



&*%&@#*!

```
int f (void) {  
    int s = 1;  
    int t = 1;  
    int *ps = &s;  
    int **pps = &ps;  
    int *pt = &t;  
  
    **pps = 2;    s == 1, t == 1  
    pt = ps;      s == 2, t == 1  
    *pt = 3;      s == 3, t == 1  
    t = s;        s == 3, t == 3  
}
```

Rvalues and Lvalues

What does = really mean?

```
int f (void) {  
    int s = 1;  
    int t = 1;  
    t = s;  
    t = 2;  
}
```

left side of = is an "lvalue"
it evaluates to a location (address)!

right side of = is an "rvalue"
it evaluates to a value

There is an implicit *
when a variable is
used as an rvalue!

Parameter Passing in C

- Actual parameters are rvalues

```
void swap (int a, int b) {  
    int tmp = b; b = a; a = tmp;  
}
```

```
int main (void) {  
    int i = 3;  
    int j = 4;  
    swap (i, j);  
    ...  
}
```

The value of i (3) is passed, not its location!

swap does nothing

Parameter Passing in C

```
void swap (int *a, int *b) {  
    int tmp = *b; *b = *a; *a = tmp;  
}
```

```
int main (void) {  
    int i = 3;  
    int j = 4;  
    swap (&i, &j);  
    ...  
}
```

The value of &i is passed, which is the address of i

Is it possible to define swap in Python?

Beware!

```
int *value (void)
{
    int i = 3;
    return &i;
}
```

```
void callme (void)
{
    int x = 35;
}
```

```
int main (void) {
    int *ip;
    ip = value ();
    printf ("*ip == %d\n", *ip);
    callme ();
    printf ("*ip == %d\n", *ip);
}
```

*ip == 3

*ip == 35

But it could really be anything!

Manipulating Addresses

```
char s[6];  
s[0] = 'h';  
s[1] = 'e';  
s[2] = 'l';  
s[3] = 'l';  
s[4] = 'o';  
s[5] = '\0';  
printf ("s: %s\n", s);
```

s: hello

`expr1[expr2]` in C is just
syntactic sugar for
`*(expr1 + expr2)`

Obfuscating C

```
char s[6];  
*s = 'h';  
*(s + 1) = 'e';  
2[s] = 'l';  
3[s] = 'l';  
*(s + 4) = 'o';  
5[s] = '\0';  
printf ("s: %s\n", s);
```

s: hello

Fun with Pointer Arithmetic

```
int match (char *s, char *t) {  
    int count = 0;  
    while (*s == *t) { count++; s++; t++; }  
    return count;  
}
```

```
int main (void)  
{  
    char s1[6] = "hello"; The \0 is invisible!  
    char s2[6] = "hohoh";  
  
    printf ("match: %d\n", match (s1, s2));  
    printf ("match: %d\n", match (s2, s2 + 2));  
    printf ("match: %d\n", match (&s2[1], &s2[3]));  
}
```

$\&s2[1]$
 $\rightarrow \&(* (s2 + 1))$
 $\rightarrow s2 + 1$

match: 1
match: 3
match: 2

Condensing match

```
int match (char *s, char *t) {  
    int count = 0;  
    while (*s == *t) { count++; s++; t++; }  
    return count;  
}
```

```
int match (char *s, char *t) {  
    char *os = s;  
    while (*s++ == *t++);  
    return s - os - 1;  
}
```

$s++$ evaluates to s_{pre} , but changes the value of s
Hence, C++ has the same value as C, but has unpleasant side effects.

Quiz

- What does `s = s++;` do?

It is undefined!

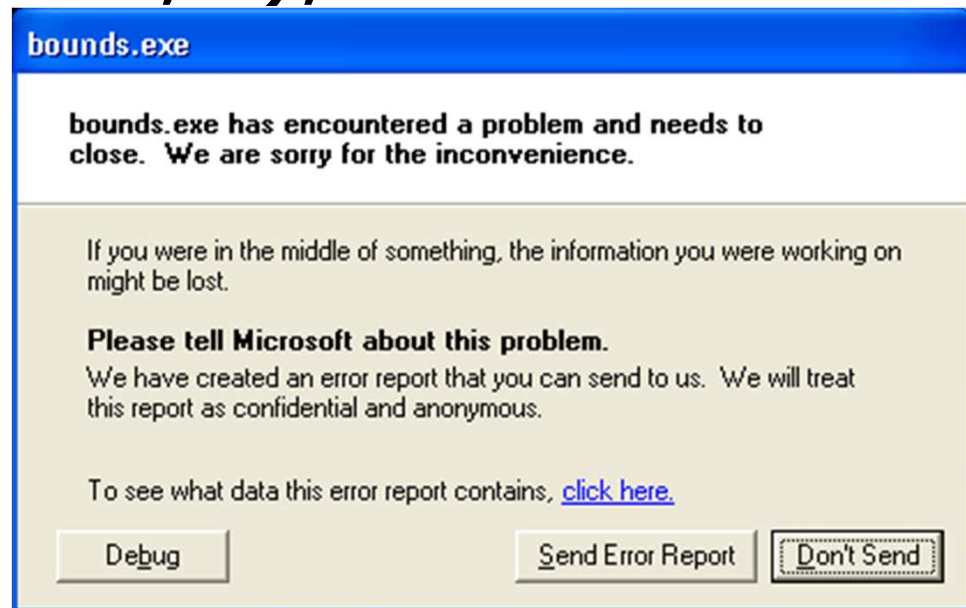
If your C programming contains it, a correct interpretation of your program could make `s = spre + 1`, `s = 37`, or blow up the computer.

Type Checking in C

- Java: only allow programs the compiler can prove are type safe
Exception: **run-time** type errors for downcasts and array element stores.
- **C: trust the programmer.** If she really wants to compare apples and oranges, let her.
- **Python: don't trust the programmer or compiler – check everything at runtime.**

Type Checking

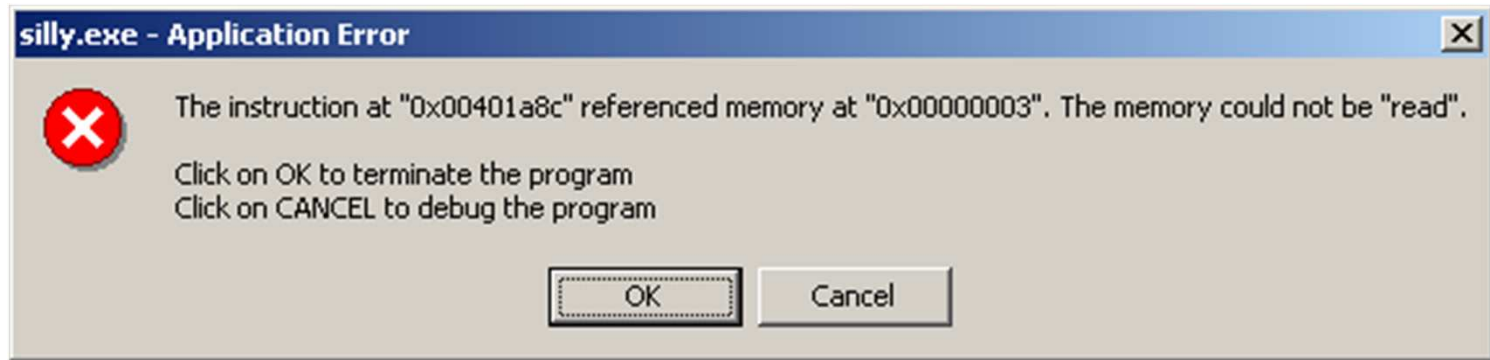
```
int main (void) {  
    char *s = (char *) 3;  
    printf ("s: %s", s);  
}
```



Windows XP (SP 2)

Type Checking

```
int main (void) {  
    char *s = (char *) 3;  
    printf ("s: %s", s);  
}
```



Windows 2000

(earlier versions of Windows would just crash the whole machine)

Exercise

- Study Section 6.3 from the textbook

Skip

- Study section 6.5 from the textbook
- We studied section 6.6 Strings under one dimensional char arrays in ch 5

6.7 Dynamic Memory Allocation

- Dynamically allocated memory is determined at *runtime*
- A program may create as many or as few variables as required, offering greater flexibility
- Dynamic allocation is often used to support data structures such as stacks, queues, linked lists and binary trees.
- Dynamic memory is finite
- Dynamically allocated memory may be freed during execution

Dynamic Memory Allocation

- Memory is allocated using the:
 - malloc function (memory allocation)
 - calloc function (cleared memory allocation)
- Memory is released using the:
 - free function
- The size of memory requested by malloc or calloc can be changed using the:
 - realloc function

malloc and calloc

- Both functions return a pointer to the newly allocated memory
- If memory can not be allocated, the value returned will be a **NULL** value
- The pointer returned by these functions is declared to be a **void pointer**
- A cast operator should be used with the returned pointer value to coerce it to the proper pointer type

Example of malloc and calloc

```
int n = 6, m = 4;  
double *x;  
int *p;
```



X

```
/* Allocate memory for 6 doubles. */  
x = (double *)malloc(n*sizeof(double));
```



p

```
/* Allocate memory for 4 integers. */  
p = (int *)calloc(m,sizeof(int));
```