Important From Last Time

- Embedded C
  - Pros and cons
- Macros and how to avoid them
- Intrinsics
- Interrupt syntax
- Inline assembly

Today

- Advanced C
- What C programs mean
- How to create C programs that mean nothing

- The point: Embedded systems need to work all the time
  - You cannot create systems that really work unless you understand your programming language and your tools
  - This is a major theme for the rest of this class

Is the assembly code right?

```
int my_loop (int base) {
  int index, count = 0;
  for (index = base;
       index < (base+10);
       index++)
    count++;
  return count;
}
```

```
my_loop:
        movl    $10, %eax
        ret
```

Is the assembly code right?

```
int my_compare (void) {
  signed char a = 1;
  unsigned char b = -1;
  return (a > b);
}
```

```
my_compare:
        movl    $1, %eax
        ret
```

Which compiler is right?

```
int another_compare (void)  {
    return -1 < (unsigned short)1;
}
```

gcc 4.3.2 for x86:

```
another_compare:
  movl $1, %eax
  ret
```

gcc 3.2.3 for msp430:

```
another_compare:
  mov #llo(0), r15
  ret
```

```
$ gcc foo.c -o foo
$ ./foo
Segmentation fault (core dumped)
$ gcc -O2 foo.c -o foo
$ ./foo
Hello, world.
```

Is it OK for the optimizer to turn a broken program into a working one?

What about the other way around– is it OK for the optimizer to break a working program?

What mathematical function is equivalent to this C function?

```
unsigned foo (unsigned a, unsigned b) {
  return a+b;
}
```

$$foo(a,b) = (a+b) \bmod 2^{\text{CHAR\_BIT} * \text{sizeof(unsigned)}}$$

How about this function?

```
int foo (int a, int b) {
  return a+b;
}
```

$$foo(a,b) = \begin{cases} a+b & \text{if } \ \text{INT\_MIN} \le (a+b) \le \text{INT\_MAX} \\ & \text{undefined otherwise} \end{cases}$$

- **What mathematical function is equivalent to Internet Explorer 7?**

What does each of these mean?

- `x = y = z;`
- `v++`
- `v + v++`
- `*p + (i=1)`
- `(x=0) + (x=0)`
- `i = i + 1;`
- `i = (i = i + 1);`

Point of all this?

- **Arithmetic, logical, and comparison operators are not equivalent to their mathematical counterparts**
- **Expression evaluation is nontrivial**
- **Other parts of the C language have similar counterintuitive behavior**
- **We need a way to figure out what programs mean**
  - **The C standard is an English language description of this**
  - **It is a free download**

How To Think About C

- **The C standard describes an “abstract machine”**
  - **Think of it as a simple interpreter for C**
- **For everything your program does, the C abstract machine tells us the result**
- **C implementation has to act “as if” it implements the computation described by the abstract machine**
  - **But actually, it may do things very differently**

```
int my_loop (int base) {
  int index, count = 0;
  for (index = base;
       index < (base+10);
       index++)
    count++;
  return count;
}
```

```
my_loop:
        movl    $10, %eax
        ret
```

However…

- If you program breaks certain rules, the C implementation can do anything it wants
  - It’s very hard to create C programs that provably don’t break the rules
- Let’s look in more detail about things you can do in C
  - 4 basic categories

- Some operations are defined to behave in a certain way for all C implementations

`(1+1)`

`a[5]=3;` where a[5] is in-bounds

`*p` where p is “int *p” and p points to an int

`if (z) { … }` where z is initialized

- As a programmer your goal is to execute mostly operations with well-defined behavior

- Some operations have implementation-defined behavior
  - The C implementation chooses how to implement the behavior
  - The choice must be consistent and documented
- Examples
  - Sizes of various integers (long, short, etc.)
  - Integer representation
    - Two’s complement? Ones’ complement? Sign magnitude?
  - Effect of bitwise operations on signed values
  - Floating-point rounding behavior
- Use of implementation-defined constructs in unavoidable in real C programs
  - This can limit portability of code

- Some operations have unspecified behavior
  - Implementation has freedom of choice
  - Can make a different choice each time
- Examples
  - Value of padding bytes in structures
  - Order of evaluation of subexpressions
  - Order of evaluation of function arguments
- Total of 53 kinds of unspecified behavior mentioned in the C standard
- Your program must never rely on something that is unspecified

- Code that may depend on unspecified behavior:

```
foo (x(),y());
```

- Code that definitely has unspecified behavior:

```
printf (“a”) + printf (“b”) + printf (“c”)
```

- Try this code at different optimization levels

- Some operations have undefined behavior
  - Consequences are arbitrary
  - Undefined behavior is always a serious bug
- Examples
  - Null pointer dereference
  - Improper type cast
  - Out of bounds array access
  - Divide by zero
  - Signed integer overflow
  - Shift by negative or past bitwidth
  - Read uninitialized value
  - Access to dead stack variable
  - Double-free, use-after-free

- Total of about 190 kinds undefined behavior in the C standard
  - However, some can be reliably detected at compile time
- In practice, what happens when your problem executes an operation with undefined behavior?
  - Maybe the program does just what you expected
  - Maybe it crashes
  - Maybe nothing obvious – program appears to continue normally but it is corrupted somehow
- The vast majority of security holes in C applications are the result of undefined behavior

Type 1 Functions

- Well-defined behavior for all inputs

```
int32_t safe_div_int32_t (int32_t a,
                          int32_t b)
{
  if ((b == 0) ||
      ((a == INT32_MIN) && (b == -1))) {
    report_integer_math_error();
    return 0;
  } else {
    return a / b;
  }
}
```

Type 3 Functions

- Function always has undefined behavior
  - Never write a function like this!
  - In practice they happen by accident
  - Compiler will often silently eliminate some or all code in a function like this

```
int bad (void)
{
  int x;
  return x;
}
```

Another Type 3 Function

```
void str2 (void)
{
  char *s = "hello";
  printf("%s\n", s);
  s[0] = 'H';
  printf("%s\n", s);
}
```

- Why is it type 3?
- What are the compiler’s obligations?

```
str2:
        subq    $8, %rsp
        movl    $.LC1, %edx
        movl    $.LC0, %esi
        movl    $1, %edi
        xorl    %eax, %eax
        call    __printf_chk
        movl    $.LC1, %edx
        movl    $.LC0, %esi
        movl    $1, %edi
        xorl    %eax, %eax
        addq    $8, %rsp
        jmp     __printf_chk
```

Type 2 Functions

- Has undefined behavior for some inputs

```
int32_t div_int32_t (int32_t a,
                     int32_t b)
{
  return a / b;
}
```

- When is it OK to call this function?
- When is it OK to write this function?

Compiling Type 2 Funcs

```
int stupid (int a) {
  return (a+1) > a;
}
```

- What is this function’s precondition?

Compiling Type 2 Funcs

- Case 1: a != INT_MAX
  - Behavior of + is defined → Computer is obligated to return 1
- Case 2: a == INT_MAX
  - Behavior of + is undefined → Compiler has no particular obligations
- Generated code by “gcc –O2”:

```
stupid:
  movl $1, %eax ret
```

Another Type 2

```
void __devexit agnx_pci_remove
  (struct pci_dev *pdev)
{
  struct ieee80211_hw *dev =
    pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;
  if (!dev) return;
  ... do stuff using dev ...
}
```

Case Analysis

- Case 1: dev == NULL
  - “dev->priv” has undefined behavior → Compiler has no particular obligations
- Case 2: dev != NULL
  - Null pointer check won’t fail → Null pointer check is dead code and may be deleted
- This is real Linux kernel code!
  - Since 2009 the Linux kernel us compiled using a special GCC flag that say never to delete null pointer checks
  - Why not just fix the code?

- Why not require the C implementation to emit a compile-time warning when a program might contain undefined behavior?
- Why not require that the C implementation throw an exception in order to avoid undefined behavior?
- How should you deal with undefined behavior?

Signed/Unsigned in C

- Operators like +, -, <, <= in C have signed and unsigned versions
  - The version that gets chosen depends on the signs of the operands
  - Rule: If at least one operand is unsigned, the operator is unsigned

```
int a,b;
unsigned c,d;
(a < b)
(c < d)
(a < c)
```

Integer Promotion

- Operators like +, -, <, <= in C have different versions for different types
  - float, double
  - int, long, long long
- Rule: Both operands are "promoted" to int before the operator executes

```
char c1, c2;
c1 = c1 + c2;
```

- Tricky: If an int can hold all of the values in the original type, a value is promoted to int; if not, it is promoted to unsigned int
  - So, integer promotions always preserve value

- If one of the operands is larger than an int, the other argument is promoted (if necessary) to that size
- The type of the result of an arithmetic operator is the promoted type of the operands
- The type of the result of a comparison operator is int, regardless of the types of the operands
- Integer promotion is performed before the operator is chosen to be signed vs. unsigned

Side Effects

- A C program interacts with the world using side effects
- Side effects are…
  - Accessing a volatile object
  - Calling a function that is side-effecting
- Side effects do not occur immediately, but may be kept pending
  - Why would this seem like a good idea?

Sequence Points

- A "sequence point" in C is a barrier that side effects cannot pass
- When a sequence point is reached…
  - All previous side effects must have taken effect
  - No subsequent side effects can have taken effect
- Between a pair of sequence points, side effects can occur in any order
  - It's your problem to ensure that your code contains enough sequence points to make it correct

Finding Sequence Points

- Point of calling a function, after all arguments are evaluated
- End of evaluating the first operand to && or ||
- End of evaluating the first operand to ? :
- End of each operand to the comma operator
- Completing the evaluation of a full expression, defined as:
  - Evaluating an initializer
  - Expression in a regular statement terminated by a ;
  - Controlling expressions in do, while, switch, for
  - The other two expressions in a for
  - Expression in a return

More Sequence Points

- C standard tells us that
  - Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.
- Violating this rule leads to undefined behavior
- So don't both read and write any single variable in between a pair of sequence points
- However, ++ and – are OK
  - But just once per variable

Important

- Sequence points are about the abstract machine
- They have nothing to do with the generated code
- E.g.

```
a++;
b++;
```

Can be translated to:

```
incl b;
incl a;
```

- Why?

Is the assembly code right?

```
int my_loop (int base) {
  int index, count = 0;
  for (index = base;
       index < (base+10);
       index++)
    count++;
  return count;
}
```

```
my_loop:
        movl    $10, %eax
        ret
```

Is the assembly code right?

```
int my_compare (void) {
  signed char a = 1;
  unsigned char b = -1;
  return (a > b);
}
```

```
my_compare:
        movl    $1, %eax
        ret
```

Which compiler is right?

```
int another_compare (void)  {
    return -1 < (unsigned short)1;
}
```

gcc 4.3.2 for x86:

```
another_compare:
  movl $1, %eax
  ret
```

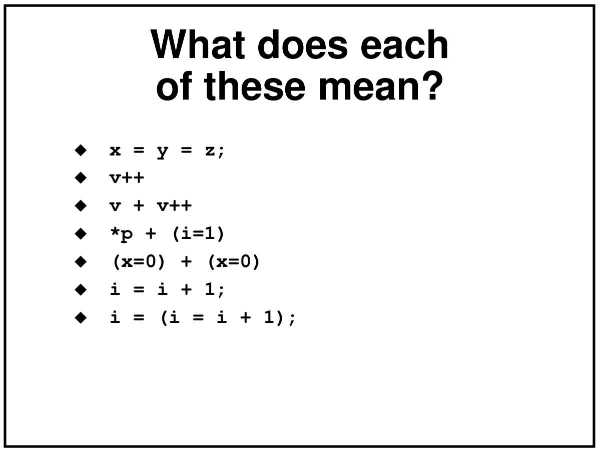
gcc 3.2.3 for msp430:

```
another_compare:
  mov #llo(0), r15
  ret
```

```
$ gcc foo.c -o foo
$ ./foo
Segmentation fault (core dumped)
$ gcc -O2 foo.c -o foo
$ ./foo
Hello, world.
```

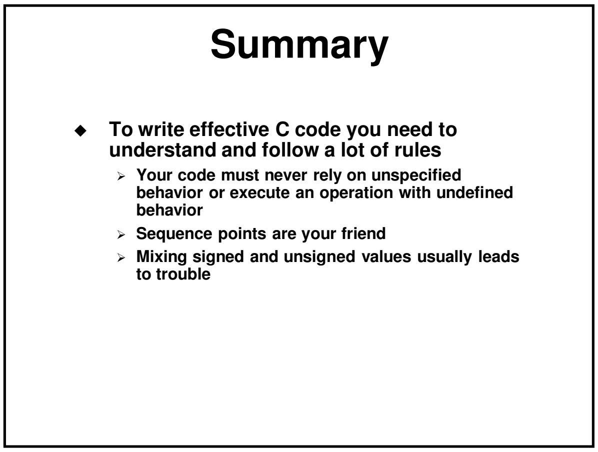
Is it OK for the optimizer to turn a broken program into a working one?

What about the other way around– is it OK for the optimizer to break a working program?



What does each of these mean?

- `x = y = z;`
- `v++`
- `v + v++`
- `*p + (i=1)`
- `(x=0) + (x=0)`
- `i = i + 1;`
- `i = (i = i + 1);`



Summary

- **To write effective C code you need to understand and follow a lot of rules**
  - Your code must never rely on unspecified behavior or execute an operation with undefined behavior
  - Sequence points are your friend
  - Mixing signed and unsigned values usually leads to trouble