Lecture 12: C++ and structs
3A – post mortem
Why Can’t I Modify the Input in Yellow Diagonal

• Imagine I handed you the Mona Lisa with you and asked you to produce a version with a moustache...

• Would you?:
  – make a reproduction and add the moustache to the reproduction
  – vandalize the original
With Respect to 3B...
How to Make a Reproduction

```c
struct name *
ToLowerGood(struct name *input)
{
    struct name *rv = malloc(sizeof(struct name));
    int nchars = strlen(input->buffer);
    rv->buffer = malloc(sizeof(char)*nchars + 1);
    for (int i = 0 ; i < nchars ; i++)
    {
        char c = input->buffer[i];
        if (c >= 'A' && c <= 'Z')
            c = 'a' + (c-'A');
        rv->buffer[i] = c;
    }
    rv->buffer[nchars] = '\0';

    return rv;
}

/* goal: hank childs */

printf("N2's buffer is %s\n", n2->buffer);
printf("N1's buffer is %s\n", n1.buffer);
```

```c
struct name *
ToLowerBad(struct name *input)
{
    int nchars = strlen(input->buffer);
    for (int i = 0 ; i < nchars ; i++)
    {
        char c = input->buffer[i];
        if (c >= 'A' && c <= 'Z')
            c = 'a' + (c-'A');
        input->buffer[i] = c;
    }

    return input;
}
```
REVIEW
Conditional compilation

```c
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

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

C02LN00GFD58:project hank$ make
gcc -I. -c rectangle.c
In file included from rectangle.c:2:
In file included from ./prototypes.h:2:
./struct.h:2:8: error: redefinition of 'Rectangle'
struct Rectangle
^

./struct.h:2:8: note: previous definition is here
struct Rectangle
^

1 error generated.
make: *** [rectangle.o] Error 1
gcc -E rectangle.c

```c
C02LN0GFD58: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

```c
#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__)
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'
int doubler(int f) { return 2*f; }
  ^

t.c:1:7: note: previous definition is here
float doubler(float f) { return 2*f; }
  ^
```
C++ also gives you access to mangling via “namespaces”

```
C02LN00GFD58:330 hank$ cat cis330.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”
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.
Examples of References

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

```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.
C++ and Structures
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”
  – ... when I am done with that topic (probably 1+ lectures more), 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
Function vs Method

(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
C02LN00GF58:TC hank$ gcc tallycounter_c.c
C02LN00GF58: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());
}
```

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
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++.
struct TallyCounter
{
    int count;

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

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

int main()
{
    TallyCounter tc(10);
    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
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`)
“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());
}
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

```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.

```
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;

    This will compile, since main now has access to the private data member “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.
```
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) → inherits from
  • Everything from A is accessible in B
    – (b.x is valid!!)
Object sizes

#include <stdio.h>

struct A
{
    int x;
};

struct B : A
{
    int y;
};

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

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

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

    private:
    int    count;

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

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

• Virtual function: function defined in the base type, but can be re-defined in derived type.
• When you call a virtual function, you get the version defined by the derived type
#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());
}
ID = 387
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

```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;
}
```

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)
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!!
Web pages

• ssh –l <user name> ix.cs.uoregon.edu
• cd public_html
• put something in index.html
• → it will show up as

http://ix.cs.uoregon.edu/~<username>
Web pages

• You can also exchange files this way
  – scp file.pdf
    <username>@ix.cs.uoregon.edu:~/public_html
  – point people to
    http://ix.cs.uoregon.edu/~<username>/file.pdf

Note that ~/public_html/dir1 shows up as
http://ix.cs.uoregon.edu/~<username>/dir1

(“~/dir1” is not accessible via web)