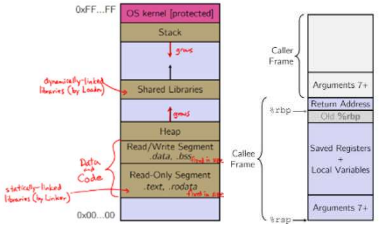
THE CSE 333 – SYSTEMS PROGRAMMING MIDTERM CHEAT SHEET

C Data Type	32-bit	64-bit	printf
char	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	12	16	%Lf
pointer	4	8	%p

malloc vs. new

	malloc()	new
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a void* (should be cast)	appropriate pointer type (doesn't need a cast)
When out of memory	returns NULL	throws an exception
Deallocating	free()	delete or delete[]



A few stdio function prototypes

- fopen(filename, "rb") – open to read bytes
- fgets(buffer, max, FILE* f) – returns NULL if eof or error, otherwise reads a line of up to max-1 characters into buffer, including the \n, and adds a \0 at the end
- fread(buf, 1, count, FILE* f) • fwrite(buf, 1, count, FILE* f)
- fprintf(format_string, data…, FILE *f) • feof(FILE* f) – is eof indicator for f set?
- ferror(FILE* f) – was there an error on f?
- fprintf(stderr, "Error Message\n");

Loading: When the OS loads a program it: 1) Creates an address space 2) Inspects the executable file to see what's in it 3) (Lazily) copies regions of the file into the right place in the address space 4) Does any final linking, relocation, or other needed preparation.

Memory Management: Local variables on the Stack - Allocated and freed via calling conventions (push, pop, mov) . **Global and static variables in Data** - Allocated/freed when the process starts/exits.

Dynamically-allocated data on the Heap - malloc() to request; free() to free, otherwise memory leak.

Pointers: Data types or variables that store addresses, it points to somewhere in the process' virtual address space. For example, &foo produces the virtual address of foo. Generic definition: **type* name;** or **type *name;** Dereference a pointer using the unary * operator. We use them for several things in C, such as: Simulating "pass-by-reference" Using function arguments as return values (also known as "output parameters"). Avoiding copying huge structs when passing arguments into functions. **Faking Call-By-Reference in C:** Can use pointers to approximate call-by-reference. Callee still receives a copy of the pointer (i.e. call-by-value), but it can modify something in the caller's scope by dereferencing the pointer parameter.

Pass Copy of Struct or Pointer? Value passed: passing a pointer is cheaper and takes less space unless struct is small. **Field access:** indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize. For small structs, passing a copy of the struct can be faster and often preferred; for large structs use pointers.

typedef: Generic format: **typedef type name.** The purpose of typedef is to assign alternative names to existing types, most often those whose standard declaration is cumbersome, potentially confusing, or likely to vary from one implementation to another**.** It allows you to define new data type names/synonyms. Both type and name are usable and refer to the same type; Be careful with pointers – * before name is part of type! You can malloc and free structs, just like other data type. *sizeof* (typename) is particularly helpful here.

Compiling the Program: 4 parts: ♣ 1/2) Compile example_ll_customer.c into an object file ♣ 2/1) Compile ll.c into an object file ♣ 3) Link both object files into an executable ♣ 4) Test, Debug, Rinse, Repeat.

External Linkage: *extern* makes a declaration of something externally visible. Every global (variables and functions) is extern by default. Unless you add the static specifier, if some other module uses the same name, you'll end up with a collision! **Internal Linkage:** *static* (in the global context) restricts a definition to visibility within that file. It's good practice to use static to "defend" your global variables. C has a different use for the word "static": to create a persistent local variable. The storage for that variable is allocated when the program loads, in either the .data or .bss segment.

make Basics: ♣ Colon after target is required, ♣ Command lines must start with a TAB, NOT SPACES ♣ Multiple commands for same target are executed in order. • Can split commands over multiple lines by ending lines with '\' ♣ You can define variables in a Makefile: All values are strings of text, no "types" Variable names are case-sensitive and can't contain ':', '#', ' =', or whitespace. ♣ clean is a convention . Removes generated files. ♣ Special variables: $@ for target name ; $^ for all sources ; $< for left-most source.

POSIX – Portable Operating System Interface. open(), read(), write(), close(), lseek().

Templates: ♣ A function or class that accepts a type as a parameter. • You define the function or class once in a type-agnostic way • When you invoke the function or instantiate the class, you specify (one or more) types or values as arguments to it. ♣ At compile-time, the compiler will generate the "specialized" code from your template using the types you provided.• Your template definition is NOT code • Code is only generated if you use your template.

- Template to **compare** two "things":

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(const T &value1, const T &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  return 0;
}
```

functiontemplate.cc

- You can use non-types (constant values) in a template:

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

// prints a value out N times
template <typename T, int N>
int printmultiple(const T &value) {
  for (int i = 0; i < N; ++i) {
    std::cout << value << std::endl;
  }
}

int main(int argc, char **argv) {
  std::string h("hello");
  printmultiple<std::string, 3>(h);
  printmultiple<const char *, 4>("hi");
  printmultiple<int, 5>(10);
  return 0;
}
```

nontypeparameter.cc

Pair Class Definition

Pair.h

```
#ifndef _PAIR_H_
#define _PAIR_H_

template <typename Thing> class Pair {
 public:
  Pair() { };  // default constructor

  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(Thing &copyme);
  void set_second(Thing &copyme);
  void Swap();

 private:
  Thing first_, second_;
};

#include "Pair.cc"

#endif  // _PAIR_H_
```

Pair Function Definitions

Pair.cc

```
template <typename Thing>
void Pair<Thing>::set_first(Thing &copyme) {
  first_ = copyme;
}

template <typename Thing>
void Pair<Thing>::set_second(Thing &copyme) {
  second_ = copyme;
}

template <typename Thing>
void Pair<Thing>::Swap() {
  Thing tmp = first_;
  first_ = second_;
  second_ = tmp;
}

template <typename T>
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", "
             << p.get_second() << ")";
}
```

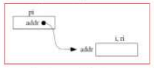
The CSE 333 – Systems Programming Midterm Cheat Sheet

```
int open(const char *name, int mode);
        mode is one of O_RDONLY, O_WRONLY, O_RDWR
int creat(const char *name, int mode);
        create a new file
int close(int fd);
ssize_t read(int fd, void *buffer, size_t count);
        returns # bytes read or 0 (eof) or -1 (error)
ssize_t write(int fd, void *buffer, size_t count);
        returns # bytes written or -1 (error)
```

1) References vs. Pointers (5 min)

a) Consider the following code. Draw the corresponding box-and-arrow diagram.

```
int i;
int *pi = &i;
int &ri = i;
```

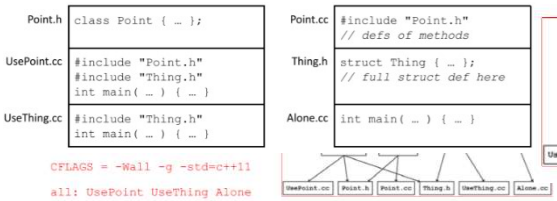


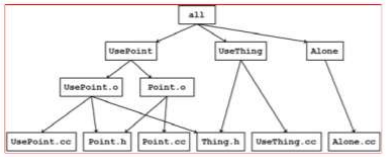
b) What are some key differences between pointers and references?

- References can't be reassigned. Once a reference is created, it cannot be later made to reference another object. Pointers are often reassigned.
- References can't be initialized to null, whereas pointers can.
- References can never be uninitialized or re-initialized.

c) When would it be a good idea to use to references instead of pointers?

- When you don't want to deal with pointer semantics. (hw1, anyone???)
- Doesn't create a copy! (especially for parameters and/or return values)
- Style Guide Tip: use const reference parameters to pass input; use pointers to pass output parameters; input parameters first, then output parameters last.






```
CFLAGS = -Wall -g -std=c++11

all: UsePoint UseThing Alone

UsePoint: UsePoint.o Point.o
        g++ $(CFLAGS) -o UsePoint UsePoint.o Point.o

UsePoint.o: UsePoint.cc Point.h Thing.h
        g++ $(CFLAGS) -c UsePoint.cc

Point.o: Point.cc Point.h
        g++ $(CFLAGS) -c Point.cc

UseThing: UseThing.cc Thing.h
        g++ $(CFLAGS) -o UseThing UseThing.cc

Alone: Alone.cc
        g++ $(CFLAGS) -o Alone Alone.cc

clean:
        rm UsePoint UseThing Alone *.o *~
```

```
#include <stdio.h>   // printf, scanf, fopen, etc.
#include <string.h> // string library
#include <stdlib.h>  // EXIT_SUCCESS, EXIT_FAILURE
// add any additional preprocessor commands you need below
#define BUFSIZE 500   // size of input buffer
#define TRUNC 40      // max # data characters in input line
int main(int argc, char ** argv) {
  FILE * text;        // input text file
  char buf[BUFSIZE];  // input lines

  // verify that there is one command-line argument
  if (argc != 2) {
    fprintf(stderr, "missing file name\n");
    return EXIT_FAILURE;
  }
  // open file and quit if failure
  text = fopen(argv[1], "r");    // "rb" also ok
  if (text == NULL) {
    fprintf(stderr, "unable to open file\n");
    return EXIT_FAILURE;
  }
  // copy & print lines; shorten lines to TRUNC chars if longer
  while (fgets(buf, BUFSIZE, text) != NULL) {
    if (strlen(buf) > TRUNC) {
      buf[TRUNC] = '\n';
      buf[TRUNC+1] = '\0';
    }
    printf("%s", buf);
  }
  // print message if input terminated because of error
  if (ferror(text)) {
    fprintf(stderr, "error on input");
    return EXIT_FAILURE;
  }
  fclose(text);
  return EXIT_SUCCESS;
}
```

```
// multiplication assignment operator in class Str
Str & Str::operator*=(int k) {
  // strategy: create an array large enough to hold copies,
  // then cat that many copies onto an initial empty string
  int ncopies = k<=0 ? 0 : k;
  char * newst = new char[ncopies*strlen(st_) + 1];
  newst[0] = '\0';
  for (int i = 1; i <= ncopies; i++) {
    strcat(newst, st_);
  }
  delete [] st_;
  st_ = newst;
  return *this;
}

// non-member overloaded multiplication operators
Str operator*(int k, const Str &s) {
  Str result(s);
  result *= k;
  return result;
}
Str operator*(const Str &s, int k) {
  Str result(s);
  result *= k;
  return result;
}
```

```
#include <iostream>
using namespace std;
#include "./Vector.h"
namespace vector333 {
// default constructor
Vector::Vector() {
  init(0.0, 0.0, 0.0);
}
// Vector(x,y,z) constructor
Vector::Vector(const float x, const float y, const float z) {
  init(x, y, z);
}
// copy constructor
Vector::Vector(const Vector &other) {
  init(other.v_[0], other.v_[1], other.v_[2]);
}
// private initialization function
// allocate array for vector and initialize with given coordinates
void Vector::init(const float x, const float y, const float z) {
  v_ = new float[3];
  v_[0] = x;
  v_[1] = y;
  v_[2] = z;
}
// destructor - free dynamic storage
Vector::~Vector() {
  delete [] v_;
}
// Vector assignment
Vector &Vector::operator=(const Vector &rhs) {
  // replace state of this with values from rhs; do nothing if
  // self-assignment. (Even though in this particular case there would
  // be no harm, it's always best to check for self-assignment and do
  // nothing if detected.)
  if (this != &rhs) {
    v_[0] = rhs.v_[0];
    v_[1] = rhs.v_[1];
    v_[2] = rhs.v_[2];
  }
  // return reference to lhs of assignment
  return *this;
}
// Updating assignments for vectors
Vector &Vector::operator+=(const Vector &rhs) {
  v_[0] += rhs.v_[0];
  v_[1] += rhs.v_[1];
  v_[2] += rhs.v_[2];
  return *this;
}
Vector &Vector::operator-=(const Vector &rhs) {
  v_[0] -= rhs.v_[0];
  v_[1] -= rhs.v_[1];
  v_[2] -= rhs.v_[2];
  return *this;
}
// Friend functions that are not members of class Vector
// dot-product: if a is (a,b,c) and b is (x,y,z),
// return ax+by+cz
double operator*(const Vector &a, const Vector &b) {
  return a.v_[0]*b.v_[0] + a.v_[1]*b.v_[1] + a.v_[2]*b.v_[2];
}
// scalar multiplication: if v is (a,b,c), return (ak,bk,ck)
Vector operator*(const double k, const Vector &v) {
  return Vector(v.v_[0] * k, v.v_[1] * k, v.v_[2] * k);
}
Vector operator*(const Vector &v, const double k) {
  return Vector(v.v_[0] * k, v.v_[1] * k, v.v_[2] * k);
}
// Stream output: << for Vectors
ostream & operator<<(ostream &out, const Vector &v) {
  out << "(" << v.v_[0] << "," << v.v_[1] << "," << v.v_[2] << ")";
  return out;
}
// Additional non-member functions that are part of the Vector abstraction
// Vector addition
Vector operator+(const Vector &a, const Vector &b) {
  Vector tmp = a;
  tmp += b;
  return tmp;
}
// Vector subtraction
Vector operator-(const Vector &a, const Vector &b) {
  Vector tmp = a;
  tmp -= b;
  return tmp;
}
} // namespace vector333
```

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

namespace vector333 {

// A Vector represents a vector in 3-space.
class Vector {
 public:
  // constructors:

  // Default: construct the vector (0,0,0)
  Vector();

  // Construct the vector (x,y,z)
  Vector(const float x, const float y, const float z);

  // Copy constructor
  Vector(const Vector &v);

  // Destructor
  ~Vector();

  // Assignment
  Vector &operator=(const Vector &rhs);

  // Updating assignment
  Vector &operator+=(const Vector &rhs);
  Vector &operator-=(const Vector &rhs);

  // Additional functions that are not members of Vector but
  // need to be friends so they can access instance variables:

  // dot-product: if a is (a,b,c) and b is (x,y,z),
  // return ax+by+cz
  // (note: ok if result is float; specification wasn't specific)
  friend double operator*(const Vector &a, const Vector &b);
  // scalar multiplication: if v is (a,b,c), return (ak,bk,ck)
  friend Vector operator*(const double k, const Vector &v);
  friend Vector operator*(const Vector &v, const double k);

  // Stream output: define << for Vectors
  friend ostream & operator<<(ostream &out, const Vector &v);

 private:
  // A Vector is represented by a heap-allocated array of three
  // floats giving the x, y, and z magnitudes in v[0], v[1],
  // and v[2] respectively.
  float *v_;

  // private helper function used by constructors: initialize
  // vector state to given x, y, z values.
  void init(const float x, const float y, const float z);
};

// additional operations that are not members or friend functions but
// are part of the Vector abstraction

// addition and subtraction: produce a new Vector that results from
// element-wise addition or subtraction of this with other
Vector operator+(const Vector &a, const Vector &b);
Vector operator-(const Vector &a, const Vector &b);

} // namespace vector333
#endif // _VECTOR_H_
```

One method to `read()` n bytes

```
int fd = open(filename, O_RDONLY);
char* buf = ...;  // buffer of appropriate size
int bytes_left = n;
int result;

while (bytes_left > 0) {
  result = read(fd, buf + (n - bytes_left), bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened, so return an error result
    }
    // EINTR happened, so do nothing and try again
    continue;
  } else if (result == 0) {
    // EOF reached, so stop reading
    break;
  }
  bytes_left -= result;
}
close(fd);
```

prevent infinite loop if EOF reached