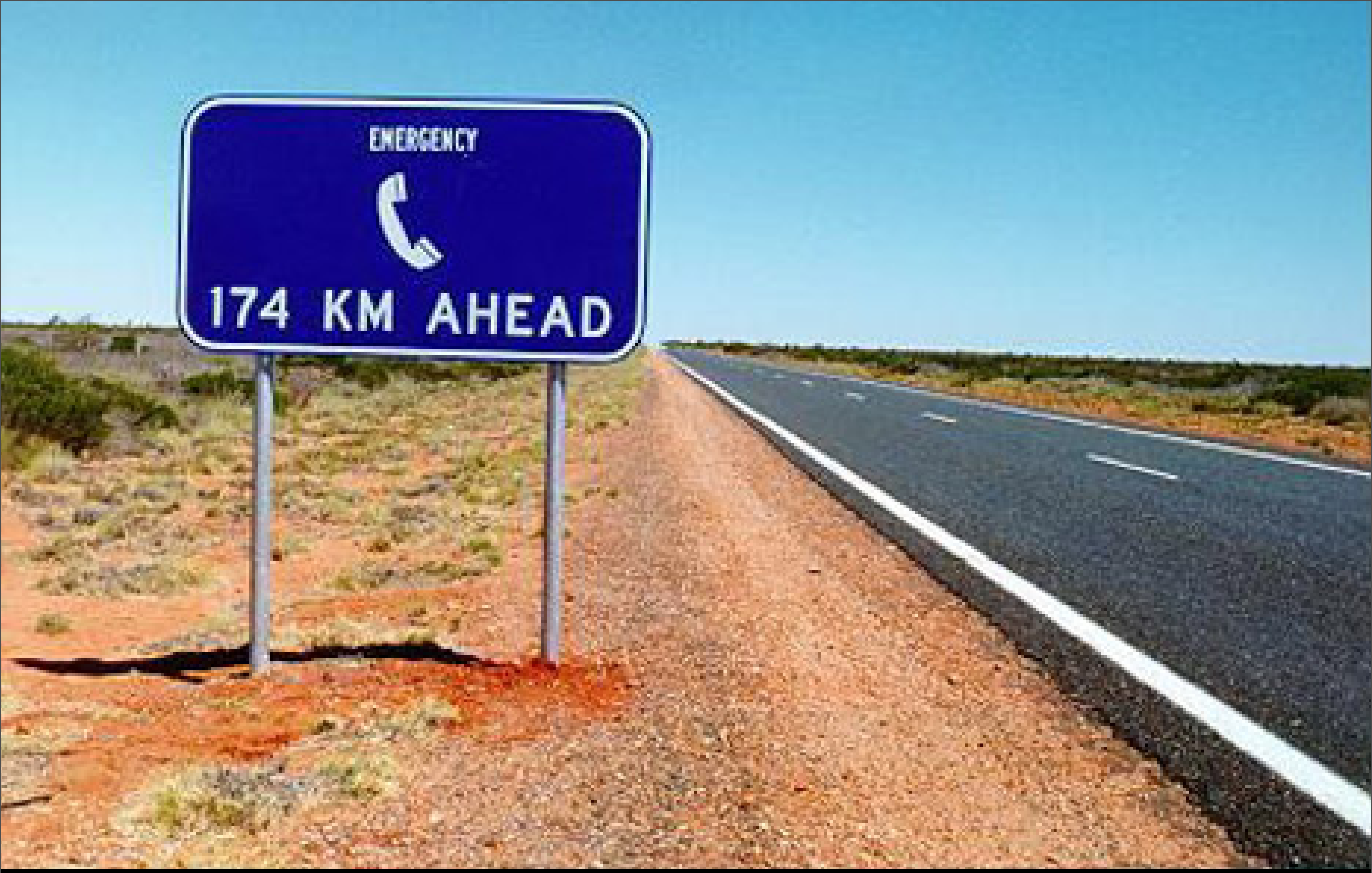


PROGRAMMING IN C++

Logistics

- Introductions
- Go over syllabus
- About programming and languages
- A little history
- About C++ as a language
- How to write and compile a simple program
- How to write programs in general

What is Programming?

- Computers are dumb - making them do useful things involves telling them *exactly* what to do, step by step.
- Programming is the process of taking a program and breaking it down into a sequence of steps the computer can handle (CPU instructions).
- CPU instructions are *very* simple: e.g. add two numbers, fetch a number from memory, see if a number is equal to zero, etc.

Low Level Language

- Usually this refers to assembly code
- Each code instruction becomes a single CPU instruction
- Hard to write, hard to read, hard to maintain
- Not portable between different CPU families
- ... but great for control freaks, and can result in very efficient code

High Level Languages

- Most programming languages are high-level (C++, C, Perl, PHP, Java, Pascal, Python...)
- Easy to write, easy to read (easi*er*, anyway)
- Not (usually) machine specific
- Must be translated into assembly language
    - ... meaning it often loses something in efficiency

A Bit ‘o History

- Before C++ came C, in the early 1970s
- C was originally invented (no kidding) to play Space Travel, a video game
- Developed along with UNIX
- C was designed to be minimalist:
    - Easy to compile into fast machine code
    - Nothing “behind the scenes”
    - Low level access to memory

...and then came C++

- C++ is an extension to C, from the early 1980s
- C++ is "C with classes", and a few extra language features
- ... but still with most of the low-level-ness of C (no hand-holding)
- Fast, powerful, standardized, and very popular
- ... but error prone if you are not careful!

C++ is:

- **Compiled** (translated into machine code in advance, before run-time)
- **Strongly typed**, meaning that each variable has a type associated with it (float, int, whatever)
- **High level**... but still pretty low
- **Portable** - the same code can often be compiled on many different kinds of computers with little or no modification

Writing a C++ program

- Programs can be written in any text editor
- On the lab machines try gedit, nedit, emacs, KDevelop, etc.
- Via SSH (linux.cs.uiowa.edu) try pico
- Use whatever text editor or platform or compiler you are most comfortable with, but your program **must** compile on Linux using g++!

```
// a sample program, by Greg
#include <iostream>
using namespace std;

int main()
{
   int baz = 2;
   int foo = 21;
   int result;

   // multiply some stuff
   result = foo * baz;

   // output the result
   cout << “Hello world: ”
        << result << endl;

   return 0;
}
```

Compiling/Running it

- 1. Compile with g++
- 2. If it works, run the resulting executable
- 3. Like this:

```
[gbnichol@serv16 ~/cpp]$ ls
main.cpp
[gbnichol@serv16 ~/cpp]$ g++ -o program main.cpp
[gbnichol@serv16 ~/cpp]$ ls
main.cpp program
[gbnichol@serv16 ~/cpp]$ ./program
Hello world: 42
[gbnichol@serv16 ~/cpp]
```

Errors!

- Errors are problems with your program
- Different kinds of errors:
  - Compiler errors
  - Linker errors
  - Runtime errors

Compiler Errors

- Compiler errors are problems with your code that result in it not compiling
- Code errors, typos, spelling errors, etc.
- Errors must be fixed before the code will compile; warnings don't *have* to be fixed (but you should probably fix them anyway, if you can)

Linker Errors

- Each cpp file is compiled into an *object file*, which contains the compiled version of that code
- All the object files are "linked" together into a single executable program
- If the object files don't mesh together well (missing functions, duplicate functions, etc.) you get linker errors
- These must be fixed before you can run your program

Runtime Errors

- You know... bugs!
- Anytime your program crashes or in general doesn’t work correctly
- Divide by zero, running out of memory, or just doing the wrong thing

Errors

- Finding and fixing errors can be tricky and sometimes frustrating - some errors can be hard to find (= vs == for example)
- Solution: practice and be patient.
- The only way to get good at this is to do lots of it!

Thoughts on Programming

- Programming is the process of taking a program and breaking it down into a sequence of steps you can put into code.
- This is not always easy.
- Doing it *well* requires patience and practice.
- It can be fun, though. Really. :-)

Programming (in general)

- Divide the project up into small chunks.
- Write each chunk independently. Use comments to document anything that needs it.
- *Test* that chunk. Make sure it works.
- *Then* move onto other chunks.
- Compile early and often, and fix any errors and warnings before moving on.

Variables and Memory

- Each variable:
  - has a name (identifier)
  - has a type (bound at compile-time)
  - has its own location in memory (address)
  - takes up a certain number of bytes
  - ... and of course has a value

Variable Names

- Rules for variable names in C++:
    - Can contain letters, numbers, or underscores
    - Must begin with a letter or an underscore
    - Usually a length limit (compiler dependent) but long enough to not matter
    - Can't be a reserved word
- C++ is *case sensitive*
    - varName != VARNAME != VarName != varname

Which variable names are valid?

```
int 8pmDinner;
char test-case;
int this_is_a_really_long_variable_name;
float isThisValid;
double wake_up;
char $bob;
double return;
gregWasHere;
```

C++ Reserved Words

asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while

Basic Data Types

The types you might care about:

- **int** - 124, 3, -100
- **float** - 12.4, 45.68, -34.22
- **char** - ‘a’, ‘b’, ‘$’, ‘%’, 128, 7, 254
- **bool** - true, false

Except for **bool**, any of these can be signed or unsigned.

Variable Types (32-bit)

char	character, small integer	1 bytes	signed: -128 to 127 unsigned: 0 to 255
short	short integer	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int / long	integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	boolean value	4 bytes	true or false
float	floating point value	4 bytes	3.4e +/- 38 (7 digits)
double	double precision floating point value	8 bytes	1.7e +/- 308 (15 digits)
wchar_t	wide character	2 bytes	1 wide character

Declaring Variables

- All variables must be declared before they can be used.
- Declarations allocate memory for that variable.

semicolon
every statement ends with one

```
int result = 25;
```

type (implies size)
required

name (identifier)
required

initial value
optional

Declaring Variables

Variables can be declared one per line:

```
int type;
int score = 3;
int aliensKilled;
bool awesome = true;
```

Or, variables of the *same type* can be declared on the same line:

```
int type, score = 3, aliensKilled;
bool awesome = true;
```

Variable Initialization

- Variable initializations are optional...
- What happens if a variable is not initialized with a value?

```
int result = 25;
```

initial value is 25

```
int result;
```

initial value is **what?**

Variable Initialization

- Answer: initial value ends being whatever was in that chunk of memory beforehand
- Probably a garbage value
- C++ compilers do *not* pre-initialize variables!
- Rule of thumb: always initialize variables

```
int result = -19358221;
```

Assigning stuff to variables

- Using the = operator (aka `i = 25.3;`)
- We can assign numeric literals:
  - int types: 3, 0, -42, 167, *not* 1,345,293
  - float types: 2.0, -0.33365f, 3.0e5
  - bool: true or false
- ... or an expression of some sort

Arithmetic Operators

- Assignment (=), as in **`a = 4;`**
- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulo (%)
  - this only works for integers
  - 5 % 3 = 2

a quick note about...

Integer Division

- The result of an integer divide is an integer - the remainder is discarded
  - 5 / 3 = 1
  - what about 3 / 5?
- Division by zero causes a runtime error

More Operators!!!!

As a shortcut for this:

```
aliensKilled = aliensKilled + 10;
```

You can do this:

```
aliensKilled += 10;
```

Operators of this style:

```
+=
-=
*=
/=
%=
```

*Wow! Even **More** Operators!!!1!1!*

Stuff like this happens a lot:

```
numberOfLives = numberOfLives + 1;
```

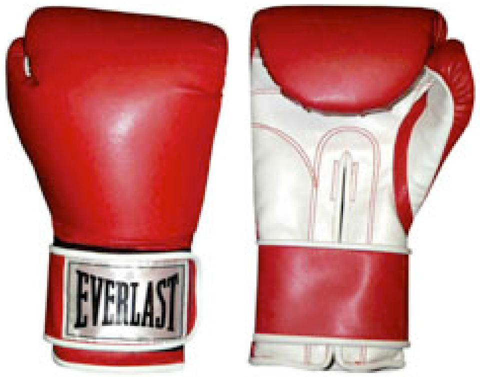
You can do this instead:

```
numberOfLives++;        (post-increment)
        or
++numberOfLives;        (pre-increment)
```

In the above case, the two are equivalent - but they're not always.

Any idea what the difference is?

Pre-Increment vs Post Increment



- **Pre-increment:**
  - first increments the value, then returns it
- **Post-increment:**
  - first returns the value, *then* increments it
  - this involves making a copy of the original value, which is in theory less efficient
  - doesn't matter all that much for built-in types

Pre-increment vs Post-increment

post-increment:

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

int main()
{
    int a = 10;
    cout << a++ << endl;
    return EXIT_SUCCESS;
}
```

pre-increment:

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

int main()
{
    int a = 10;
    cout << ++a << endl;
    return EXIT_SUCCESS;
}
```

There are similar operators for decrementing: aka **`var--`** and **`--var`**

Examples

```
int john = 5, chris = 5, bob = 5;
float perry = 2.5;

john++ * 3;
++chris * 3;

bob / 2;    // doesn't change the
bob % 2;    // value of bob

bob *= 3;
perry %= 2;
```

Conditional Execution

- Most programs don’t unconditionally compute things straight through
- Often we need to decide whether to execute a chunk of code, based on some condition
- Enter conditional statements!

Example

a code snippet...

This code chunk reads in two numbers, and prints out the bigger one.

Note that { and } are used to group blocks of statements.

```
int num1, num2;

// get two numbers from the user
cin >> num1;
cin >> num2;

// compare the numbers
if( num1 > num2 )
{
  // this gets executed if the above
  // condition is true
  cout << num1;
}
else
{
  // and this gets executed if not
  cout << num2;
}
```

Comparison Operators (!!1!)

- Equality: == `if( a == b )`
- Not Equal; != `if( a != b )`
- Greater: > `if( a > b )`
- Less: < `if( a < b )`
- Greater or Equal: >= `if( a >= b )`
- Less or Equal: <= `if( a <= b )`

Boolean Logic:
combining comparisons

And operator: `&&`
Or operator: `||`
Not operator: `!`

Examples:

- `if( (x > 0) && (x < 12) )`
- `if( (x % 2 == 0) || (x < 2) )`
- `if( (x < 3) && !(x < 0) )`

Boolean Logic

```
!true == false
!false == true

(true && true) == true
(true && false) == false
(false && true) == false      ← both must be true
(false && false) == false

(true || true) == true
(true || false) == true
(false || true) == true       ← either can be true
(false || false) == false
```

Operator Precedence

()	left to right
++x; --x	left to right
x++; x--; +x; -x	right to left
*; /; %	left to right
+; -	left to right
<<; >>	left to right
<; <=; >; >=	left to right
==; !=	left to right
&&	left to right
||	left to right
=; +=; -=; *=; /=; %=	right to left

a few quick

Examples

```
int foo = 5, bat = 5;

bat++ * 3 / 2 + 1

foo * 3 % 4 / 2

foo *= 2*2
```

- Tip: just use parenthesis to make your meaning clear

... back to **`if`** statements

- if the condition is true, an **`if`** statement executes the following single statement or block of statements
  - A statement is any valid expression followed by a semicolon
  - A block of statements is anything contained within a set of **`{ }`** brackets

```
if( !milkSmellsBad )
{
   drinkMilk();
}
```

=

```
if( !milkSmellsBad )
   drinkMilk();
```

else statements

- an **`else`** statement is optional; it is executed if the matching **`if`** statement is *not* true
- same rules apply; the statement or block immediately following the else is what gets executed

```
if( jokeIsFunny )
    humor += 10;

else
{
    throwTomatoes();
    humor -= 10;
}
```

fun with `if` and `else`

- you can pile together multiple if/else statements to produce a chain of conditions

```
if( scrubsIsOn )
   watchScrubs();

else if( theOfficeIsOn )
   watchTheOffice();

else if( isNiceDay )
   goOutside();

else
   doHomework();
```

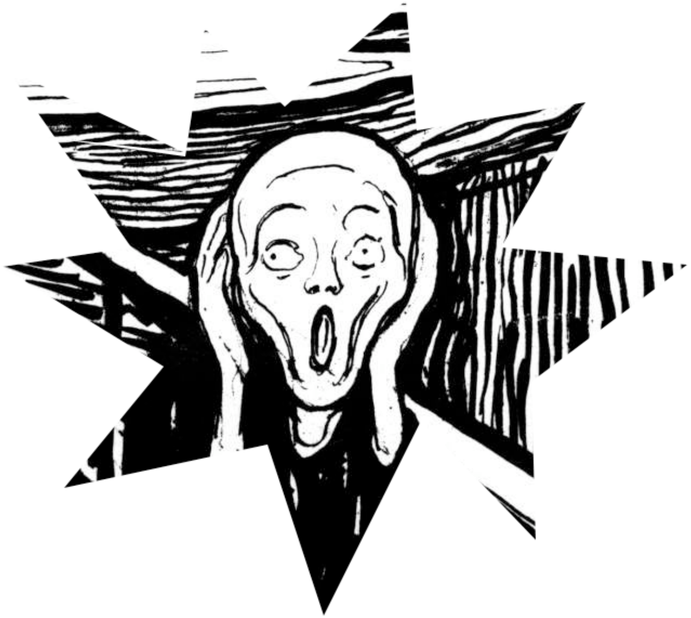
nested `if` statements

if/else statements can be nested in practically any pattern to produce complicated conditional execution

```
if( tornadoSirenIsSounding )
{
   if( !(isFirstMondayOfMonth && is9AM) )
   {
      if( houseHasBasement )
         hideInBasement();
      else
         runAway();

      whimper();
   }
}
```

But be ye careful!



```
if( value == true )
   doThis();
   doThat();
   playCheckers();

watchScrubs();
```

what does this *really* do?

how 'bout this one?

```
if( selfDestructInitiated );
   blowUpShip();
```

Sample Program

- Formula to convert Celsius to Fahrenheit:
  - F = C*1.8 + 32
- Write a program that:
  - Accepts Celsius temperature as input
  - Converts it to Fahrenheit and displays result
  - Classifies the result as too cold, too hot, or just right