System calls from C programs

- Essentially operating system functions
  - So *not portable* across systems
  - But usually more efficient than C functions
- Input-output uses a file descriptor integer
  - `0` - `stdin`, `1` - `stdout`, `2` - `stderr`: always open
  - Basic I/O: `read(int fd, char *buf, int n)` and `write(int fd, char *buf, int n)` – both return `int`
  - For files – use `open`, `creat`, `close`, `unlink`
  - Also `lseek(int fd, long offset, int origin)`

More system calls

- See `~cs60/demo05/basiccopy.c` and `basiccat.c`
- Can also inquire *about* a file with `stat(…)`
  - Fills a struct with same info can get by `ls -l`
  - See `~cs60/demo05/filemode.c` for example
    - Note: also works on Windows, but results may differ
- Note: "`syscalls.h`" (in K&R examples) not standard
- Also note: *if you want to communicate directly with Unix, then chances are good that you should be writing a Unix script instead of a C program!*

C++ – a better C, and more

- Born in 1982 – “Classes: An Abstract Data Type Facility for the C Language”
  - A collection of C macros and library routines by Bjarne Stroustrup, Bell Labs
- Evolved during 1980s to whole new language
  - But always *backward compatible* to C
    - Means any C program is also a C++ program
    - Also means C++ has “legacy problems”
  - Effective use requires abandoning some C features
    - Most notably C’s I/O library, strings, and memory allocation

Well-styled C++ programs … don't look like C programs

- Starts with new style of commenting – `//`
- Includes new way to initialize variables
  - `int x(7), sum();`
  - `MyClass myObject(`*constructor arguments*`);`
  - And a related philosophy – Nagler's PITA rule: "prefer initialization to assignment"
- A Boolean type: `bool cool = true;`
- `main` – must be declared `int` explicitly
  - And automatically returns 0

`using namespace std;` // huh?

- Namespaces – a way to manage global symbols
- To create – `namespace A {` /* *C++ code* */ `}`
- 3 ways to access (hmm … like Java packages):
  - Directly – each time a name is used – with scope resolution operator – `std::cout << data;`
  - Or with a using *declaration* – `using std::cout;`
  - Or access all names in a namespace with a using *directive* – `using namespace std;`
- `std` – *the* namespace for most C++ library tools

#include <iostream>

- Instead of <stdio.h> (actually cstdio now)
  - So no printf, scanf, putchar, FILE *, ...
  - Is legal to mix, but generally frowned upon
  - The good news – iostream is easier to use
    - Except for formatting – see Nagler chapters 14-16
- cout – an ostream object (so is cerr)
  - << – insertion operator, overloaded for many types
- cin – an istream object
  - >> – extraction operator, also works for many types
- See addvalues.cpp in ~cs60/demo05/

const

- Part of an object's type – always enforced in C++
  - `char const *s = "a string";` // must initialize
    - Cannot ever change `s` – same as Java `final` modifier
  - `someFunction(s);` // only okay if const parameter
  - `void someFunction(char const *string) { … }`
    - `string` may not be changed in scope of function
    - So okay to pass in a constant argument
- In fact any string literal ("…") is constant in C++
  - So always must "const qualify" parameters to handle
- Nagler: SCO rule – "support constant objects"

New ways of casting

- C way still works, but discouraged
  - `int x = (int)7.25;` // okay, but not recommended
  - `int x = static_cast<int>(7.25);` // better
- `static_cast` – for equivalent value in different representation
- `reinterpret_cast` – interpret bits differently
  - Very rarely useful
- `const_cast` – to add or take away `const`-ness
  - Do not use lightly – i.e., consider ramifications
- `dynamic_cast` – relates to inheritance (*upcoming*)

Default function arguments

- Can specify parameter values in the function declaration
  - `void func(int x = 12);`
- Then function user can choose to accept the default value(s) or specify new one(s)
  - `func()` // accept default value for x
  - `func(97)` // specify new value for x
- Mandatory arguments are ones without default values – and these *must* come first
  - `void func(int x = 12, int y);` // illegal
  - `void func(int y, int x = 12);` // okay
- Note also must *specify in declaration*, not definition – so compiler is sure to know about default values

Reference variables

- Neither C nor Java has anything like this
- A.k.a. references – actually *aliases – not pointers*
  - `int x, &y = x;` // now y is an alias for x
  - `y = 12;` // now x is 12
- Usefulness comes in with reference parameters
  - `void aFunction(int &value) { value = 12; }`
    - Note how parameter passing is like assignment
  - No need to pass `&x`, nor to use `*value` in function
- See `params.cpp` and `array.cpp` in …`/demo05/`

More reference variables

- Okay to const-qualify a reference:
  - `void display(int const &ref) { … }`
  - Then all of the following are legal calls:
    - `int const var = 2;  display(var);`
    - `display(7);` // an actual constant
    - `display(9 + 3);` // a constant temporary result
- Can also have a reference to a pointer
  - `void initialize(char *&ptr) { … }`
- And can return references from functions
  - `int &oneGreater(int x) { return ++x; }`
  - `int y = ++oneGreater(6); cout << y;` // prints 8

Declaring variables, tags, enums

- Can declare anywhere in block now
  - Even in for loop header: `for (int i = 0; …)`
- Tag name is the type name – no two-word types to declare struct, enum or class objects
  - `struct Foo { … };` // still need it to define type
  - `Foo myFoo;` // not `struct Foo myFoo;`
- The enum constants are exclusive sets
  - No more treating as if just another int
  - `enum Answer {yes, no};`
  - `Answer a = 0;` // illegal – must be `yes` or `no`