Course Introduction / C++ at Velocity

Ben Yuan
CS 002 - WI 2014
January 6 - 10, 2014

General remarks

- This course is taught in C++
- We expect you to have a Linux environment
    - later assignments need platform-specific tools
    - VirtualBox image will be available on course website
    - can use the Annenberg lab
    - ask TAs for help setting up!
- Office hours: Sunday / Monday 18 - 24
- TA mailing list: cs2-tas@ugcs.caltech.edu
    - reaches all of us
    - please ask us questions if you have them!

General remarks

- We recommend you have a CMS cluster account
  - http://acctreq.cms.caltech.edu/cgi-bin/request.cgi
- If you don't know Linux / UNIX, then spend some time playing with a Linux system
  - ITS tutorial: http://www.imss.caltech.edu/node/324

General remarks

- Collaboration policy: do your own work!
  - Helping other students with debugging: keep your own code "50 feet" away
  - Discussing problems: discard shared work product after discussion
  - External resources: fine unless stated, but don't borrow code / look up solutions
  - Policy unclear? Ask a Head TA
- Learning needs discovery, synthesis, and practice - NOT just "correct answers"

General remarks

- Assignment grading: 20 points
  - ~15 'main' points
  - ~5 'advanced' points
- Due date/time: 1700 Tuesday
- Extension policy: two 48-hr extensions
  - applied automatically (no need to tell us)
  - Health Center / Deans' note / emergencies: talk to a Head TA (even after the fact)
- Late policy: don't be late!
  - 1/3 of unmodified grade deducted per 24h
- Course pass line: 120 pt + no missing work!

General remarks

- This presentation moves very quickly!
- This presentation will refer to slides in the CS11 C and C++ slides (lecture.slide)
    - CS11 C slides: http://courses.cms.caltech.edu/cs11/material/c/mike/
    - CS11 C++ slides: http://courses.cms.caltech.edu/cs11/material/cpp/donnie/
- Ask questions if you have them!

What is C++?

- **General-purpose compiled** programming language
- Emphasis on **object-oriented** design and programming
- Capable of creating **fast**, **efficient** programs if used right
- Capable of **horrible things** if used wrong!

(C 1.6)

A simple C++ program

```
#include <cstdio>

int main(int argc, char ** argv)
{
  printf("Hello, world!\n");
  return 0;
}
```

(C 1.7)

Language Overview

- C++ programs are built up from **functions**
  - take zero or more **arguments**
  - do some computation
  - (possibly) alter the program state
  - (possibly) **returns** some result
- C++ source code is organized into **source** files and **header** files
  - header files: function / class declarations
  - source files: implementation details
- Every C++ program starts at **`main()`**

(C 1.9, 1.12)

Types and variables

- C++ is a **statically typed** language
  - You have to tell the compiler what type a variable is.
  - Variables hold data of a single type only.
  - Variables must be declared before use.
- To declare a variable, give it a type and a name, and optionally an initial value:

  ```
  int foo = 42;
  ```

  - Uninitialized variables have an undefined value until assigned to by some statement!

(C 1.13-14, 2.15-16)

Type conversion

- You can convert variables from one type to another (where allowed).
    - Let's see why this might be useful...

```
int a = 3, b = 4;
double c;
// Let's do some math.
c = a / b;
// c = 0!?
```

(Type conversion: C 2.16-17)

Type conversion

- You can convert variables from one type to another (where allowed).
    - This is the **type conversion operator**.

```
int a = 3, b = 4;
double c;
// Let's do some math.
c = ((double) a) / ((double) b);
// c = 0.75 :)
```

(Type conversion: C 2.16-17)

Operators and expressions

- Many other kinds of **operators**
  - Assignment: `=  +=  -=` etc.
  - Arithmetic: `+  -  *  /  %`
  - Increment/decrement: `++  --`
  - Bitwise: `&  |  ~  ^  <<  >>`
  - Comparison: `==  !=  <  >  <=  >=`
  - Logical: `&&  ||  !`
- Operators are used to build **expressions**
  - `i * 3 + 4 * 5`
  - expressions have values (assignable to variables)

(C 1.16-23)

Functions

- We define functions like this:

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

  - Return type, function name, argument list
- If we don't need to return anything, return `void`:

  ```
  void print_sum(double x, double y)
  {
    printf("%f\n", x + y);
  }
  ```

(Functions: C 1.16-23; printf: C 1.45)

Functions

- We can call functions we defined:

  ```
  print_sum(5.0, 6.0); // prints 11.0
  ```

- Functions with return values can be part of expressions:

  ```
  double foo = square(6.0) + 1;
  // foo is now 37.0
  ```

Variable scope

- Variables are only valid within a particular **scope**.
- **Local** variables only exist within the function or block in which they are defined.

```
void f()
{
    int a;
    // ... stuff ...
}

void g()
{
    a = 5; // invalid
}
```

(C 1.42-43)

Functions and variable scope

- By default, C++ functions are **pass-by-value**
  - Function receives a copy of the passed-in value

```
void f(int a)
{
   a = 8; // this is a local change
}

void g()
{
   int p = 4;
   f(p); // p is still 4!
}
```

Variable scope

- Variables defined outside any function are **global**.
- Global variables are available "everywhere".
- Try not to use global variables!
  - Problems arise when global variables are changed from multiple places.

```
int a; // is global
void f()
{
  a = 2; // fine
}
void g()
{
  a = 3; // also OK
}
```

(C 1.42-44)

#define

- If you need a constant, use **#define**

```c
#define PI 3.14159

void f()
{
    printf("%f\n", PI); // prints 3.14159
}
```

(C 2.3-7)

printf()

- More than just a string printer: printf == 'formatted print'

```
int a = 5;
double pi = 3.14159;
char s[] = "I am a string!";
printf("a = %d, pi = %f, s = %s\n", a, pi, s);
// prints a = 5, pi = 3.14159, s = I am a string!
```

- Substitutes values for `%d`, `%f`, `%s`, etc.
  - `%d`: int
  - `%f`: float
  - `%s`: string
  - `\n`: new line

(C 1.45)

Conditional statements

- Indicated by `if` keyword
- Does something iff the given expression is 'true' (nonzero)
- Optional else if statement allows further condition check if preceding (`else`) `if` block not matched
- Optional `else` statement executed if preceding (`else`) `if` block not matched

```c
if (a < 1)
{
    printf("less than 1\n");
}
// optionally
else if (a == 1)
{
    printf("is 1\n");
}
// optionally
else
{
    printf("more than 1\n");
}
```

(C 1.48-53)

Note

Beware: = IS NOT ==

```
if (a = 3)
{
  // this ALWAYS executes no matter
  // what a may have been before
  // and overwrites a with 3!
  printf("a is 3\n");
}
```

Note

Beware: = IS NOT ==

```
if (a == 3)
{
  // this does what we want
  printf("a is 3\n");
}
```

Looping

- `while` loop - repeats contents while the given condition is true
  - condition check at beginning of loop

```
int i = 0;
...
while (i < 5)
{
   printf("hi ");
   i += 1;
}
```

(C 2.21-22)

Looping

- **`do-while`** loop - repeats contents while the given condition is true
    - condition check at end of loop
    - guaranteed to run contents at least once

```
int i = 0;
...
do
{
  printf("hi ");
  i += 1;
} while (i < 5);
```

(C 2.25)

Looping

- **`do-while`** loop - repeats contents while the given condition is true
    - condition check at end of loop
    - guaranteed to run contents at least once

```
int i = 6;
...
do
{
   printf("hi ");
   i += 1;
} while (i < 5);
```

(C 2.25)

Looping

- **`for`** loop - like **`while`**, but with extra sugar
  - runs a statement when loop is first reached
  - checks a condition before every iteration
  - runs a statement after every iteration

```
for (int i = 0;
     i < 5;
     i++)
{
  printf("hi ");
}
```

(C 1.54-59)

Arrays

- **Arrays** are linear sequences of data
- Simplest vector/sequence type
  - supports random access
  - fixed size - nonresizable!
  - elements are contiguous in memory
  - no range checking D:

```
// uninitialized array, length 10
int arr1[10];
// initialized array, length 5
int arr2[5] = {1, 1, 2, 3, 5};
```

(C 3.2-7)

Arrays

- Arrays can be addressed by element index
  - arrays are 0-indexed

```
int arr[10];
...
for(int i = 0; i < 10; i++)
{
   arr[i] = i * 2;
}
```

(C 3.7)

Arrays and Strings

- Arrays are often used to buffer string data
  - a string is really an array of characters
  - C-style strings are null-terminated

```
char in[100];
scanf("%99s", in); // read in some text
printf("You said: %s\n", in);
```

(C 2.28-34)

Pointers!

- Pointers are cool!
- The notation can be confusing...
- but pointers are extremely useful.
  - Indirection
  - Call-by-reference
  - Dynamic memory allocation

(C 4.6)

Pointers

- When a variable `i` is declared, some memory is reserved for its contents.
- This memory has an **address** `&i`.

```
int i = 10;
printf("i is at %p\n", &i);
```

- This prints something like "i is at 0xff831f2c".
- This number is `i`'s address.

(C 4.6-7)

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)

Pointers

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

```
int i = 10;
int * j = &i; // j 'points' to i
printf("j = %p\n", j);
printf("j points to: %d\n", *j);
```

- `*j` is the contents of memory at the address in `j`; `*` operator **dereferences** j
  - "What is j pointing to?"

(C 4.8-9)

The many uses of *

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

```
int * p1; // pointer declaration
```

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

(C 4.10)

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 4.19-23)