CIS 330:

Lecture 11:
C++ and structs

May 6th, 2015
Hank Childs, University of Oregon
Outline

• Review
• C++ and structs
Outline

• Review
• C++ and structs
3 files: prototypes.h, rectangle.c, driver.c

prototypes.h

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

rectangle.c

```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

```c
#include <prototypes.h>

int main()
{
  struct Rectangle r;
  InitializeRectangle(r, 0, 1, 0, 1.5);
}
```
Makefile for prototypes.h, rectangle.c, driver.c

proj2B: driver.c rectangle.c prototypes.h
    gcc -l -c rectangle.c
    gcc -l -c driver.c
    gcc -o proj2B driver.o rectangle.o

Is this a good Makefile?
What’s the problem with it?
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: 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
{
    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);
}
```

driver.c

```
“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.

```c
#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”.

```c
struct Rectangle;

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

This is it. If the compiler can’t figure out how to make object file with this, then it has to give up.
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
Conditional compilation controlled via compiler flags

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

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

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

This is how configure/cmake controls the compilation.
struct Rectangle
{
    double minX, maxX, minY, maxY;
};

#include <struct.h>
#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
    Initialize();
}

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

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

#ifndef / #define to the rescue
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)
.c vs .C

• Unix is case sensitive
  – (So are C and C++)

• Conventions:
  – .c: C file
  – .C: C++ file
  – .cxx: C++ file
  – .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++!

```c
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++!

```cpp
int C_fun(void)
{
    int sum = 0;
    for (int i = 0; i < 10; i++)
    {
        sum += i;
    }
    return sum;
}
```

Why is this bad?

```
C02LN00GFD58:L10 hank$ g++ t.C
t.C:16:17: error: invalid '+=' at end of declaration; did you mean '='?
    int sum += i;
          ^

2 errors generated.
```

What compiler error would you get?

```cpp
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++ also gives you access to mangling via “namespaces”

```cpp
#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 “::”
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 other namespaces.
Note: following slides are slides we didn’t make it to on Friday’s lecture
References

• 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.
Reference vs address of

- References come up when you declare variables.
  - Otherwise “address of”

- int &x = y; // reference ... with the variable
- int *x = &y; // address of ... the value of the variable
Examples of References

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

```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.
References

• Simplified version of a pointer.

• Key differences:
  – You cannot manipulate it
    • Meaning: you are given a reference to exactly one instance ... you can’t do pointer arithmetic to skip forward in an array to find another object
  – A reference is always valid
    • No equivalent of a NULL pointer ... must be a valid instance
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.
Outline

• Review
• C++ and structs
Learning classes via structs

• structs and classes are closely related in C++
• I will lecture today on changes on how “structs in C++” are different than “structs in C”
• ... at the end of the lecture, I will describe how classes and structs in C++ differ.
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
(left) function is separate from struct
(right) function (method) is part of struct

(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));
}
```

```bash
C02LN00GFD58:TC hank$ gcc tallycounter_c.c
C02LN00GFD58: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
C02LN00GFD58:330 hank$ cat tallycounter.C
C02LN00GFD58:330 hank$ g++ tallycounter.C
C02LN00GFD58:330 hank$ ./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
typedef struct
{
    int    count;
}
TallyCounter;

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

#include <stdio.h>

struct TallyCounter
{
    int    count;
    TallyCounter(void) { count = 0; };
}

int main()
{
    TallyCounter tc;
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is %d\n", tc.GetCount());
}
Argument can be passed to constructor.
(This is the flavor called “parameterized constructor”)
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
Methods can be defined outside the struct definition. They use C++’s namespace concept, which is automatically in place. (e.g., TallyCounter::IncrementCount)
“this”: pointer to current object

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

```
C02LN00GFD58:TC hank$ cat main.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());
}

C02LN00GFD58:TC hank$ ./main

```

Conversion Constructor

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

struct MetricDistance
{
    double kilometers;

    MetricDistance() { kilometers = 0; }
    MetricDistance(ImperialDistance &id)
    {
        kilometers = id.miles*1.609;
    }
};
```
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

```cpp
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.
public / private

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

    private:
        int count;

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

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$ 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

```c
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

```cpp
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--; }
}
```

FancyTallyCounter inherits all of TallyCounter, and adds a new method: DecrementCount
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());
}
```

Virtual functions: example

You get the method furthest down in the inheritance hierarchy.
Virtual functions: example

You can specify the method you want to call by specifying it explicitly
**Access controls and inheritance**

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

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

**Public inheritance:** derived types gets 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>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
More on virtual functions upcoming

• “Is A”
• Multiple inheritance
• Virtual function table
• Examples
  – (Shape)
Outline

• Review
• C++ and structs
• Bonus topics
Backgrounding

• “&”: tell shell to run a job in the background
  – Background means that the shell acts as normal, but the command you invoke is running at the same time.

• “sleep 60” vs “sleep 60 &”

When would backgrounding be useful?
Suspending Jobs

• You can suspend a job that is running
  Press “Ctrl-Z”
• The OS will then stop job from running and not schedule it to run.
• You can then:
  – make the job run in the background.
    • Type “bg”
  – make the job run in the foreground.
    • Type “fg”
      – like you never suspended it at all!!
Unix and Windows difference

• Unix:
  – “
” : goes to next line, and sets cursor to far left

• Windows:
  – “
” : goes to next line (cursor does not go to left)
  – “\m” : sets cursor to far left

• Text files written in Windows often don’t run well on Unix, and vice-versa
  – There are more differences than just newlines

vi: “set ff=unix” solves this
memcpy

#include <string.h>

void *
memcpy(void *restrict dst, const void *restrict src, size_t n);

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dst`. If `dst` and `src` overlap, behavior is undefined. Applications in which `dst` and `src` might overlap should use `memmove(3)` instead.

RETURN VALUES
The `memcpy()` function returns the original value of `dst`.

I mostly use C++, and I still use memcpy all the time