(C++ 2.27-29)

# Coding style guidelines

- Commenting: explain, don't repeat
  - Don't tell me **`i++`** increments **`i`**; tell me why it's there
- Explain complex or subtle code
- Write function headers (even small ones)
  - Invalid arguments? Return value?

If you woke up one day without your memories then how would you reconstruct your life?

# Git

- Git is a version control system
  - Allows us to create snapshots and backups of our code conveniently
- See http://pgbovine.net/git-tutorial.htm for a series of three introductory videos (30 minutes)
  - Interested in more advanced git?
  - Check out http://pcottle.github.io/learnGitBranching/
- We will be using Git to distribute all assignments
- You will have to use Git for the final project!

# Two Git Commands

Fetch code for assignment 1

```
git clone
 https://bitbucket.org/caltechcs2/cs2_week
 1_2016.git
```

Pull any updates to the assignment

```
git pull
```

It is a good idea to run this before beginning work each day

# Pointers crash course

- All variables have an address and a value
  - Everything is stored *somewhere* in the computer's memory; that location is specified by a unique **address**
  - Every variable has some kind of value (though the type changes the way we interpret that value); when we say "`int i = 5`", we are setting the **value** of `i`
- `&` takes the address of a variable
  - So `&i` is the address of the variable `i`
- `*` gets the value stored at some address; this is called **dereferencing**
  - So if `addr` is a valid address, `*addr` is the value stored there

# The basics of pointers

When a variable **i** is declared, some memory is reserved for its contents.

This memory has an **address** **&i**.

```
int i = 10;
cout << "i is at " << &i << endl;
```

This prints something like `"i is at 0xff831f2c"`.

This number is **i**'s address.

# The basics of pointers

A pointer is a variable that holds an **address**

```
int i = 10;
int * j = &i; // j 'points' to i
```

| name | address | contents |
|---|---|---|
| i | **0xff831f2c** | 10 |
| j | 0xff831f30 | **0xff831f2c** |

**&** is the **address-of** operator.

(C 4.7-4.9)

# The basics of pointers

A pointer is a variable that holds an **address**

```
int i = 10;
int * j = &i; // j 'points' to i
cout << "j = " << j << endl;
cout << "j points to: " << *j
 <<endl;
```

`*j` is the contents of memory at the address in `j`; * operator **dereferences** `j`

# Beware: The many uses of *

```
c = a * b; // multiplication

int * p1; // pointer declaration

int foo = *p2; // dereferencing
```

# Pointers and call-by-reference

- Function calls in C++ copy arguments by default.
  - normally can't change a variable we pass to a function
- Passing a memory address instead lets us make changes.
- We also avoid copying large amounts of data
  - imagine having to copy an entire picture each time you want to change it!

```
void incr(int * i)
{
   (*i)++;
}

// ... later ...

int j = 10;
incr(&j);
// j is now 11
```

# C++ references and call-by-reference

- C++ allows us to mark arguments as references.
  - Use them exactly as you would the variable.
  - Changes are passed through.
- Often cleaner syntax!
  - fewer parentheses and * floating around
- Can't do everything pointers can do.

```
void incr(int & i)
{
  i++;
}

// ... later ...

int j = 10;
incr(j);
// j is also now 11!
```

# Pointer arithmetic

- We can change where a pointer points
  - Usually by assigning a new memory address as its value
- We can also add to, and subtract from, pointer variables
  - Changing the memory address!

```
int i[50];
int * j = &i[0]; // j points to i[0]
j += 5; // j now points to i[5]
```

# Pointer arithmetic

Note that we are adding and subtracting multiples of the type size!

```
int i[50]; // sizeof(int) usually 4
// suppose i[0] lives at 0xff000000
int * j = &(i[0]);
j += 5; // j now points to 0xff000014
        // NOT 0xff000005!
        // we added 5 * sizeof(int)
```

# Array-pointer equivalence

Array notation is syntactic sugar for pointers.

```
int arr[5] = {1, 2, 3, 4, 5};
cout << "arr[3] = " << arr[3] << endl;
cout << "arr[3] = " << *(arr + 3) <<endl;
```

**`arr[3]`** and **`*(arr + 3)`** are identical!
**`arr`** is identical to **`&arr[0]`**

Any array operation can be replaced by a pointer operation.

# Dynamic memory allocation

Pointers become useful when we need to allocate arrays on the fly

Here's how:

```
int n = 9001;
// ... later ...
int * buf = new int[n];
```

This creates a new block of integers of size n

(C++ 2.9, 2.11-12)

# Dynamic memory allocation

Free your memory after using it!
C++ does not clean up automatically
Forgetting to free memory == memory leaks

```
double a = new double;
int * buf = new int[90001];
// ... later ...
delete a;     // destroy singleton
delete[] buf; // destroy array
```

(C++ 2.9, 2.11-12)

# Tools - gdb

- gdb is a **debugger**
- Allows systematic, careful examination of program execution and state
- Command-line based
- Use when you want to step through a program line-by-line
- Use when a program crashes

(C 6.2 (I'm basically mirroring Mike's slides))

# Tools - gdb

Before using **gdb**:

- compile with **-g**
- this includes source code information with program

Then run **gdb ./programname**

You're now in a **gdb** terminal

```
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-
  2012.04
  …
  (gdb)
```

# gdb – Running a Program

- **`run`** starts a program running
  - If all goes well, the program runs normally
  - If something goes wrong, GDB usually tells you, then stops
- Command-line arguments are given to **`run`**
  - e.g. **`run 1 2 3`**
- **`start`** starts the program, then stops at the first line
  - Useful if you need to step through from the beginning

# gdb - Breakpoints

To make sure the debugger stops somewhere, set a breakpoint with **`break`**

- **`break foo.c: 100`**
  stops when execution reaches the specified line

To clear a breakpoint, use **`clear`** or **`delete`**

- **`clear foo.c: 100`** (by line number)
- **`delete 1`** (by breakpoint number)

# gdb - Moving Around

**continue** - runs until the next breakpoint

**next** - runs the next source code line

**step** - runs the next source code line, tracing into function calls

**finish** - traces out of the current function

# gdb - Gathering Information

**`print`** - prints arbitrary expressions

can even call functions!

```
(gdb) print i
$1 = 42
(gdb) print square(i)
$2 = 1764
```

**`display`** - prints a value each time the program stops

```
(gdb) display *p
```

# gdb - Gathering Information

- **`backtrace`** - prints call stack
  - what functions are waiting to be resolved?

```
(gdb) bt
#0  divide (a=15625, b=37) at debugging2.cpp:31
#1  0x0000000000400566 in main (argc=1, argv=0x7fffffffe288)
    at debugging2.cpp:74
```

  - topmost = innermost function
- Things to look for
  - Accessing arrays out of bounds
  - Dereferencing invalid pointers (segmentation fault)
  - Freeing memory that was never allocated
  - Freeing memory that was already freed

# Next time...

- Classes
- Additional topics, time permitting
  - Operator overloading
  - Class hierarchy
  - The C++ STL