Lectures 6 & 7: C++ structs and classes
(Still) Moving Some Content To Later Lectures

• Enum / union (for 2D) ← doing today
• Memory errors (for 4A) ← not doing today
Projects

• Hopefully you are done with 3B
• Assigned this week: 2D
• 3C (due 11/15) will take this lecture and next to set up
• Timeline for rest of term:
  – 3C: 11/15
  – 3D (optional): 11/15
  – 3E: 11/22 (a thinker, not much code)
  – 3F: 11/22 or 11/29
  – 3G: 11/29
  – 3H+4B: 12/8
  – Weds Dec 8 @ 10am: Hank does live code of project 3
Reading

• No new reading
Stuff for 2D
Enums

• Enums make your own type
  – Type is “list of key words”

• Enums are useful for code clarity
  – Always possible to do the same thing with integers

• Be careful with enums
  – … you can “contaminate” a bunch of useful words
C keyword “enum” – means enum definition is coming

```
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};
```

This enum contains 6 different student types

semi-colon!!!
enum example

int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == GradStudent)
        return 26;

    return -1;
}
enums translate to integers ... and you can set their range

128-223-223-72-wireless:330  hank$  cat enum2.c
#include <stdio.h>

enum StudentType
{
    HighSchool = 105,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};

int main()
{
    printf("HighSchool = %d, GradStudent = %d\n", HighSchool, GradStudent);
}
128-223-223-72-wireless:330  hank$  gcc enum2.c
128-223-223-72-wireless:330  hank$  ./a.out
HighSchool = 105, GradStudent = 110
But enums can be easier to maintain than integers

```c
enum StudentType
{
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    PostBacc,
    GradStudent
};

int AverageAge(enum StudentType st)
{
    if (st == HighSchool) return 16;
    if (st == Freshman) return 18;
    if (st == Sophomore) return 19;
    if (st == Junior) return 21;
    if (st == Senior) return 23;
    if (st == PostBacc) return 24;
    if (st == GradStudent) return 26;

    return -1;
}
```

If you had used integers, then this is a bigger change and likely to lead to bugs.
Unions

• Union: special data type
  – store many different memory types in one memory location

```c
typedef union
{
    float x;
    int   y;
    char  z[4];
} cis330_union;
```

This data structure has 4 bytes

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.
Why are unions useful?

```c
#include <stdio.h>

typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;

int main()
{
    cis330_union u;
    u.x = 3.5; /* u.x is 3.5, u.y and u.z are not meaningful */
    u.y = 3;  /* u.y is 3, now u.x and u.z are not meaningful */
    printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}

128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
```
Unions Example

typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;
typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    char *carMake;
    char *carModel;
    US_State state;
    LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}

typedef enum
{
    CA,
    OR,
    WY
} US_State;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;
Building Large Projects
More traditional file organization

Methods can be defined outside the struct definition. They use C++’s namespace concept, which is automatically in place. (e.g., TallyCounter::IncrementCount)

Why do we use this organization? Will discuss in great detail (20 slides) now
More traditional file organization

• struct definition is in .h file
  – #ifndef / #define
• method definitions in .C file
• driver file includes headers for all structs it needs
struct Rectangle;
void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4);

struct Rectangle {
    double minX, maxX, minY, maxY;
};

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4) {
    r->minX = v1;  r->maxX = v2;  r->minY = v3;  r->maxY = v4;
}

#include <prototypes.h>

int main() {
    struct Rectangle r;
    InitializeRectangle(r, 0, 1, 0, 1.5);
}
proj2B: rectangle.o driver.o
   gcc -o proj2B driver.o rectangle.o

driver.o: prototypes.h driver.c
   gcc -I -c driver.c

rectangle.o: prototypes.h rectangle.c
   gcc -I -c rectangle.c
Definition of Rectangle in rectangle.c

Why is this a problem?

prototypes.h

```
struct Rectangle;
void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4);
```

debugger.c

```
#include <prototypes.h>

int main()
{
    struct Rectangle r;
    InitializeRectangle(&r, 0, 1, 0, 1.5);
}
```

“gcc -c driver.c” needs to make an object file. It needs info about Rectangle then, not later.
The fix is to make sure driver.c has access to the Rectangle struct definition.

```
#include <prototypes.h>

int main()
{
    struct Rectangle r;
    InitializeRectangle(r, 0, 1, 0, 1.5);
}
```

gcc –E shows what the compiler sees after satisfying “preprocessing”, which includes steps like “#include”.

```
# 1 "driver.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 162 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "driver.c" 2
# 1 "./prototypes.h" 1

struct Rectangle;

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4);

int main()
{
    struct Rectangle r;
    InitializeRectangle(r, 0, 1, 0, 1.5);
}
```

This is it. If the compiler can’t figure out how to make object file with this, then it has to give up.
What is the problem with this configuration?
Compilation error

C02LN00GFD58:project hank$ make
gcc -I. -c rectangle.c
In file included from rectangle.c:2:
In file included from ./prototypes.h:2:
./struct.h:2:8: error: redefinition of 'Rectangle'
struct Rectangle
^
./struct.h:2:8: note: previous definition is here
struct Rectangle
^
1 error generated.
make: *** [rectangle.o] Error 1
gcc -E rectangle.c

```c
C02LN00GF58:project hank$ gcc -E -I. rectangle.c
# 1 "rectangle.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 162 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "rectangle.c" 2
# 1 "/struct.h" 1

struct Rectangle {
    double minX, maxX, minY, maxY;
};
# 2 "rectangle.c" 2
# 1 "/prototypes.h" 1
# 1 "/struct.h" 1

struct Rectangle {
    double minX, maxX, minY, maxY;
};
# 3 "/prototypes.h" 2

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4);
# 3 "rectangle.c" 2

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4) {
    r->minX = v1;
    r->maxX = v2;
    r->minY = v3;
    r->maxY = v4;
}
```
How to fix?

• Solution #1: don’t include it twice
  – → Turns out that is hard

• Solution #2: need more infrastructure – macros
  – (This motivates the next ten slides)
Preprocessor

• Preprocessor:
  – takes an input program
  – produces another program (which is then compiled)

• C has a separate language for preprocessing
  – Different syntax than C
  – Uses macros ("#")

macro ("macroinstruction"): rule for replacing input characters with output characters
Preprocessor Phases

• Resolve #includes
  – (we understand #include phase)
• Conditional compilation (#ifdef)
• Macro replacement
• Special macros
This is an example of macro replacement.
#define via gcc command-line option

```c
int main()
{
    return RV;
}
```

```
C02LN00GFD58:330 hank$ cat defines.c
C02LN00GFD58:330 hank$ gcc -DRV=4 defines.c
C02LN00GFD58:330 hank$ ./a.out
C02LN00GFD58:330 hank$ echo $?
4
```
Conflicting –D and #define

```
C02LN00GFD58:330 hank$ cat defines.c
#define RV 2
int main()
{
    return RV;
}
C02LN00GFD58:330 hank$ gcc -DRV=4 defines.c
defines.c:1:9: warning: 'RV' macro redefined
#define RV 2
^<command line>:1:9: note: previous definition is here
#define RV 4
^1 warning generated.
C02LN00GFD58:330 hank$ ./a.out
C02LN00GFD58:330 hank$ echo $? 2
```
C02LN00GFD58:330 hank$ cat conditional.c
#define USE_OPTION 1

int main()
{
    DoMainCode();
    #ifdef USE_OPTION
        UseOption();
    #endif
    DoCleanupCode();
}
Conditional compilation controlled via compiler flags

```
#include <stdio.h>

int main()
{
    #ifdef DO_PRINTF
        printf("I am doing PRINTF!!\n");
    #endif
}
```

```
$ gcc conditional_printf.c
$ ./a.out
I am doing PRINTF!!
```

This is how configure/cmake controls the compilation.
4 files: struct.h, prototypes.h, rectangle.c, driver.c

```
#include <struct.h>

struct Rectangle
{
    double minX, maxX, minY, maxY;
};

#include <prototypes.h>

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4);

#include <struct.h>
#include <prototypes.h>

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4)
{
    r->minX = v1;  r->maxX = v2;  r->minY = v3;  r->maxY = v4;
}

#include <struct.h>
#include <prototypes.h>

int main()
{
    struct Rectangle r;
    InitializeRectangle(&r, 0, 1, 0, 1.5);
}
```

```
What is the problem with this configuration?
```
Compilation error

C02LN00GFD58:project hank$ make
gcc -I. -c rectangle.c
In file included from rectangle.c:2:
In file included from ./prototypes.h:2:
./struct.h:2:8: error: redefinition of 'Rectangle'
  struct Rectangle
  
  ./struct.h:2:8: note: previous definition is here
  struct Rectangle
  
1 error generated.
maker: *** [rectangle.o] Error 1
gcc –E rectangle.c

C02LNO00GFD58:project hank$ gcc -E -I. rectangle.c
# 1 "rectangle.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 162 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "rectangle.c" 2
# 1 "./struct.h" 1

struct Rectangle
{
    double minX, maxX, minY, maxY;
};
# 2 "rectangle.c" 2
# 1 "./prototypes.h" 1
# 1 "./struct.h" 1

struct Rectangle
{
    double minX, maxX, minY, maxY;
};
# 3 "./prototypes.h" 2

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4);
# 3 "rectangle.c" 2

void InitializeRectangle(struct Rectangle *r, double v1, double v2, double v3, double v4)
{
    r->minX = v1;
    r->maxX = v2;
    r->minY = v3;
    r->maxY = v4;
}
#ifndef
#define to the rescue

#ifndef RECTANGLE_330
#define RECTANGLE_330

struct Rectangle
{
    double minX, maxX, minY, maxY;
};

#endif

Why does this work?

This problem comes up a lot with big projects, and especially with C++.
There is more to macros...

- Macros are powerful & can be used to generate custom code.
  - Beyond what we will do here.

- Two special macros that are useful:
  - __FILE__ and __LINE__

```c
#include <stdio.h>

int main()
{
    printf("This print happens on line %d of file %s\n", __LINE__, __FILE__);
    printf("But this print happens on line %d\n", __LINE__);
}
```

(Do an example with __LINE__, __FILE__)
(done with motivating file layout, now want to introduce some new C++ stuff)
Inline function

• inlined functions:
  – hint to a compiler that can improve performance
  – basic idea: don’t actually make this be a separate function that is called
    • Instead, just pull the code out of it and place it inside the current function
  – new keyword: inline

```c
inline int doubler(int X)
{
    return 2*X;
}

int main()
{
    int Y = 4;
    int Z = doubler(Y);
}
```

The compiler sometimes refuses your inline request (when it thinks inlining won’t improve performance), but it does it silently.
Inlines can be automatically done within class definitions

- Even though you don’t declare this as inline, the compiler treats it as an inline

```cpp
class MyDoublerClass
{
    int doubler(int X) { return 2*X; }
};
```
You should only do inlines within class definitions

Left: function is inlined in every .C that includes it ... no problem
Right: function is defined in every .C that includes it ... duplicate symbols
“this”: pointer to current object

• From within any struct’s method, you can refer to the current object using “this”

```cpp
TallyCounter::TallyCounter(int c)
{
    count = c;
}

this->count = c;
```
C++ memory management

• C++ provides new constructs for requesting heap memory from the memory manager
  – stack memory management is not changed
    • (automatic before, automatic now)
• Allocate memory: “new”
• Deallocate memory: “delete”
new / delete syntax

```c
int main()
{
    int *oneInt = new int;
    *oneInt = 3;
    int *intArray = new int[3];

    delete oneInt;
    delete [] intArray;
}
```

- No header necessary
- Allocating array and single value is the same.
- Deleting array takes [], deleting single value doesn’t.
- new knows the type and allocates the right amount.
  - new int → 4 bytes
  - new int[3] → 12 bytes
new calls constructors for your classes

- Declare variable in the stack: constructor called
- Declare variable with “malloc”: constructor not called
  - C knows nothing about C++!
- Declare variable with “new”: constructor called
Destructors

- A destructor is called automatically when an object goes out of scope (via stack or delete)
- A destructor’s job is to clean up before the object disappears
  - Deleting memory
  - Other cleanup (e.g., linked lists)
- Same naming convention as a constructor, but with a prepended ~ (tilde)
Destructors example

```cpp
struct Pixel
{
    unsigned char R, G, B;
};

class Image
{
    public:
        Image(int w, int h);
        ~Image();

    private:
        int width, height;
        Pixel *buffer;
};

Image::Image(int w, int h)
{
    width = w; height = h;
    buffer = new Pixel[width*height];
}

Image::~Image()
{
    delete [] buffer;
}
```

Class name with ~ prepended

Defined like any other method, does cleanup

If Pixel had a constructor or destructor, it would be getting called (a bunch) by the new’s and delete’s.
3 big changes to structs in C++

1) You can associate “methods” (functions) with structs

2) You can control access to data members and methods
Access Control

• New keywords: public and private
  – public: accessible outside the struct
  – private: accessible only inside the struct
  • Also “protected” ... we will talk about that later

```c
struct TallyCounter
{
    private:
    int count;

    public:
    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);
    void Reset();
    int GetCount();
    void IncrementCount();
};
```

Everything following is private. Only will change when new access control keyword is encountered.

Everything following is now public. Only will change when new access control keyword is encountered.
You can issue public and private as many times as you wish…
The compiler prevents violations of access controls.

```c
#include <stdio.h>
#include <TallyCounter.h>

int main()
{
    TallyCounter tc;
    tc.count = 10;
}
```

```
128-223-223-72-wireless:TC hank$ cat main.C
#include <stdio.h>
#include <TallyCounter.h>

int main()
{
    TallyCounter tc;
    tc.count = 10;
}
128-223-223-72-wireless:TC hank$ make
g++ -I. -c main.C
main.C:7:8: error: 'count' is a private member of 'TallyCounter'
    tc.count = 10;
^
./TallyCounter.h:12:12: note: declared private here
    int    count;
    ^
1 error generated.
make: *** [main.o] Error 1
```
The friend keyword can override access controls

```cpp
struct TallyCounter
{
    friend int main();

public:
    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);

private:
    int count;
}
```

- Note that the struct declares who its friends are, not vice-versa
  - You can’t declare yourself a friend and start accessing data members

- friend is used most often to allow objects to access other objects

This will compile, since main now has access to the private data member “count”
class vs struct

• class is new keyword in C++

• classes are very similar to structs
  – the only differences are in access control
    • primary difference: struct has public access by default, class has private access by default

• Almost all C++ developers use classes and not structs
  – C++ developers tend to use structs when they want to collect data types together (i.e., C-style usage)
  – C++ developers use classes for objects … which is most of the time

You should use classes!
Even though there isn’t much difference …
3 big changes to structs in C++

1) You can associate “methods” (functions) with structs
2) You can control access to data members and methods
3) Inheritance
Simple inheritance example

```
struct A {
    int x;
};

struct B : A {
    int y;
};

int main() {
    B b;
    b.x = 3;
    b.y = 4;
}
```

• Terminology
  - B inherits from A
  - A is a base type for B
  - B is a derived type of A

• Noteworthy
  - “:” (during struct definition) → inherits from
    - Everything from A is accessible in B
      - (b.x is valid!!)
Object sizes

#include <stdio.h>

struct A
{
    int x;
};

struct B : A
{
    int y;
};

int main()
{
    B b;
    b.x = 3;
    b.y = 4;
    printf("Size of A = %lu, size of B = %lu\n", sizeof(A), sizeof(B));
}

128-223-223-72-wireless:330 hank$ g++ simple_inheritance.C
128-223-223-72-wireless:330 hank$ ./a.out
Size of A = 4, size of B = 8
struct TallyCounter
{
    friend    int main();

    public:
        TallyCounter(void);
        TallyCounter(int c);
        TallyCounter(TallyCounter &);

    private:
        int    count;

    public:
        void    Reset();
        int    GetCount();
        void    IncrementCount();
};

struct FancyTallyCounter : TallyCounter
{
    void    DecrementCount() { count--; }
}
Virtual functions

• Virtual function: function defined in the base type, but can be re-defined in derived type.
• When you call a virtual function, you get the version defined by the derived type
Virtual functions: example

```c
#include <stdio.h>

struct SimpleID
{
    int id;
    virtual int GetIdentifier() { return id; };
};

struct ComplexID : SimpleID
{
    int extraId;
    virtual int GetIdentifier() { return extraId*128+id; };
};

int main()
{
    ComplexID cid;
    cid.id = 3;
    cid.extraId = 3;
    printf("ID = %d\n", cid.GetIdentifier());
}
```

```
128-223-223-72-wireless:330 hank$ g++ virtual.C
128-223-223-72-wireless:330 hank$ ./a.out
ID = 387
```
Virtual functions: example

You get the method furthest down in the inheritance hierarchy

```c
#include <stdio.h>

struct SimpleID
{
    int id;
    virtual int GetIdentifier() { return id; };
};

struct ComplexID : SimpleID
{
    int extraId;
    virtual int GetIdentifier() { return extraId*128+id; };
};

struct C3 : ComplexID
{
    int extraExtraId;
};

int main()
{
    C3 cid;
    cid.id = 3;
    cid.extraId = 3;
    cid.extraExtraId = 4;
    printf("ID = %d\n", cid.GetIdentifier());
}
```
Virtual functions: example

You can specify the method you want to call by specifying it explicitly
The “is a” test

• Inheritance should be used when the “is a” test is true
• Base class: Shape
• Derived types: Triangle, Rectangle, Circle
  – A triangle “is a” shape
  – A rectangle “is a” shape
  – A circle “is a” shape

You can define an interface for Shapes, and the derived types can fill out that interface.
Now Let’s Consider More Topics with Inheritance / Virtual Functions...

- Access control
- Constructors/destructors
- Multiple inheritance
- Diamond-shape inheritance
- Pure virtual functions (+ abstract/concrete types)
Now Let’s Consider More Topics with Inheritance / Virtual Functions...

- Access control
- Constructors/destructors
- Multiple inheritance
- Diamond-shape inheritance
- Pure virtual functions (+ abstract/concrete types)
Access controls and inheritance

C02LN00GFD58:330 hank$ cat inheritance.C
struct A { int x; };
struct B : A { int y; };
struct C : public A { int y; };
struct D : private A { int y; };

int main()
{
    C c;
    c.x = 2;
    D d;
    d.x = 2;
}

B and C are the same. public is the default inheritance for structs

Public inheritance ("is a"): derived types gets access to base type’s data members and methods

Private inheritance ("has a"): derived types don’t get access. Rarely used and often composition is used instead. Key difference: you can override virtual functions.
One more access control word: protected

- Protected means:
  - It cannot be accessed outside the object
    - Modulo “friend”
  - But it can be accessed by derived types
    - (assuming public inheritance)
Public, private, protected

<table>
<thead>
<tr>
<th>Accessed by derived types*</th>
<th>Accessed outside object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Yes</td>
</tr>
<tr>
<td>Protected</td>
<td>Yes</td>
</tr>
<tr>
<td>Private</td>
<td>No</td>
</tr>
</tbody>
</table>

* = with public inheritance
Now Let’s Consider More Topics with Inheritance / Virtual Functions...

- Access control
- Constructors/destructors
- Multiple inheritance
- Diamond-shape inheritance
- Pure virtual functions (+ abstract/concrete types)
Inheritance and Constructors/Destructors: Example

• Constructors from base class called first, then next derived type second, and so on.

• Destructor from base class called last, then next derived type second to last, and so on.

• Derived type always assumes base class exists and is set up
  – ... base class never needs to know anything about derived types
Inheritance and Constructors/Destructors: Example

```c
#include <stdio.h>

class C
{
    public:
        C() { printf("Constructing C\n"); }
        ~C() { printf("Destructing C\n"); }
    
};
class D : public C
{
    public:
        D() { printf("Constructing D\n"); }
        ~D() { printf("Destructing D\n"); }
    
};

int main()
{
    printf("Making a D\n");
    
    D b;

    printf("Making another D\n");
    
    D b;
}
```
Possible to get the wrong destructor

• With a constructor, you always know what type you are constructing.
• With a destructor, you don’t always know what type you are destructing.
• This can sometimes lead to the wrong destructor getting called.
#include <stdio.h>

class C {
    public:
        C() { printf("Constructing C\n"); }
        ~C() { printf("Destructing C\n"); }
};

class D : public C {
    public:
        D() { printf("Constructing D\n"); }
        ~D() { printf("Destructing D\n"); }
};

D* D_as_D_Creator() { return new D; }
C* D_as_C_Creator() { return new D; }

int main()
{
    C* c = D_as_C_Creator();
    D* d = D_as_D_Creator();

    delete c;
    delete d;
}

fawcett:330 child$ ./a.out
Constructing C
Constructing D
Constructing C
Constructing D
Destructing C
Destructing D
Destructing C
Virtual destructors

• Solution to this problem:
  Make the destructor be declared virtual
• Then existing infrastructure will solve the problem
  – ... this is what virtual functions do!
Virtual destructors

#include <stdio.h>

class C {
  public:
    C() { printf("Constructing C\n"); }
    virtual ~C() { printf("Destructing C\n"); }
};

class D : public C {
  public:
    D() { printf("Constructing D\n"); }
    virtual ~D() { printf("Destructing D\n"); }
};

D* D_as_D_Creator() { return new D; }
C* D_as_C_Creator() { return new D; }

int main()
{
  C* c = D_as_C_Creator();
  D* d = D_as_D_Creator();

  delete c;
  delete d;
}

fawcett:330 childs$ ./a.out
Constructing C
Constructing D
Constructing C
Constructing D
Destructing D
Destructing C
Destructing D
Destructing C
Destructing C
Now Let’s Consider More Topics with Inheritance / Virtual Functions...

• Access control
• Constructors/destructors
• Multiple inheritance
• Diamond-shape inheritance
• Pure virtual functions (+ abstract/concrete types)
Multiple inheritance

• A class can inherit from more than one base type
• This happens when it “is a” for each of the base types
  – Inherits data members and methods of both base types
Multiple inheritance

class Professor
{
    void Teach();
    void Grade();
    void Research();
};

class Father
{
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor
{
};
Now Let’s Consider More Topics with Inheritance / Virtual Functions...

• Access control
• Constructors/destructors
• Multiple inheritance
• Diamond-shape inheritance
• Pure virtual functions (+ abstract/concrete types)
Diamond-Shaped Inheritance

• Base A, has derived types B and C, and D inherits from both B and C.
  – Which A is D dealing with??

• Diamond-shaped inheritance is controversial & really only for experts
  – (For what it is worth, we make heavy use of diamond-shaped inheritance in my project)
Diamond-Shaped Inheritance Example

```java
class Person {
    int X;
};

class Professor : public Person {
    void Teach();
    void Grade();
    void Research();
};

class Father : public Person {
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor {
};
```
Diamond-Shaped Inheritance Pitfalls

```c
#include <stdio.h>

class Person {
    public:
        Person(int h) { hoursPerWeek = h; }
    protected:
        int hoursPerWeek;
};

class Professor : public Person {
    public:
        Professor(int h) : Person(90) {
        void Teach();
        void Grade();
};

class Hank : public Father, public Professor {
    public:
        int GetHoursPerWeek() { return hoursPerWeek; }
    }
}

int main()
{
    Hank hrc;
    printf("HPW = %d\n", hrc.GetHoursPerWeek());
}
```
Diamond-Shaped Inheritance Pitfalls

```c
#include <stdio.h>

class Person {
  public:
    Person(int h) { hoursPerWeek = h; };
  protected:
    int hoursPerWeek;
};

class Professor : public Person {
  public:
    Professor() : Person(90) { ; };
    void Teach();
    void Grade();
    void Research();
};

class Father : public Person {
  public:
    Father() : Person(20) { ; };
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor {
  public:
    int GetHoursPerWeek() { return Professor::hoursPerWeek + Father::hoursPerWeek; };
  }

ing main()
{
  Hank hrc;
  printf("HPW = %d\n", hrc.GetHoursPerWeek());
}
```

This can get stickier with virtual functions

You should avoid diamond-shaped inheritance until you feel really comfortable with OOP.

```
fawcett:330 child$ ./a.out
HPW = 110
```
Now Let’s Consider More Topics with Inheritance / Virtual Functions...

• Access control
• Constructors/destructors
• Multiple inheritance
• Diamond-shape inheritance
• Pure virtual functions (+ abstract/concrete types)
Pure Virtual Functions

• Pure Virtual Function: define a function to be part of the interface for a class, but do not provide a definition
  • Syntax: add “=0” after the function definition
  • This makes the class be “abstract”
    – It cannot be instantiated
• When derived types define the function, then are “concrete”
  – They can be instantiated
Pure Virtual Functions Example

class Shape
{
    public:
        virtual double GetArea(void) = 0;
};

class Rectangle : public Shape
{
    public:
        virtual double GetArea() { return 4; }
};

int main()
{
    Shape s;
    Rectangle r;
}

fawcett:330 childs$ g++ pure_virtual.C
pure_virtual.C: In function ‘int main()’:
pure_virtual.C:15: error: cannot declare variable ‘s’ to be of abstract type ‘Shape’
pure_virtual.C:2: note: because the following virtual functions are pure within ‘Shape’:
pure_virtual.C:4: note:    virtual double Shape::GetArea()
Two Additional C++ Topics

- Objects in Objects
- Initializers
Two Additional C++ Topics

• Objects in Objects
• Initializers
Objects in objects

```c++
#include <stdio.h>

class A {
    public:
        A() { printf("Constructing A\n"); }
        ~A() { printf("Destructing A\n"); }
};

class B {
    public:
        B() { printf("Constructing B\n"); }
        ~B() { printf("Destructing B\n"); }
    private:
        A a1, a2;
};

int main() {
    printf("Making a B\n");
    B b;
    printf("Making another B\n");
    B b;
}
```

By the time you enter B’s constructor, a1 and a2 are already valid.
Objects in objects

#include <stdio.h>

class A {
    public:
    A() { printf("Constructing A\n"); }
    ~A() { printf("Destructing A\n"); }
};

class B {
    public:
    B() { printf("Constructing B\n"); }
    ~B() { printf("Destructing B\n"); }
};

class C {
    public:
    C() { printf("Constructing C\n"); }
    ~C() { printf("Destructing C\n"); }
    private:
    A a;
    B b;
};

int main()
{
    C c;
}
#include <stdio.h>

class A
{
  public:
    A() { printf("Constructing A\n"); }
    ~A() { printf("Destructing A\n"); }
};

class B
{
  public:
    B() { printf("Constructing B\n"); }
    ~B() { printf("Destructing B\n"); }
};

class C
{
  public:
    C() { printf("Constructing C\n"); }
    ~C() { printf("Destructing C\n"); }
  private:
    B b;
    A a;
};

int main()
{
  C c;
}
Two Additional C++ Topics

- Objects in Objects
- Initializers
Initializers

- New syntax to have variables initialized before even entering the constructor

```c
#include <stdio.h>

class A
{
    public:
        A() : x(5)
        {
            printf("x is %d\n", x);
        }
    private:
        int x;
};

int main()
{
    A a;
}
```

fawcett:330 childs$ ./a.out
x is 5
Initializers

• Initializers are a mechanism to have a constructor pass arguments to another constructor

• Needed because
  – Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class
  – Constructors for objects contained in a class are called before the container class & need to pass arguments in container class’s destructor
Initializers

• Needed because
  – Constructors for objects contained in a class are called before the container class & need to pass arguments in container class’s destructor

```c
#include <stdio.h>

class A
{
    public:
        A(int x) { v = x; }
    private:
        int v;
};
class B
{
    public:
        B(int x) { v = x; }
    private:
        int v;
};
class C
{
    public:
        C(int x, int y) : b(x), a(y) { }
    private:
        B b;
        A a;
};

int main()
{
    C c(3,5);
}
```
Initializers

- Needed because
  - Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class

```cpp
class A
{
    public:
        A(int x) { v = x; }
    private:
        int v;
};
class C : public A
{
    public:
        C(int x, int y) : A(y), z(x) { }
    private:
        int z;
};
int main()
{
    C c(3,5);
}
```
#include <stdio.h>

int doubler(int X)
{
    printf("In doubler\n");
    return 2*X;
}

class A
{
    public:
        A(int x) { printf("In A's constructor\n"); }
};

class B : public A
{
    public:
        B(int x) : A(doubler(x)) { printf("In B's constructor\n"); }
};

int main()
{
    B b(3);
}

What’s the output?
Data Flow Networks

• This is not a C++ idea
• It is used for image processing, visualization, etc
• So we need to know it for Project 3
Data Flow Overview

• Basic idea:
  – You have many modules
    • Hundreds!!
  – You compose modules together to perform some desired functionality

• Advantages:
  – Customizability
  – Design fosters interoperability between modules to the extent possible
Data Flow Overview

• Participants:
  – Source: a module that produces data
    • It creates an output
  – Sink: a module that consumes data
    • It operates on an input
  – Filter: a module that transforms input data to create output data

• Nominal inheritance hierarchy:
  – A filter “is a” source
  – A filter “is a” sink
Example of data flow (image processing)

• Sources:
  – FileReader: reader from file
  – Color: generate image with one color

• Filters:
  – Crop: crop image, leaving only a sub-portion
  – Transpose: view image as a 2D matrix and transpose it
  – Invert: invert colors
  – Concatenate: paste two images together

• Sinks:
  – FileWriter: write to file
Example of data flow (image processing)
Example of data flow (image processing)

- **Participants:**
  - **Source:** a module that produces data
    - It creates an output
  - **Sink:** a module that consumes data
    - It operates on an input
  - **Filter:** a module that transforms input data to create output data

- **Pipeline:** a collection of sources, filters, and sinks connected together
Benefits of the Data Flow Design

• Extensible!
  – write infrastructure that knows about abstract types (source, sink, filter, and data object)
  – write as many derived types as you want

• Composable!
  – combine filters, sources and sinks in custom configurations
Drawbacks of Data Flow Design

What do you think the drawbacks are?

• Operations happen in stages
  – Extra memory needed for intermediate results
  – Not cache efficient

• Compartmentalization can limit possible optimizations
Data Flow Networks

• Idea:
  – Many modules that manipulate data
    • Called filters
  – Dynamically compose filters together to create “networks” that do useful things
  – Instances of networks are also called “pipelines”
    • Data flows through pipelines
  – There are multiple techniques to make a network “execute” … we won’t worry about those yet
Data Flow Network: the players

• **Source**: produces data
• **Sink**: accepts data
  – Never modifies the data it accepts, since that data might be used elsewhere
• **Filter**: accepts data and produces data
  – A filter “is a” sink and it “is a” source

Source, Sink, and Filter are abstract types. The code associated with them facilitates the data flow.

There are concrete types derived from them, and they do the real work (and don’t need to worry about data flow!).
3C assignment: make your code base be data flow networks with OOP

- Source
- Filter
- Sink
- PNMreader
- LRConcat
- Shrinker
- TBCOncat
- Blender
- PNMwriter
C++ lets you define operators

• You declare a method that uses an operator in conjunction with a class
  – +, -, /, !, ++, etc.

• You can then use your operator in your code, since the compiler now understands how to use the operator with your class

• This is called “operator overloading”
  – ... we are overloading the use of the operator for more than just the simple types.

You can also do this with functions.
Example of operator overloading

```cpp
class MyInt
{
public:
    MyInt(int x) { myInt = x; }
    MyInt& operator++();

    int GetMyInt() const { return myInt; }

protected:
    int myInt;
};

MyInt &
MyInt::operator++()
{
    myInt++; return *this;
}

int main()
{
    MyInt mi(6);
    ++mi;
    ++mi;
    printf("Value is %d\n", mi.GetMyInt());
}
```

We will learn more about operator overloading later in the quarter.

Define operator ++ for MyInt

Declare operator ++ will be overloaded for MyInt

Call operator ++ on MyInt.
New operators: \texttt{\&\&} and \texttt{\&\&\&}

- "\texttt{\&\&}": Insertion operator
- "\texttt{\&\&\&}": Extraction operator
  - Operator overloading: you can define what it means to insert or extract your object.

- Often used in conjunction with “streams”
  - Recall our earlier experience with C streams
    - stderr, stdout, stdin
  - Streams are communication channels
cout: the C++ way of accessing stdout

New header file (and no ".h"!)

New way of accessing stdout stream.

Insertion operation (<<)
cout is in the “standard” namespace

```cpp
#include <iostream>

using std::cout;

int main()
{
    cout << "The answer is: ";
    cout << 8;
    cout << "\n";
}
```

“using” command puts the “cout” portion of the standard namespace (“std”) in the global namespace.

Don’t need “std::cout” any more…
endl: the C++ endline mechanism

- prints a newline
- flushes the stream
  - C version: fflush(stdout)
  - This is because printf doesn’t always print when you ask it to.
    - It buffers the requests when you make them.
    - This is a problem for debugging!!
endl in action

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "The answer is: ";
    cout << 8;
    cout << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$ 
```
<< and >> have a return value

- ostream & ostream::operator<<(int);
  - (The signature for a function that prints an integer)

- The return value is itself
  - i.e., the cout object returns “cout”

- This allows you to combine many insertions (or extractions) in a single line.
  - This is called “cascading”.
Cascading in action

```cpp
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "The answer is: " << 8 << endl;
}
```

```
fawcett:330 childs$ cat printCPP.C
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```
Putting it all together

```
fawcett:330 childs$ cat print.c
#include <stdio.h>

int main()
{
    printf("The answer is: ");
    printf("%d", 8);
    printf("\n");
}
fawcett:330 childs$ gcc print.c
fawcett:330 childs$ ./a.out
The answer is: 8
```

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

int main()
{
    std::cout << "The answer is: ";
    std::cout << 8;
    std::cout << "\n";
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$ ./a.out
The answer is: 8
```

```
fawcett:330 childs$ cat print.C
#include <stdio.h>

int main()
{
    printf("The answer is: %d\n", 8);
}
fawcett:330 childs$ g++ print.C
fawcett:330 childs$
```

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "The answer is: " << 8 << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```
Three pre-defined streams

- `cout <= => fprintf(stdout, ...`
- `cerr <= => fprintf(stderr, ...`
- `cin <= => fscanf(stdin, ...`
cin in action

```
fawcett:330 childs$ cat cin.C
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

int main()
{
    int X, Y, Z;
    cin >> X >> Y >> Z;
    cout << Z << "", " << Y << "", " << X << endl;
}
fawcett:330 childs$ ./a.out
3 5
4
4, 5, 3
```
cerr

- Works like cout, but prints to stderr
- Always flushes everything immediately!

```
fawcett:330 childs$ cat cerr.C
#include <iostream>

using std::cerr;
using std::cout;
using std::endl;

int main()
{
    int *X = NULL;
    stream << "The value is ";
    stream << *X << endl;
}
fawcett:330 childs$ g++ -Dstream=cerr cerr.C
fawcett:330 childs$ ./a.out
The value is Segmentation fault
fawcett:330 childs$ g++ -Dstream=cout cerr.C
fawcett:330 childs$ ./a.out
Segmentation fault
```

“See the error”
fstream

- ifstream: input stream that does file I/O
- ofstream: output stream that does file I/O

- Not lecturing on this, since it follows from:
  - C file I/O
  - C++ streams

http://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm
Project 3D

• Important: if you skip this project, you will still be able to do future projects (3E, 3F, etc)

• Assignment:
  – Write PNMreaderCPP and PNMwriterCPP ... new version of the file reader and writer that use fstream.
More on Memory...
Memory Segments

• Von Neumann architecture: one memory space, for both instructions and data

• so break memory into “segments”
  – ... creates boundaries to prevent confusion

• 4 segments:
  – Code segment
  – Data segment
  – Stack segment
  – Heap segment
Code Segment

• Contains assembly code instructions
• Also called text segment
• This segment is modify-able ... but that is a bad idea
  – “Self-modifying code”
    • Typically ends in a bad state very quickly
Data Segment

• Contains data not associated with heap or stack
  – global variables
  – statics (to be discussed later)
  – character strings you have compiled in
    ```c
    char *str = "hello world\n"
    ```
Stack: data structure for collection

• A stack contains things
• It has only two methods: push and pop
  – Push puts something onto the stack
  – Pop returns the most recently pushed item (and removes that item from the stack)
• LIFO: last in, first out

Imagine a stack of trays.
You can place on top (push).
Or take one off the top (pop).
Stack

• Stack: memory set aside as scratch space for program execution

• When a function has local variables, it uses this memory.
  – When you exit the function, the memory is lost
Stack

• The stack grows as you enter functions, and shrinks as you exit functions.
  – This can be done on a per variable basis, but the compiler typically does a grouping.
    • Some exceptions (discussed later)
• Don’t have to manage memory: allocated and freed automatically
Heap

• Heap (data structure): tree-based data structure

• Heap (memory): area of computer memory that requires explicit management (malloc, free).

• Memory from the heap is accessible any time, by any function.
  – Contrasts with the stack
Memory Segments

+-----------------+
| text            |
+-----------------+
<table>
<thead>
<tr>
<th>(fixed size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>(fixed size)</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>stack</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>free</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>^</td>
</tr>
</tbody>
</table>

Source: http://www.cs.uwm.edu/classes/cs315/Bacon/
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deal location</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```

Return copies into location specified by calling function
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
This code is very problematic ... why?

```c
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}

int *bar()
{
    int stack_varD[2] = { 2, 3 };  
    return stack_varD;
}

int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```

foo and bar are returning addresses that are on the stack ... they could easily be overwritten (and bar’s stack_varD overwrites foo’s stack_varC in this program)
Nested Scope

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}

Code
Data
Stack
stack_varA
stack_varB
Free
Heap
You can create new scope within a function by adding '{' and '}'.

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deal location</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
</tbody>
</table>

Memory pages associated with stack are almost always immediately available.

Memory pages associated with heap may be located anywhere ... may be caching effects.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>
foo is bad code … never return memory on the stack from a function

bar returned memory from heap

The calling function – i.e., the function that calls bar – must understand this and take responsibility for calling free.

If it doesn’t, then this is a “memory leak”.

```c
int *foo()
{
    int stack_varA[2] = { 0, 1 };
    return stack_varA;
}

int *bar()
{
    int *heap_varB;
    heap_varB = malloc(sizeof(int)*2);
    heap_varB[0] = 2;
    heap_varB[1] = 2;
    return heap_varB;
}

int main()
{
    int *stack_varA;
    int *stack_varB;
    stack_varA = foo(); /* problem */
    stack_varB = bar(); /* still good */
}
Memory leaks

It is OK that we are using the heap ... that’s what it is there for

The problem is that we lost the references to the first 49 allocations on heap

The heap’s memory manager will not be able to re-claim them ... we have effectively limited the memory available to the program.

```c
{
    int i;
    int stack_varA;
    for (i = 0 ; i < 50 ; i++)
        stack_varA = bar();
}
```
Running out of memory (stack)

```c
int endless_fun()
{
    endless_fun();
}

int main()
{
    endless_fun();
}
```

stack overflow: when the stack runs into the heap.
There is no protection for stack overflows.
(Checking for it would require coordination with the heap’s memory manager on every function calls.)
Running out of memory (heap)

```c
int *heaps_o_fun()
{
    int *heap_A = malloc(sizeof(int)*1000000000);
    return heap_A;
}

int main()
{
    int *stack_A;
    stack_A = heaps_o_fun();
}
```

If the heap memory manager doesn’t have room to make an allocation, then malloc returns NULL … a more graceful error scenario.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deal</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>location</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Variable scope</strong></td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Fragmentation

- Memory fragmentation: the memory allocated on the heap is spread out of the memory space, rather than being concentrated in a certain address space.
Memory Fragmentation

int *bar()
{
    int *heap_varA;
    heap_varA = malloc(sizeof(int)*2);
    heap_varA[0] = 2;
    heap_varA[1] = 2;
    return heap_varA;
}

int main()
{
    int i;
    int stack_varA[50];
    for (i = 0 ; i < 50 ; i++)
        stack_varA[i] = bar();
    for (i = 0 ; i < 25 ; i++)
        free(stack_varA[i*2]);
}

Negative aspects of fragmentation?
(1) can’t make big allocations
(2) losing cache coherency
Even if there is lots of memory available, the memory manager can only accept your request if there is a big enough contiguous chunk.
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deal location</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Errors

• Array bounds read
  ```
  int main()
  {
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
  }
  ```

• Array bounds write
  ```
  int main()
  {
    int var = 2;
    int arr[3];
    arr[3]=var;
  }
  ```
Memory Errors

- Free memory read / free memory write

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```
Memory Errors

• Freeing unallocated memory

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.
Memory Errors

• Freeing non-heap memory

```c
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```
Memory Errors

• NULL pointer read / write
  
  ```c
  int main()
  {
      char *str = NULL;
      printf(str);
      str[0] = 'H';
  }
  ```

• NULL is never a valid location to read from or write to, and accessing them results in a "segmentation fault"
  
  …. remember those memory segments?
Memory Errors

- Uninitialized memory read

```c
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```