Lectures 8: How C++ Works Under the Covers
Projects

• Hopefully you are done with 3C
• 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
Function Pointers
Function Pointers

• Idea:
  – You have a pointer to a function
  – This pointer can change based on circumstance
  – When you call the function pointer, it is like calling a known function
Function Pointer Example

```c
#include <stdio.h>

int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    multiplier = doubler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
    multiplier = tripler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
}
```

```
128-223-223-72-wireless:cli hank$ gcc function_ptr.c
128-223-223-72-wireless:cli hank$ ./a.out
Multiplier of 3 = 6
Multiplier of 3 = 9
```
Function Pointers vs Conditionals

What are the pros and cons of each approach?
Function Pointer Example #2

```
#include <stdio.h>
void doubler(int *X) { X[0] *= 2; X[1] *= 2; }
void tripler(int *X) { X[0] *= 3; X[1] *= 3; }
int main()
{
    void (*multiplier)(int *);
    multiplier = doubler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
    multiplier = tripler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
}
```

Don’t be scared of extra ‘*’s ... they just come about because of pointers in the arguments or return values.
Simple-to-Exotic Function Pointer Declarations

void (*foo)(void);
void (*foo)(int **, char ***);
char ** (*foo)(int **, void (*)(int));

These sometimes come up on interviews.
Callbacks

• Callbacks: function that is called when a condition is met
  – Commonly used when interfacing between modules that were developed separately.
  – ... libraries use callbacks and developers who use the libraries “register” callbacks.
Callback example

128-223-223-72-wireless:callback hank$ cat mylog.h
void RegisterErrorHandler(void (*eh)(char *));
double mylogarithm(double x);

128-223-223-72-wireless:callback hank$ cat mylog.c
#include <mylog.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* NULL is an invalid memory location. */
/* Useful for setting to something known, rather than leaving uninitialized */
void (*error_handler)(char *) = NULL;

void RegisterErrorHandler(void (*eh)(char *))
{
    error_handler = eh;
}

void Error(char *msg)
{
    if (error_handler != NULL)
        error_handler(msg);
}

double mylogarithm(double x)
{
    if (x <= 0)
    {
        char msg[1024];
        sprintf(msg, "Logarithm of a negative number: %f !!", x);
        Error(msg);
        return 0;
    }

    return log(x);
}
Callback example

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

FILE *F1 = NULL;
void HanksErrorHandler(char *msg)
{
    if (F1 == NULL)
    {
        F1 = fopen("error", "w");
    }
    fprintf(F1, "Error: %s\n", msg);
}

int main()
{
    RegisterErrorHandler(HanksErrorHandler);

    mylogarithm(3);
    mylogarithm(0);
    mylogarithm(-2);
    mylogarithm(5);
    if (F1 != NULL)
        fclose(F1);
}
```

```
128-223-223-72-wireless:callback hank$ cat program.c
128-223-223-72-wireless:callback hank$ ./program
128-223-223-72-wireless:callback hank$ cat error
Error: Logarithm of a negative number: 0.000000 !
Error: Logarithm of a negative number: -2.000000 !
```
Function Pointers

• We are going to use function pointers to accomplish “sub-typing” in Project 2D.
How C++ does OOP under the covers
“this”: pointer to current object

- From within any struct’s method, you can refer to the current object using “this”
How methods work under the covers (1/4)

class MyIntClass
{
public:
    MyIntClass(int x) { myInt = x; }

friend void FriendIncrementFunction(MyIntClass *);
int GetMyInt() { return myInt; }

protected:
    int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (2/4)

class MyIntClass
{
    public:
        MyIntClass(int x) { myInt = x; }
    friend void FriendIncrementFunction(MyIntClass *);
    int GetMyInt() { return myInt; }

protected:
    int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
class MyIntClass
{
public:
    MyIntClass(int x) { myInt = x; }

friend void FriendIncrementFunction(MyIntClass *);
void IncrementMethod(void);
int GetMyInt() { return myInt; }

protected:
    int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{  
    mic->myInt++;
}

void MyIntClass::IncrementMethod(void)
{
    this->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}

fawcett:330 childs$ g++ this.C
fawcett:330 childs$ ./a.out
My int is 14
fawcett:330 childs$
How methods work under the covers (4/4)

The compiler secretly slips “this” onto the stack whenever you make a method call.

It also automatically changes “myInt” to this->myInt in methods.

```cpp
class MyIntClass {
    public:
        int myInt; // 0x8000

    private:
        int myICnt; // 0x8004

    void IncrementMethod(void) {
        this->myInt++; // 0x8004
    }

    void Main()
    { // 0x8000
        MyIntClass MIC(12); // 0x8000
        FriendIncrementFunction(&MIC); // 0x8000
        MIC.IncrementMethod(); // 0x8000
        cout << "My int is " << MIC.GetMyInt() << endl; // 0x8000
    }
};
```

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>MIC/ myInt</td>
<td>12</td>
</tr>
<tr>
<td>0x8004</td>
<td>mic</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>MIC/myInt</td>
<td>13</td>
</tr>
<tr>
<td>0x8004</td>
<td>this</td>
<td>0x8000</td>
</tr>
</tbody>
</table>
Virtual Function Tables
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

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());
}

ID = 387
Picking the right virtual function

```cpp
class A
{
public:
    virtual const char *GetType() { return "A"; }
};

class B : public A
{
public:
    virtual const char *GetType() { return "B"; }
};

int main()
{
    A a;
    B b;
    cout << "a is " << a.GetType() << endl;
    cout << "b is " << b.GetType() << endl;
}
```

It seems like the compiler should be able to figure this out ... it knows that a is of type A and it knows that b is of type B
Picking the right virtual function

```cpp
class A {
  public:
      virtual const char *GetType() { return "A"; }
};

class B : public A {
  public:
      virtual const char *GetType() { return "B"; }
};

void ClassPrinter(A *ptrToA) {
  cout << "ptr points to a " << ptrToA->GetType() << endl;
}

int main() {
  A a;
  B b;
  ClassPrinter(&a);
  ClassPrinter(&b);
}
```

So how does the compiler know?

How does it get “B” for “b” and “A” for “a”??
Virtual Function Table

• Let C be a class and X be an instance of C.
• Let C have 3 virtual functions & 4 non-virtual functions
• C has a hidden data member called the “virtual function table”
• This table has 3 rows
  – Each row has the correct definition of the virtual function to call for a “C”.
• When you call a virtual function, this table is consulted to locate the correct definition.
Showing the existence of the virtual function pointer with sizeof()

```cpp
class A {
    public:
        virtual
};
class B : public A {
    public:
        virtual
};
class C {
    public:
        const char *getType() { return "C"; }
};
int main() {
    A a;
    B b;
    cout << "Size of A is " << sizeof(A) << endl;
    cout << "Size of a pointer is " << sizeof(int *) << endl;
    cout << "Size of C is " << sizeof(C) << endl;
}
```

what will this print?

```
fawcett:330 child$ $./a.out
Size of A is 8
Size of a pointer is 8
Size of C is 1
```
Virtual Function Table

• Let C be a class and X be an instance of C.
• Let C have 3 virtual functions & 4 non-virtual functions
• Let D be a class that inherits from C and Y be an instance of D.
  – Let D add a new virtual function
• D’s virtual function table has 4 rows
  – Each row has the correct definition of the virtual function to call for a “D”.
More notes on virtual function tables

• There is one instance of a virtual function table for each class
  – Each instance of a class shares the same virtual function table

• Easy to overwrite (i.e., with a memory error)
  – And then all your virtual function calls will be corrupted
  – Don’t do this! ;(
Virtual function table: example

CIS 330: Project #2C
Assigned: April 17th, 2014
Due April 24th, 2014
(which means submitted by 6am on April 25\textsuperscript{th}, 2014)
Worth 6\% of your grade

Please read this entire prompt!

Assignment: You will implement subtypes with C.

1) Make a union called \texttt{ShapeUnion} with the three types (Circle, Rectangle, Triangle).
2) Make a struct called \texttt{FunctionTable} that has pointers to functions.
3) Make an \texttt{enum} called \texttt{ShapeType} that identifies the three types.
4) Make a struct called \texttt{Shape} that has a \texttt{ShapeUnion}, a \texttt{ShapeType}, and a \texttt{FunctionTable}.
5) Modify your 9 functions to deal with Shapes.
6) Integrate with the new driver function. Test that it produces the correct output.
Virtual function table: example

class Shape
{
    virtual double GetArea() = 0;
    virtual void GetBoundingBox(double *) = 0;
};

class Rectangle : public Shape
{
    public:
        Rectangle(double, double, double, double, double);
    virtual double GetArea();
    virtual void GetBoundingBox(double *);
    protected:
        double minX, maxX, minY, maxY;
};

class Triangle : public Shape
{
    public:
        Triangle(double, double, double, double, double);
    virtual double GetArea();
    virtual void GetBoundingBox(double *);
    protected:
        double pt1X, pt2X, pt2Y, minY, maxY;
};
Questions

• What does the virtual function table look like for a Shape?

```c
typedef struct {
    double (*GetArea)(Shape *);
    void    (*GetBoundingBox)(Shape *, double *);
} VirtualFunctionTable;
```

• What does Shape’s virtual function table look like?
  – Trick question: Shape can’t be instantiated, precisely because you can’t make a virtual function table
    • abstract type due to pure virtual functions
Questions

• What is the virtual function table for Rectangle?

```c
    c->ft.GetArea = GetRectangleArea;
    c->ft.GetBoundingBox = GetRectangleBoundingBox;
```

• (this is a code fragment from my 2C solution)
Calling a virtual function

• Let X be an instance of class C.

• Assume you want to call the 4\textsuperscript{th} virtual function

• Let the arguments to the virtual function be an integer Y and a float Z.

• Then call:

\[(X.vptr[3])(\&X, Y, Z);\]

- The 4\textsuperscript{th} virtual function has index 3 (0-indexing)
- The pointer to the virtual function pointer (often called a vptr) is a data member of X
- Secretly pass “this” as first argument to method
Inheritance and Virtual Function Tables

This whole scheme gets much harder with multiple inheritance, and you have to carry around multiple virtual function tables.

```cpp
class A {
    public:
        virtual void Foo2();
};

class C : public B {
    public:
        virtual void Foo1();
        virtual void Foo2();
        virtual void Foo3();
};
```

This is how you can treat a C as a B. Same as B’s Location of Foo1

<table>
<thead>
<tr>
<th>A</th>
<th>Location of Foo1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Location of Foo2</td>
</tr>
<tr>
<td>C</td>
<td>Location of Foo3</td>
</tr>
</tbody>
</table>
Virtual Function Table: Summary

• Virtual functions require machinery to ensure the correct form of a virtual function is called
• This is implemented through a virtual function table
• Every instance of a class that has virtual functions has a pointer to its class’s virtual function table
• The virtual function is called via following pointers
  – Performance issue
Now show Project 2D in C++

• Comment:
  – C/C++ great because of performance
  – Performance partially comes because of a philosophy of not adding “magic” to make programmer’s life easier
  – C has very little pixie dust sprinkled in
    • Exception: ‘\0’ to terminate strings
  – C++ has more
    • Hopefully this will demystify one of those things (virtual functions)
class Shape
{
    public:
    int s;
    virtual double GetArea() = 0;
    virtual void GetBoundingBox(double *) = 0;
};

class Triangle : public Shape
{
    public:
    virtual double GetArea() { cerr << "In GetArea for Triangle" << endl; return 1; }
    virtual void GetBoundingBox(double *) { cerr << "In GetBBox for Triangle" << endl; }
};

class Rectangle : public Shape
{
    public:
    virtual double GetArea() { cerr << "In GetArea for Rectangle" << endl; return 2; }
    virtual void GetBoundingBox(double *) { cerr << "In GetBBox for Rectangle" << endl; }
};

struct VirtualFunctionTable
{
    double (*GetArea)(Shape *);
    void (*GetBoundingBox)(Shape *, double *);
};

int main()
{
    Rectangle r;
    cerr << "Size of rectangle is " << sizeof(r) << endl;

    VirtualFunctionTable *vft = *((VirtualFunctionTable**)&r);
    cerr << "Vptr = " << vft << endl;
    double d = vft->GetArea(&r);
    cerr << "Value = " << d << endl;

    double bbox[4];
    vft->GetBoundingBox(&r, bbox);
}
Exceptions
Exceptions

• C++ mechanism for handling error conditions
• Three new keywords for exceptions
  – try: code that you “try” to execute and hope there is no exception
  – throw: how you invoke an exception
  – catch: catch an exception ... handle the exception and resume normal execution
Exceptions

```
fawcett:330 childs$ cat exceptions.C
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try {
        cout << "About to throw 105" << endl;
        throw 105;
        cout << "Done throwing 105" << endl;
    } catch (int &theInt) {
        cout << "Caught an int: " << theInt << endl;
    }
}

fawcett:330 childs$ g++ exceptions.C
```
Exceptions: catching multiple types

```c++
fawcett:330 childs$ cat exceptions2.C
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 105" << endl;
        throw 105;
        cout << "Done throwing 105" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
}
```

```bash
fawcett:330 childs$ g++ exceptions2.C
fawcett:330 childs$ ./a.out
About to throw 105
Caught an int: 105
```
Exceptions: catching multiple types

```c++
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 10.5" << endl;
        throw 10.5;
        cout << "Done throwing 10.5" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
}
```

Exceptions: catching multiple types

```c++
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 10.5" << endl;
        throw 10.5;
        cout << "Done throwing 10.5" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
    catch (double &theDouble)
    {
        cout << "Caught a double: " << theDouble << endl;
    }
}
```
Exceptions: throwing/catching complex types

class MyExceptionType { }; 
void Foo();

int main() 
{ 
  try 
  { 
    Foo(); 
  } 
  catch (MemoryException &e) 
  { 
    cout << "I give up" << endl; 
  } 
  catch (OverflowException &e) 
  { 
    cout << "I think it is OK" << endl; 
  } 
  catch (DivideByZeroException &e) 
  { 
    cout << "The answer is bogus" << endl; 
  } 
}
Exceptions: cleaning up before you return

```c
void Foo(int *arr);

int *
Foo2(void)
{
    int *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (MyExceptionType &e)
    {
        delete [] arr;
        return NULL;
    }

    return arr;
}
```
Exceptions: re-throwing

```c
void Foo(int *arr);

int *
Foo2(void)
{
    int *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (MyExceptionType &e)
    {
        delete [] arr;
        throw e;
    }

    return arr;
}
```
Exceptions: catch and re-throw anything

```cpp
void Foo(int *arr);

int *
Foo2(void)
{
    int *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (...)
    {
        delete [] arr;
        throw;
    }
    return arr;
}
```
Exceptions: declaring the exception types you can throw

```c
int *
MyIntArrayMemoryAllocator(int num) throw(FloatingPointException)
{
    int *arr = new int[num];
    if (arr == NULL)
        throw DivideByZeroException();

    return arr;
}
```
Exceptions: declaring the exception types you can throw ... not all it is cracked up to be

```cpp
int *
MyIntArrayMemoryAllocator(int num) throw(FloatingPointException)
{
    int *arr = new int[num];
    if (arr == NULL)
        throw MemoryException();

    return arr;
}
```

This will compile ... compiler can only enforce this as a run-time thing.

As a result, this is mostly unused (I had to read up on it)

But: “standard” exceptions have a “throw” in their declaration.
std::exception

- C++ provides a base type called "std::exception"
- It provides a method called "what"

```cpp
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
    virtual const char* what() const throw()
    {
        return "My exception happened";
    }
} myex;

int main () {
    try
    {
        throw myex;
    }
    catch (exception& e)
    {
        cout << e.what() << '\n';
    }
    return 0;
}
```
Exceptions generator by C++ standard library

<table>
<thead>
<tr>
<th>exception</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_alloc</td>
<td>thrown by <code>new</code> on allocation failure</td>
</tr>
<tr>
<td>bad_cast</td>
<td>thrown by <code>dynamic_cast</code> when it fails in a dynamic cast</td>
</tr>
<tr>
<td>bad_exception</td>
<td>thrown by certain dynamic exception specifiers</td>
</tr>
<tr>
<td>bad_typeid</td>
<td>thrown by typeid</td>
</tr>
<tr>
<td>bad_function_call</td>
<td>thrown by empty function objects</td>
</tr>
<tr>
<td>bad_weak_ptr</td>
<td>thrown by <code>shared_ptr</code> when passed a bad <code>weak_ptr</code></td>
</tr>
</tbody>
</table>
Project 3F in a nutshell

• Logging:
  – infrastructure for logging
  – making your data flow code use that infrastructure

• Exceptions:
  – infrastructure for exceptions
  – making your data flow code use that infrastructure

The webpage has a head start at the infrastructure pieces for you.
Warning about 3F

• My driver program only tests a few exception conditions
• Your stress tests later will test a lot more.
  – Be thorough, even if I’m not testing it
3F: warning

- 3F will almost certainly crash your code
  - It uses your modules wrong!
- You will need to figure out why, and add exceptions
  - gdb will be helpful
GDB / valgrind?
Additional Material
## Operator Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++ --</td>
<td>Suffix/postfix increment and decrement</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td>()</td>
<td>Function call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[]</td>
<td>Array subscripting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>Structure and union member access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt; (type)[]</td>
<td>Structure and union member access through pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type){list}</td>
<td>Compound literal(C99)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>++ --</td>
<td>Prefix increment and decrement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td>Unary plus and minus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>! ~</td>
<td>Logical NOT and bitwise NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type)</td>
<td>Type cast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Indirection (dereference)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>Address-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sizeof</td>
<td>Size-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>_Alignof</td>
<td>Alignment requirement(C11)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>Multiplication, division, and remainder</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Addition and subtraction</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;=</td>
<td>For relational operators &lt; and \le respectively</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>For relational operators &gt; and \ge respectively</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>== !=</td>
<td>For relational = and \ne respectively</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td>Bitwise XOR (exclusive or)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Bitwise OR (inclusive or)</td>
</tr>
<tr>
<td>11</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>?:</td>
<td>Ternary conditional([note 2])</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>14</td>
<td>=</td>
<td>Simple assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+= -=</td>
<td>Assignment by sum and difference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*= /= %=</td>
<td>Assignment by product, quotient, and remainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>Assignment by bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;= ^=</td>
<td>=</td>
<td>Assignment by bitwise AND, XOR, OR</td>
</tr>
<tr>
<td>15</td>
<td>,</td>
<td>Comma</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>

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
    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 (fixed size)
- data (fixed size)
- stack growth
- free
- heap growth

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/Deallocation</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);
}
```

- **Code**
- **Data**
- **Stack**
  - stack_varC
  - stack_varD
  - <info for how to get back to main>
  - A (= 3)
  - <Location for RV>
  - stack_varA
- **Free**
- **Heap**
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);
}
```

Return copies into location specified by calling function.

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack_varC</td>
</tr>
<tr>
<td>stack_varD</td>
</tr>
</tbody>
</table>

<info for how to get back to main>

A (= 3)

<Location for RV>

stack_varA

Free

Heap
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)
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}

Nested Scope

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

![Diagram showing stack and heap with variables stack_varA and stack_varB]
Nested Scope

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

You can create new scope within a function by adding '{' and '}'.

---

**Nested Scope**
# 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>
</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>
Variable scope: stack

```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 */
}
```

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”.
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>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 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

```c
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
Fragmentation and Big Allocations

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

```java
int main()
{
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
}
```

• Array bounds write

```java
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];
}
```