CIS 607:

Lecture 5: Moving to C++
Moving Some Content To Later Lectures

• Enum / union
• Memory errors
• I really want you to be able to do 3B after this lecture and keep advancing
Projects

• Project 3B
Reading

• No new reading
YouTube Videos (from last week – all are now posted)

• (1) character strings
  – Completed, but YouTube being very slow to process
  – [https://youtu.be/l7YiqkfMUI8](https://youtu.be/l7YiqkfMUI8)

• (2) images – need this for 3A
  – [https://www.youtube.com/watch?v=98uRZSVsQYY](https://www.youtube.com/watch?v=98uRZSVsQYY)
  – (on class website)
3A / 2B: read the data all at once and not one byte at a time
DRAM vs NV-RAM

• DRAM: Dynamic Random Access Memory
  – stores data
  – each bit in separate capacitor within integrated circuit
  – loses charge over time and must be refreshed
  – → volatile memory

• NV-RAM: Non-Volatile Random Access Memory
  – stores data
  – information unaffected by power cycle
  – examples: Read-Only Memory (ROM), flash, hard drive, floppy drive, ...
Seagate Expansion 5TB Desktop External Hard Drive USB 3.0 (STEB5000100)
by Seagate
$133.99 $169.99  Prime
Get it by Friday, Nov 20
More Buying Choices
$133.99 new (68 offers)
$117.24 used (1 offer)

Crucial Ballistix Sport 16GB Kit (8GBx2) DDR3 1600 MT/s (PC3-12800) UDIMM Memory BLS2KIT8G3D1609DS1S00/ BLS2CP8G3D1609DS1S00
by Crucial
$74.99 $159.99  Prime
Get it by Thursday, Nov 19
More Buying Choices
$69.95 new (73 offers)

Corsair Vengeance 16GB (2x8GB) DDR3 1600 MHz (PC3 12800) Desktop Memory (CMZ16GX3M2A1600C10)
by Corsair
$83.90 $188.79  Prime
Get it by Thursday, Nov 19
More Buying Choices
$72.50 new (101 offers)
$74.99 used (3 offers)

Crucial 16GB Kit (8GBx2) DDR3/DDR3L-1600 MHz (PC3-12800) CL11 204-Pin SODIMM Memory for Mac CT2K8G3S160BM / CT2C8G3S160BM
by Crucial
$72.99 $166.99  Prime
Get it by Thursday, Nov 19
More Buying Choices
$71.29 new (99 offers)
$62.00 used (8 offers)
Relationship to File Systems

• File Systems could be implemented in DRAM.
• However, almost exclusively on NV-RAM
  – Most often hard drives
• Therefore, properties and benefits of file systems are often associated with properties and benefits of NV-RAM.
DRAM vs NV-RAM properties

<table>
<thead>
<tr>
<th>Property</th>
<th>DRAM</th>
<th>NV-RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>~10GB</td>
<td>~10TB</td>
</tr>
<tr>
<td>Cost</td>
<td>$5/GB</td>
<td>$0.03/GB</td>
</tr>
<tr>
<td>Latency</td>
<td>&lt;100 nanoseconds</td>
<td>~10 milliseconds</td>
</tr>
</tbody>
</table>

What does 100000:1 mean?

Distance: a 20” map of Oregon is 1:100,000 scale

Time: 1 second to 27 hours is 1:100,000 scale

Time: 1 minute to 69 days is 1:100,000 scale

Time: 1 hour to 11 years is 1:100,000 scale

Time: 1 day to 273 years is 1:100,000 scale
Strategy For This Lecture

• 1) Talk about a bunch of random stuff we will need to understand “classes”

• 2) Talk about all of the changes to structs (which highly relate to classes in C++)

• 2) Talk about classes
Strategy For This Lecture

• 1) Talk about a bunch of random stuff we will need to understand “classes”
• 2) Talk about all of the changes to structs (which highly relate to classes in C++)
• 2) Talk about classes
Relationship between C and C++

• C++ adds new features to C
  – Increment operator!

• For the most part, C++ is a superset of C
  – A few invalid C++ programs that are valid C programs

• Early C++ “compilers” just converted programs to C
A new compiler: g++

• g++ is the GNU C++ compiler
  – Flags are the same
  – Compiles C programs as well
    • (except those that aren’t valid C++ programs)
\textbf{.c vs .C}

- Unix is case sensitive
  - (So are C and C++)
- Conventions:
  - \texttt{.c}: C file
  - \texttt{.C}: C++ file
  - \texttt{.cxx}: C++ file
  - \texttt{.cpp}: C++ file (this is pretty rare)

Gnu compiler will sometimes assume the language based on the extension ... CLANG won’t.
Variable declaration (1/2)

• You can declare variables anywhere with C++!

```cpp
void line_C(double X1, double X2, double Y1, double Y2) {
    double slope;
    double intercept;

    slope = (Y2-Y1)/(X2-X1);
    intercept = Y1-slope*X1;
}

void line_CPP(double X1, double X2, double Y1, double Y2) {
    double slope = (Y2-Y1)/(X2-X1);
    double intercept = Y1-slope*X1;
}
```
Variable declaration (2/2)

- You can declare variables anywhere with C++!

```c++
int C_fun(void)
{
    int sum += i;
    //
    =

    t.C:18:12: error: use of undeclared identifier 'sum'
    return sum;
    //
}

2 errors generated.
```

Why is this bad?

```c++
int CPP_fun(void)
{
    int sum = 0;
    for (int i = 0 ; i < 10 ; i++)
    {
        sum += i;
    }
    return sum;
}
```
C-style Comments

/* Here is a single line comment */

/*
   Here is a multi-line comment */

/*
 * Here is a multi-line comment
 * that makes it clearer
 * that each line is a
 * comment
 * ... because of the *'s
 */
C++-style comments

// this is a comment

/* this is still a comment */

// this is a
// multi-line C++ comment

When you type "//", the rest of the line is a comment, whether you want it to be or not.
Problem with C...

```c
float doubler(float f) { return 2*f; }

#include <stdio.h>

int doubler(int);

int main()
{
    printf("Doubler of 10 is %d\n", doubler(10));
}

int doubler(int);
```

```bash
C02LN0OOGFD58:330 hank$ cat doubler.c
C02LN0OOGFD58:330 hank$ gcc -c doubler.c
C02LN0OOGFD58:330 hank$ cat doubler_example.c
C02LN0OOGFD58:330 hank$ gcc -o doubler_example doubler.o doubler_example.o
C02LN0OOGFD58:330 hank$ ./doubler_example
Doubler of 10 is 2
```
Problem with C...

C02LN00GFD58:330  hank$ nm doubler.o
0000000000000048  s  EH_frame0
0000000000000000  T  _doubler  ←
0000000000000060  S  _doubler.eh
C02LN00GFD58:330  hank$ nm doubler
    doubler.c    doubler_example doubler_example.o
    doubler.o    doubler_example.c doubler_user.o
C02LN00GFD58:330  hank$ nm doubler_example.o
0000000000000068  s  EH_frame0
0000000000000032  s  L_.str
    U  _doubler  ←
0000000000000000  T  _main
0000000000000080  S  _main.eh
    U  _printf

No checking of type...
Problem is fixed with C++...

```c
C02LN00GFD58:330 hank$ cat doubler.c
float doubler(float f) { return 2*f; }
C02LN00GFD58:330 hank$ g++ -c doubler.c
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated
C02LN00GFD58:330 hank$ cat doubler_example.c
#include <stdio.h>

int doubler(int);

int main()
{
    printf("Doubler of 10 is %d\n", doubler(10));
}
C02LN00GFD58:330 hank$ g++ -c doubler_example.c
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated
C02LN00GFD58:330 hank$ g++ -o doubler_example doubler_example.o doubler.o
Undefined symbols for architecture x86_64:
  "doubler(int)", referenced from:
      _main in doubler_example.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
C02LN00GFD58:330 hank$ 
```
Problem is fixed with C++...

This will affect you with C++. Before you got unresolved symbols when you forgot to define the function. Now you will get it when the arguments don’t match up. Is this good?
Mangling

• Mangling refers to combining information about arguments and “mangling” it with function name
  – Way of ensuring that you don’t mix up functions
  – Return type not mangled, though

• Causes problems with compiler mismatches
  – C++ compilers haven’t standardized
  – Can’t take library from icpc and combine it with g++
C++ will let you overload functions with different types

```c
float doubler(float f) { return 2*f; }
int doubler(int f) { return 2*f; }
```

```
C02LN00GFD58:330 hank$ gcc -c t.c
```

```
t.c:2:5: error: conflicting types for 'doubler'
```

```
t.c:1:7: note: previous definition is here
float doubler(float f) { return 2*f; }
```

```
1 error generated.
C02LN00GFD58:330 hank$ g++ -c t.C
C02LN00GFD58:330 hank$
```
C++ also gives you access to mangling via “namespaces”

```c
#include <stdio.h>

namespace CIS330
{
    int GetNumberOfStudents(void) { return 56; };
}

namespace CIS610
{
    int GetNumberOfStudents(void) { return 9; };
}

int main()
{
    printf("Number of students in 330 is %d, but in 610 was %d\n",
            CIS330::GetNumberOfStudents(),
            CIS610::GetNumberOfStudents());
}
```

Functions or variables within a namespace are accessed with “::”
“::” is called “scope resolution operator”
C++ also gives you access to mangling via “namespaces”

The “using” keyword makes all functions and variables from a namespace available without needing “::”. And you can still access other namespaces.

```c++
namespace CIS610
{
    int GetNumberOfStudents(void) { return 9; }
}

using namespace CIS330;

int main()
{
    printf("Number of students in 330 is %d, but in 610 was %d\n",
            CIS610::GetNumberOfStudents(),
            CIS610::GetNumberOfStudents());
}
```

C02LN00GFD58:330 hank$ g++ cis330.C
C02LN00GFD58:330 hank$ ./a.out
Number of students in 330 is 56, but in 610 was 9
C02LN00GFD58:330 hank$
• A reference is a simplified version of a pointer
• Key differences:
  – You cannot do pointer manipulations
  – A reference is always valid
    • a pointer is not always valid
• Accomplished with & (ampersand)
  – &: address of variable (C-style, still valid)
  – &: reference to a variable (C++-style, also now valid)

You have to figure out how ‘&’ is being used based on context.
References are in variable declarations, “address of” are in statements.
Examples of References

```c
C02LN00GFD58:330 hank$ cat ref.C
#include <stdio.h>

void ref_doubler(int &x) { x = 2*x; }

int main()
{
    int x1 = 2;
    ref_doubler(x1);
    printf("Val is %d\n", x1);
}

C02LN00GFD58:330 hank$ g++ ref.C
C02LN00GFD58:330 hank$ ./a.out
Val is 4
```
References vs Pointers vs Call-By-Value

C02LN00GFD58:330 hank$ cat reference.C
#include <stdio.h>

void ref_doubler(int &x) { x = 2*x; }
void ptr_doubler(int *x) { *x = 2**x; }
void val_doubler(int x) { x = 2*x; }

int main()
{
    int x1 = 2, x2 = 2, x3 = 2;
    ref_doubler(x1);
    ptr_doubler(&x2);
    val_doubler(x3);
    printf("Vals are %d, %d, %d\n", x1, x2, x3);
}

ref_doubler and ptr_doubler are both examples of call-by-reference. val_doubler is an example of call-by-value.
Different Misc C++ Topic: initialization during declaration using parentheses

```c
#include <stdio.h>

int main()
{
    int x(3);
    printf("X is %d\n", x);
}
```

This isn’t that useful for simple types, but it will be useful when we start dealing with objects.
Strategy For This Lecture

• 1) Talk about a bunch of random stuff we will need to understand “classes”
• 2) Talk about all of the changes to structs (which highly relate to classes in C++)
• 2) Talk about classes
3 Big changes to structs in C++

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

• Methods and Functions are both regions of code that are called by name ("routines")

• With functions:
  – the data it operates on (i.e., arguments) are explicitly passed
  – the data it generates (i.e., return value) is explicitly passed
  – stand-alone / no association with an object

• With methods:
  – associated with an object & can work on object’s data
  – still opportunity for explicit arguments and return value
Function vs Method

(left) function is separate from struct
(right) function (method) is part of struct

C02LN00GFD58:330 hank$ cat function.c
typedef struct
{
    int i;
} Integer;

int doubler(int x) { return 2*x; };

int main()
{
    Integer i;
    i.i = 3;
    i.i = doubler(i.i);
}

typedef struct
{
    int i;
    void doubler(void) { i = 2*i; };
} Integer;

int main()
{
    Integer i;
    i.i = 3;
    i.doubler();
}

(left) arguments and return value are explicit
(right) arguments and return value are not necessary, since they are associated with the object
Tally Counter

3 Methods:
- Increment Count
- Get Count
- Reset
Methods & Tally Counter

• Methods and Functions are both regions of code that are called by name ("routines")

• With functions:
  – the data it operates on (i.e., arguments) are explicitly passed
  – the data it generates (i.e., return value) is explicitly passed
  – stand-alone / no association with an object

• With methods:
  – associated with an object & can work on object’s data
  – still opportunity for explicit arguments and return value
C-style implementation of TallyCounter

```c
#include <stdio.h>

typedef struct
{
    int count;
} TallyCounter;

void ResetTallyCounter(TallyCounter *tc) { tc->count = 0; }
int GetCountFromTallyCounter(TallyCounter *tc) { return tc->count; }
void TallyCounterIncrementCount(TallyCounter *tc) { tc->count++; }

int main()
{
    TallyCounter tc;
    tc.count = 0;
    TallyCounterIncrementCount(&tc);
    TallyCounterIncrementCount(&tc);
    TallyCounterIncrementCount(&tc);
    TallyCounterIncrementCount(&tc);
    printf("Count is %d\n", GetCountFromTallyCounter(&tc));
}
```

```
TC hank$ gcc tallycounter_c.c
TC hank$ ./a.out
Count is 4
```
C++-style implementation of TallyCounter

```c
#include <stdio.h>

typedef struct
{
    int    count;

    void   Reset() { count = 0; };
    int    GetCount() { return count; };
    void   IncrementCount() { count++; };
} TallyCounter;

int main()
{
    TallyCounter tc;
    tc.count = 0;
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is %d\n", tc.GetCount());
}
```

```bash
$ cat tallycounter.C
$ g++ tallycounter.C
$ ./a.out
Count is 4
```
typedef struct
{
    int     count;

    void   Initialize() { count = 0; }
    void   Reset() { count = 0; }
    int    GetCount() { return count; }
    void   IncrementCount() { count++; }
} TallyCounter;

int main()
{
    TallyCounter tc;
    tc.Initialize();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is %d\n", tc.GetCount());
}
 Constructors

- Constructor: method for constructing object.
  - Called automatically

- There are several flavors of constructors:
  - Parameterized constructors
  - Default constructors
  - Copy constructors
  - Conversion constructors

I will discuss these flavors in upcoming slides
Method for constructor has same name as struct

Constructor is called automatically when object is instantiated (This is the flavor called “default constructor”)

Note the typedef went away ... not needed with C++.
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) after constructors
Argument can be passed to constructor.
(This is the flavor called "parameterized constructor")
Copy Constructor

- Copy constructor: a constructor that takes an instance as an argument
  - It is a way of making a new instance of an object that is identical to an existing one.

```cpp
struct TallyCounter
{
    int    count;

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

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

TallyCounter::TallyCounter(TallyCounter &c) {
    count = c.count;
}
```
Constructor Types

```c
struct TallyCounter
{
    int count;
    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);
    void Reset();
    int GetCount();
    void IncrementCount();
};
```

- Default constructor
- Parameterized constructor
- Copy constructor
Example of 3 Constructors

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

int main()
{
    TallyCounter tc;        /* Default constructor */
    tc.IncrementCount();

    TallyCounter tc2(10);    /* Parameterized constructor */
    tc2.IncrementCount();
    tc2.IncrementCount();

    TallyCounter tc3(tc);    /* copy constructor */
    tc3.IncrementCount();
    tc3.IncrementCount();
    tc3.IncrementCount();

    printf("Counts are %d, %d, %d\n", tc.GetCount(),
           tc2.GetCount(), tc3.GetCount());
}
```

```bash
C02LN00GFD58:TC hank$ cat main.C
```

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

int main()
{
    TallyCounter tc;        /* Default constructor */
    tc.IncrementCount();

    TallyCounter tc2(10);    /* Parameterized constructor */
    tc2.IncrementCount();
    tc2.IncrementCount();

    TallyCounter tc3(tc);    /* copy constructor */
    tc3.IncrementCount();
    tc3.IncrementCount();
    tc3.IncrementCount();

    printf("Counts are %d, %d, %d\n", tc.GetCount(),
           tc2.GetCount(), tc3.GetCount());
}
```

```bash
C02LN00GFD58:TC hank$ ./main
```

```bash
????????????????
```
Conversion Constructor

```c
struct ImperialDistance
{
    double miles;
};

struct MetricDistance
{
    double kilometers;

    MetricDistance() { kilometers = 0; };
    MetricDistance(ImperialDistance &id)
    {
        kilometers = id.miles*1.609;
    }
};
```
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.
Building Large Projects
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 -l -c driver.c

rectangle.o: prototypes.h rectangle.c
   gcc -l -c rectangle.c
Definition of Rectangle in rectangle.c
Why is this a problem?

prototypes.h

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

rectangle.c

```
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;
}
```

driver.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.

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

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

```
C02LN00GFD58: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
#define compilation

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

This is an example of macro replacement.
#define via gcc command-line option

```c
#define

C02LN00GFD58:330 hank$ cat defines.c
int main()
{
    return RV;
}
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

```c
#include <stdio.h>

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

```
C02LN00GFD58:330 hank$ cat conditional_printf.c
C02LN00GFD58:330 hank$ gcc conditional_printf.c
C02LN00GFD58:330 hank$ ./a.out
I am doing PRINTF!!
```

This is how configure/cmake controls the compilation.
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

C02LN00GFD58: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 back to talking about classes)
“this”: pointer to current object

- From within any struct’s method, you can refer to the current object using “this”
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) \(\rightarrow\) inherits from
    • Everything from A is accessible in B
      – (b.x is valid!!)
Object sizes

```c
#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
```
Inheritance + TallyCounter

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());
}
```

#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
Access controls and 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: derived types get access to base type’s data members and methods

Private inheritance: derived types don’t get access.
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>Access</th>
<th>Accessed by derived types*</th>
<th>Accessed outside object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Protected</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Private</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* = with public inheritance
More on virtual functions upcoming

• “Is A”
• Multiple inheritance
• Virtual function table
• Examples
  – (Shape)
Lecture 5.1: Enum / Union
Lecture 5.2: Moving to C++
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/Deal</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>location</td>
<td></td>
<td></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

```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

- Code
- Data
- Stack
  - stack_varC
  - stack_varD
- Heap
- Free

<info for how to get back to main>
A (= 3)
<Location for RV>
stack_varA
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;
    }
}

// Diagram

// Code

// Data

// Stack
stack_varA
stack_varB

// Free

// Heap
Nested Scope

You can create new scope within a function by adding `{` and `}`.
## 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.
<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>
</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>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

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/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>
<tr>
<td>Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Errors

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

• Array bounds write
  ```c
  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];
}
```