CSE 374 Review

```cpp
class Square : public Rectangle {
public:
   Square(); // default
   // constructor, not conversion
   explicit Square(int s);
   // destructor
   virtual ~Square();
};
```

Bank Account ex.

- **Static fields.** A bank account has an associated ID number, which is automatically generated when the account is created. So, we added a static field accountCount_ to the BankAccount class. The "static" keyword means that there is only ONE variable for all objects of this class, not one per object like normal fields. We can use the accountCount_ to generate a unique ID number in the constructor by using that count as the ID and then incrementing the count by 1.
- **Deleted constructors.** Remember C++ automatically generates a "copy constructor" for your class if you do not provide one. However, making copies of the account would be a really bad thing! So we can declare a copy constructor in the header file and set that constructor "= delete;", which means we "delete" it and prevent it from being used anywhere in the code.
- **Pure virtual functions.** A "bank account" is a general concept; if you go into the bank and ask to open a bank account, they'll ask "what kind?" Each type of accounts has a slightly different implementation of withdrawal. All accounts do have the ability to withdraw, but since each type of account has a different implementation, we'll declare the withdraw() function but NOT provide an implementation. We do this as follows by declaring the function "virtual" and setting it equal to 0:

```
virtual void withdraw(int amount) = 0;
```

- This is called a "pure virtual" function, and it make this BankAccount class equivalent to Java's abstract class! Any subclass of BankAccount will have to implement the withdraw() function.

Savings Account

- We provide a constructor that gives more information than just the BankAccount's constructor - a SavingsAccount also generates interest, so we have an interest rate associated with the account.
- The derived class can add additional functions, like getInterestRate().
- The derived class adds an implementation of the withdraw() function from the base class BankAccount. We mark this function "override" so the compiler verifies that we've done the overriding correctly.

(Up) casting

- An object of a derived class *cannot* be cast to an object of a base class.
  - For the same reason a `struct T1 {int x,y,z;}` cannot be cast to type `struct T2 {int x,y;}` (different size)
- A pointer to an object of a derived class *can* be cast to a pointer to an object of a base class.
  - For the same reason a `struct T1*` can be cast to type `struct T2*` (pointers to a location in memory)
  - (Story not so simple with multiple inheritance)
- After such an *upcast*, field-access works fine (prefix)
  - but what do method calls mean in the presence of overriding? (see virtual)

(Down) casting

- C pointer-casts: unchecked; be careful
- Java: checked; may raise ClassCastException
- New: C++ has “all the above” (several different kinds of casts)
  - If you use single-inheritance and know what you are doing, the C-style casts (same pointer, assume more about what is pointed to) should work fine for downcasts
  - Worth learning about the differences on your own

C Datatypes

Variables in C have a type, which defines the size of the memory block and how to decode the memory

Primitive types (numerical)

Derived types (pointers / arrays)

C Datatypes

C11 std: _Bool defines booleans
No Bool in old standards
Use stdbool, or
`typedef enum {false, true} bool;`

Primitive Datatypes:

- Integral types
  - Char (1 byte), short, int (2-4 bytes), long
  - 'Unsigned' removes negative, doubles maximum #
    https://en.wikipedia.org/wiki/Two%27s_complement
- Floats, doubles, long doubles
- Type promotion - moved to a higher precision type / no dataloss. If cast to a lower precision type it is truncated.

Notice: in Bash, variables are untyped

Essentially every variable is a string

Can be cast to a number under some circumstances

C Derived Types (made from primitive types)

Functions types (returns a type)

Pointer types (points to a type)

Array types (lists of a type)

Structure types (contains types)

Union types (holds different types)

(Pointers store an integer # (of size uintptr_t), the type dictates how operations on the pointer behave.

Array types point to the beginning of a list of values, they resolve to a pointer.

Structures

Structures are containers for holding multiple variables together.

Organize data.

Facilitate passing and tracking data.

Have data type 'struct' with a 'tag' name.

```
struct fraction {
    int numerator;
    int denominator;
};
struct fraction weeks_left;
weeks_left.numerator;

struct fraction w1, w2; // declare two fractions
w1.numerator = 2;
w1.denominator = 7;
w2 = w1;  // copy struct

struct fraction *w3;,
(*f3).numerator = 4;  OR  f3->numerator = 4;
Struct fraction *part = w3; // points to same
address
```

typedef

Introduces short-cut or alias to a data type.

```
typedef <type> <name>;

typedef struct fraction {
    int numerator;
    int denominator;
} fraction;
fraction x1;

typedef struct treenode {
    int data;
    struct treenode branches[9];
} TreeNode;

typedef enum {false, true} bool;
```

Dynamic memory allocation

`void* malloc(size_t size)`

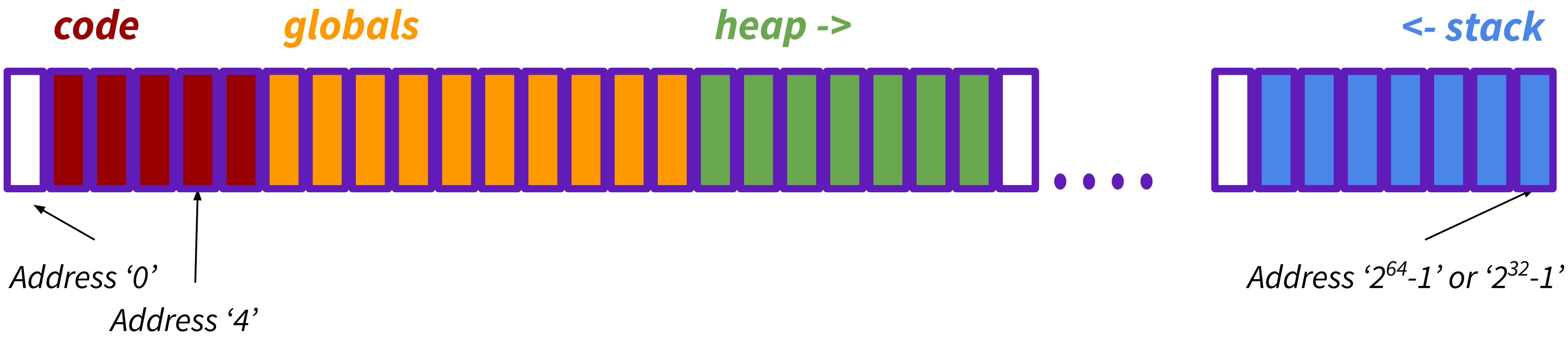
Request a contiguous block of memory of the given size in the heap.

`void free(void* block)`

The mirror image of malloc() -- free takes a pointer to a heap block earlier allocated by malloc() and returns that block to the heap for re-use.

Memory sections

- Code & global memory allocated statically at start up
- Stack memory allocated as functions are called
    - Sizes must be known at compile time
- Heap memory is allocated dynamically (upon request)
    - Sizes can be determined at run time



Using the Heap

- Do this when you don’t know how much space you need in advance
  - Variable length arrays
  - Growing lists or trees
  - Making flexible structs
- Malloc
  - Requires an integer specifying the needed number of bytes (often an uintptr_t)
  - Returns an address (another uintptr_t)
  - Address can be cast to a pointer to a specific type
- Free
  - Takes an address, and returns the chunk to the available memory list

Common problems

- Dangling pointers
  - Happens when you have an active pointer to freed memory
  - Stack memory in a popped frame
  - Freed heap memory
- Accessing a Null pointer
  - Happens when you try to use a pointer before it has been allocated
  - Or when allocation fails
- Forgetting to free memory
  - Causes a memory leak
  - Check struct de-allocation so that every dynamically allocated attribute is also freed
- Losing your pointer
  - Accidentally re-assigning your pointer to a new address
  - Use a current pointer to traverse a tree but be sure to keep a copy of the root unmodified

Multi-file Projects

Modules & Makefiles

C Modules

- Module: smallest coherent unit of a C program
    - One *.c file and *.h file
    - Set of self-contained / closely related functions: clear functionality
    - `gcc -o foo-executable foo.c`
- Project: can be made of many modules
    - `gcc -o fooproject-executable foo.c bar.c`
    - Can also compile individual object files for each module, then link together
- Modules are connected to each other with header (.h) files

Header Files

- Each Module has a .c file and a corresponding .h file
- Header files should always use include guards:
  - `#ifndef MODULE_H #define MODULE_H` and end with: `#endif`
- All declarations needed to use the module are in the header file
- Header file has only declarations and is included in the .c file
- Project-wide variables are declared with extern in header and defined in .c
- Internal declarations go in the .c file, not the header file
  - Declared with static
- Header file should include all other files needed by that header file
- Headers needed only for the .c file to compile go in the .c file
- Header files should compile on their own

Citation:http://www.umich.edu/~eecs381/handouts/CHeaderFileGuidelines.pdf?#:~:text=The%20header%20file%20contains%20only%20declarations%2C%20and%20is%20included%20by%20the%20.&text=Put%20only%20structure%20type%20declarations,c%20file.

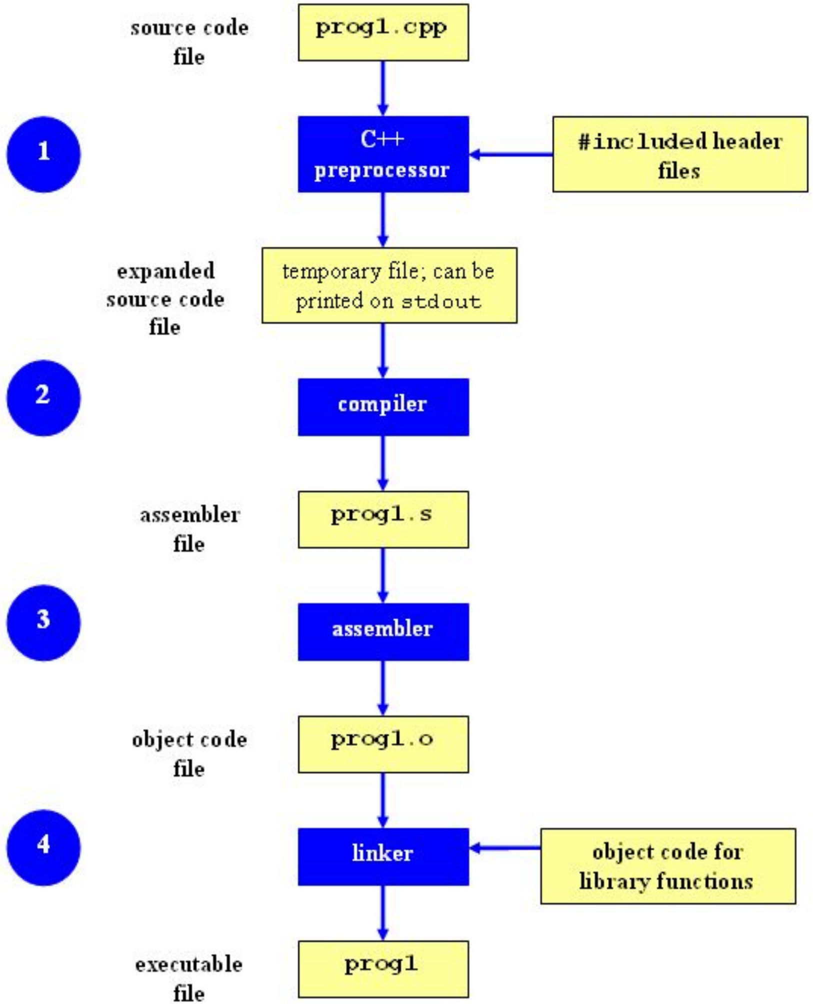
Making Projects

Compiler actually runs in stages:

a. Preprocessor
b. Compiler
c. Assembler
d. Linker

There are other tools to manage this:

- IDEs
- Projects
- Ant



Makefiles

- Figure out dependencies using gcc -MM
- Create targets for each module's object file
  - Using gcc -c
- Create target for project executable
  - Depends on all those object files
- Can also create different targets
  - Different builds
  - Project variations
  - Testing
  - Phony targets (clean)

```
# Makefile for mem memory system

CC = gcc
CARGS = -Wall -std=c11

all: bench

# basic build
bench: bench.o getmem.o freemem.o
getmemstats.o printheap.o
memutils.o
	$(CC) $(CARGS) -o bench $^

# object files
bench.o: bench.c mem.h
	$(CC) $(CARGS) -c bench.c

<... for all the object files …>


debug: CARGS += -g
debug: bench


## Utility targets
test: debug
	./bench 10 50

clean:
	rm *.o *~
```

Miscellaneous Notes

shebang

`#!` - called ‘shebang’

Use it in the ***first line*** of a script to indicate which program should be used to run

`#!/bin/bash`

Note: for our scripts ‘bash’ is the default program to run, so it this is missing they will still be executed by bash. However, specifying /bin/bash will ensure that bash is always used, even if the script is called from a different shell.

Clint.py has `#!/usr/bin/python` in the first line

Specifies that python is the program that is used to run the script

Change to `#!/usr/bin/python2` if you are having trouble using clint on a computer running python3.