C

Kenneth A. Reek, Professor
Department of Computer Science
102 Lomb Memorial Drive
Rochester Institute of Technology
Rochester, NY 14623
kar@cs.rit.edu
What Do I Talk About?

Everyone knows some C by virtue of having learned C++

- Skip boring parts of the language
- Useful old things
- Surprising old things
- Warts
- Useful new things
Development of C

C is an old language

- Dates from the 1970's
- ANSI C standardized in 1989
- New ANSI C standard: C9X
Characteristics

Warts

- We know more about designing languages now than we did back then
- But most of the warts can’t be changed without breaking lots of existing programs

Low-level language

- Access to powerful tools
- Requires skill and discipline
- Doesn’t do as much for you as modern languages
- Libraries
Useful old things
Ternary operator

• Easy to misuse:

\[ x = i > 5 \ ? \ y : z; \]

\textbf{vs.}

\begin{verbatim}
if( i > 5 )
    x = y;
else
    x = z;
\end{verbatim}

• But very useful in some situations:

\[ x[ 3 * f( 2 - a ) + 5 ] = i > 5 \ ? \ y : z; \]
An Analogy

A real-world tool:

- Powerful, dangerous, extremely useful
- Requires training and skill to use safely
An Analogy

- Powerful, dangerous, extremely useful
- Requires training and skill to use safely
An Analogy

• Powerful, dangerous, extremely useful
• Requires training and skill to use safely

Pointers are often maligned as a source of error
• Well, they are!
• But they are still extremely useful
**Insertion into Ordered Singly Linked List**

Four cases:

1. Insert into empty list
2. Insert before 1st node in the list
3. Insert after last node in the list
4. Insert into middle of the list

Most languages allow you to write code that handles only two of these without resorting to special cases, and without using bogus `header` or `trailer` nodes.
Typical Approach

Special cases:

- Insertion into empty list
- Insertion before 1st element of list
In the current node, we need the value and the link.
But in the previous node, we only need the link.
• Point to it directly
• Eliminates the special cases altogether
Inserting into the List

Insertion takes place before the CURR node, and modifies the link that PREV points to.
Inserting into the List

Insertion takes place before the CURR node, and modifies the link that PREV points to.
Inserting into the List

Insertion takes place before the CURR node, and modifies the link that PREV points to.
Inserting into the List

Insertion takes place before the CURR node, and modifies the link that PREV points to.
The C Function

```c
void sll_insert( Node **p_link, Node *new_node ){
    Node *curr;
    while( (curr = *p_link) != NULL && curr->value < new_node->value ){
        p_link = &curr->link;
    }
    new_node->link = curr;
    *p_link = new_node;
}
```

Depends upon the ability to take the address of an existing object
Efficiency

C is very efficient
• Because it doesn’t do much for you
You can decide the best way to implement your program
• But then you must do the work yourself
**Example: Length of a String**

In C++:

```cpp
string s( ... );

for( int i = 0; i < s.length(); i += 1 ){
    // ...
}
```

Design choice: maintain length member on all string operations

- Efficient here
- Not efficient if length is rarely needed

The class designer made this choice
Example: Length of a String

In C:

```c
char s[] = ...;

for( i = 0; i < strlen(s); i += 1 ){
    /* ... */
}
```
Example: Length of a String

In C:

    char s[] = ...;

    for( i = 0; i < strlen(s); i += 1){
        /* ... */
    }

Flon’s Law:
- There is not now, and never will be, a language in which it is the least bit difficult to write bad programs.
Example: Length of a String

Let’s try it again:

```c
char s[] = ...;

len = strlen(s);
for( i = 0; i < len; i += 1 ){
    /* ... */
}
```

Design choice: length computed only when needed

- Alternative: maintain the length yourself
- Drawback: must remember to do it each time the string changes

You get to make the choice
**Hidden Overhead in C++**

Constructors
- Wasted effort if the first use of the object is to assign to it
- Large arrays of objects

Copy constructors
- Function arguments (if you’re not careful with const ref)
- CS 4 exercise

The real problem:
- programmers with no understanding of the consequences of their choices
Hidden Overhead

Increases in chip speed give us the luxury of spending some overhead to take care of mundane details automatically

- It is easier to increase the hidden overhead in a program by an order of magnitude than it is to increase the chip speed by an order of magnitude
- You still need to know the consequences of what you’re doing
Warts
Operators with Side Effects

++ and --

• Use the operand, modify it as a side effect

Gives rise to:

\[ x = i++ + ++i; \]

Yields different results with different compilers

• Some compilers give different results depending on whether or not you optimize
Operators with Side Effects

++ and --

- Use the operand, modify it as a side effect

Gives rise to:

\[ x = i++ + ++i; \]

Yields different results with different compilers

- Some compilers give different results depending on whether or not you optimize

In fact, the expression is illegal: the object \( i \) is modified more than once between sequence points

- Not many people know what \``sequence points\'' are
- Few (no?) compilers issue warnings for this type of error
Unsigned Pitfalls

unsigned a, b;
/* ... */
ans1 = a < b;
ans2 = a - b < 0; /* subtract b from both sides */

The expressions are not equivalent:

• ans2 is always false
Unsigned Pitfalls

unsigned a, b;
/* ... */
ans1 = a < b;
ans2 = a − b < 0; /* subtract b from both sides */

The expressions are not equivalent:
• ans2 is always false

Similarly:
ans3 = strlen( x ) >= strlen( y );
ans4 = strlen( x ) − strlen( y ) >= 0;

• You’re vulnerable unless you realize that strlen() returns a size_t (unsigned)

Signed values in mixed expressions are promoted to unsigned!
**No Boolean Type: Use Ints Instead**

```c
int a, b;

a = 5;
b = 12;
if ( a ) ...; /* is true */
if ( b ) ...; /* is true */
if ( a == b ) ...; /* is not true */
```

a is true, b is true, but a and b are not equal

- Takes discipline to use boolean-style tests only on integer values that are truly boolean, rather than as a shortcut on arbitrary integers
No Boolean Type: Use Ints Instead

if( a < b < c ) ...

This compares the value 0 or 1 to c

- This is entirely legal
- Error cannot be detected because result of the first < is an int
Surprises
Arrays, Subscripts & Pointers

1) An array name is a pointer to the first element in the array
2) The subscript expression \( a[ b ] \) is just a shortcut for the pointer expression \( *( a + ( b ) ) \)

These two facts lead to some interesting results

```c
char arr[100];
char *p = arr + 10;
assert( &p[25] == &arr[35] );
assert( p[25] == arr[35] );
```

- Yes, you can use a subscript on a pointer variable
- This means that \( p[ -10 ] \) through \( p[ -1 ] \) are all legal
- Part of why C does not do subscript checking
Arrays, Subscripts & Pointers

What are the types of ans1 and ans2?

ans1 = "432" + 1;
ans2 = "hello"[ 2 ];
Arrays, Subscripts & Pointers

What are the types of ans1 and ans2?

```c
ans1 = "432" + 1;
ans2 = "hello"[ 2 ];
```

What does the mystery function do?

```c
void mystery( int val ){
    assert( 0 <= val && val <= 100 );
    val += 5;
    val /= 10;
    printf( "%s\n", "**********" + 10 - val ); /* 10 stars in the literal */
}
```
Arrays, Subscripts & Pointers

What are the types of a and b?

\[ a = -35[ b ]; \]
The Twelve Days of Christmas

#include <stdio.h>
main(t, a)
char *a;
{return!0<t?t<3?main(−79,−13,a+main(−87,1−_,
main(−86, 0, a+1 )+a)):1,t<_?main(±79,±13,a+main(±87,1±_, a ):3,main ( −94, −27+t, a 
)&&t == 2 ?_<_13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<−72?main(_,
t,"@n’+,#/+{*w+w#cdnr/+,}{r/*de}+,*+/w[/%+,w#q#n+,#\l+,/n{n+\ 
,k###q#n+,#k+K;+/,’d*3,}\{w+K w’K:+}\e#’;dq’l q’+d’K!/
+k#;q’r}eKK#\w’r}eKK\{nl]/#/q#n’\{w}’\{nl]/+q#n’;d\rw’ i;# )\{n\ 
l]/n{n’; r#{w’r nc[nl]/#l,+’K {rw’ iK{{nl]’/w#q#\ 
’wk nw’ iK{KK[nl]/w{%l’##w’ i; :{nl]/{q’l’d,r’}{nlwb!*/de}’c \ 
{nl]−{rw’+/},#’##*#nc,’,#nw’]/+kd’+e}+;
#’rdq#w! nr’’) }+}{rl#{n’’)# }’}+}##(!/"
:t<−50?_==*a ?putchar(a[31]):main(−65,−a+1):main((’a == ’)/+t,_,a
+1 )}0<t?main ( 2, 2 , "%s”):*a==’/’|main(0,main(−61,*a, "!ek;dc \ i@bK(q)−[w]*%n+r3#l,}\{\nuwloca−O;m .vpbks,fxntdCeghiry”),a+1);}
New in C9X
C++ Style Constructs

- // comments allowed
- Declarations and statements can be intermixed
New Types

- Complex arithmetic
- New typedef types:
  - `int8_t`, `int16_t`, etc.
  - `uint8_t`, `uint16_t`, etc.
  - `int_least8_t`, `int_least16_t`, etc.
  - `uint_least8_t`, `uint_least16_t`, etc.
  - `int_fast8_t`, `int_fast16_t`, etc.
  - `uint_fast8_t`, `uint_fast16_t`, etc.
Booleans (kind of)

New type: bool
• true & false defined as you would expect

But, to avoid breaking old programs:
• It is really called _Bool
• You must include a header file to get bool, true, false
• if and while expressions still take integer expressions (damn)
Declare Identifiers in for Statement

for( int i = 0; i < 10; i += 1 ) ...

But, that gives rise to this ambiguity:

int j = 99;

for( int i = 0, j = 0; i < 10; i += 1, j += 1 ){
    ...
}

printf( "%d\n", j );

prints 99
Variable Length Arrays

The size in an array declaration is no longer restricted to a constant expression

Yeah, big deal. So now we can say:

```c
void func( int size ){
    int arr[ size ];
}
```

instead of having to say:

```c
void func( int size ){
    int *arr = malloc( size * sizeof( int ) );
}
```
Variable Length Arrays

The size in an array declaration is no longer restricted to a constant expression.

Yeah, big deal. So now we can say:

```c
void func( int size ){
    int arr[ size ];
}
```

instead of having to say:

```c
void func( int size ){
    int *arr = malloc( size * sizeof( int ) );
}
```

Wait, it’s better than that:

```c
int ( *p )[ m ] = ...;
```

```c
p + 1 ==> p + ( m * sizeof( int ) )
```

- To do this, runtime size information must be associated with the pointer.
Variable Length Arrays

Can now do this:

```c
void matmult( int n, int m, int p, float a[n][m], float b[m][p], float r[n][p] ){
    for( int i = 0; i < n; i += 1 ){
        for( int j = 0; j < p; j += 1 ){
            r[ i ][ j ] = 0;
            for( int k = 0; k < m; k += 1 ){
                r[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ];
            }
        }
    }
}
```

Formerly, you had to do the subscript calculations yourself
Finally, the results of division with negative operands is no longer implementation defined.

<table>
<thead>
<tr>
<th>dividend</th>
<th>divisor</th>
<th>quot</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>-3</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>-3</td>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>

- div, ldiv, lldiv functions now useful primarily to obtain both the quotient and remainder in one operation
Questions?
Questions?

No questions? Good!
Pass the cookies