CSE306 Software Quality in Practice

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
These slides are a reconstruction of the lecture on Wednesday 2/6.

They capture the main points of class, not every last detail.

If you have questions, please ask away in Piazza.
Memory organization

```c
int main() {
    int X = 0;
    ... inv. rec.
    return 0;
}
```
Memory organization

Each process (a running program) has a chunk of memory at its disposal.

This memory is divided into "static" memory (allocated/structured before execution begins) and "dynamic" memory (allocated while the program executes.)
The static segment is divided into a TEXT segment (holding the machine language instructions of the program), and a DATA segment (which has space for statically allocated memory, constants, literal values, etc).
The dynamic segment is divided into STACK and a HEAP areas.

The HEAP is generally located adjacent to the STATIC segment, and grows "down" (to higher memory addresses).
Memory organization

The STACK is generally located at the far end of memory and grows "up" (to lower memory addresses).

The area between the HEAP and the STACK represents available (free) memory.

If the HEAP and STACK collide we have an out-of-memory error.
The STACK holds invocation records (also called stack frames).

An invocation record is created whenever a function is called. It has space for the function's parameters, local variables, any return value, as well as bookkeeping information related to the call itself (e.g. where to return to).
Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

The invocation record for main is pushed on the stack as soon as execution begins.

main's record is the current/active one.
Memory organization

Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

When `f()` is called, an invocation record for `f` is pushed to the top of the stack.

`f`'s record is the current/active one.
Consider this code:

```c
void g() { … }
void f() { … g(); … }
int main() { … f() ... }
```

When `g()` is called, an invocation record for `g` is pushed to the top of the stack.

`g`’s record is the current/active one.
Consider this code:

```c
void g() { … }
void f() { … g(); … }
int main() { … f() … }
```

When `g()` returns its invocation record is removed from the stack, an `f`'s invocation record becomes the current/active one.
Memory organization

Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

When `f()` returns its invocation record is removed from the stack, an `main`'s invocation record becomes the current/active one.
The HEAP is used for dynamic allocation of non-local data.

In Java allocation is done using 'new', as in

```java
px = new Foo();
```

Java's garbage collector frees heap-allocated memory when it is no longer in use.
Memory organization

In C allocation is done using 'malloc' (memory allocate):

```
px = malloc(sizeof(*px));
```

C is not garbage collected. 'free' must be called explicitly to release unused memory and make it available for re-allocation:

```
free(px);
```
In either case the (local) variable px holds the address of the chunk of memory, allocated on the heap, which holds some data.
Memory organization

int main() {
    int x = 0;
    return 0;
}

A local variable, like x in the code shown, has memory for its value set aside in the function's invocation record.

The name of the variable, x in this case, does not exist at runtime.
Memory organization

Any read from x or write to x is translated into a memory access at some offset from the current Stack Pointer (SP). SP points to a known point within an invocation record.

```c
int main() {
    int x = 0;
    .
    .
    .
    return 0;
}
```
#include <stdio.h>

int main() {
    int x = 0;
    while (x < 10) {
        printf("x has value %d\n", x);
        x = x + 1;
    }
    return 0;
}
program #1

#include <stdio.h>

int main() {
    int x = 0;
    while (x < 10) {
        printf("x has value %d\n", x);
        x = x + 1;
    }
    return 0;
}
```c
#include <stdio.h>

int main() {
    int x = 0;
    while (x < 10) {
        printf("x has value \%d\n", x);
        x = x + 1;
    }
    return 0;
}
```

We also discussed where in memory space for the value of variable x exists.

The class correctly said on the stack, in the invocation record of main.

In reality the value of x may exist just in a register at runtime, but for our purposes right now we don't need to be concerned about that.
program #1

#include <stdio.h>

int main() {
    int x = 0;
    while (x < 10) {
        printf("x has value %d\n",x);
        x = x + 1;
    }
    return 0;
}

Students were asked to discuss in groups how to transform this program so that the value being printed was stored on the heap rather than on the stack.

The next several slides show the results.
int main() {
    int *x;
    x = malloc ...  
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x++;
    }
}

#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x;
    x = malloc(sizeof(*x));
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x++;
    }
}

Add required libraries:
- stdio.h for printf
- stdlib.h for malloc and sizeof

Complete call to malloc and sizeof to allocate block of memory of correct size on heap.
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x;
    x = malloc(sizeof(*x));
    while (*x < 10) {
        printf("x has value %d\n",*x);
        *x++;
    }
}
```

```
bash-3.2$ clang -std=c11 -Wall -o g1 g1.c
g1.c:9:2: warning: expression result unused [-Wunused-value]
   *x++;
   ^~~~
1 warning generated.
bash-3.2$
```

Compilation produces a warning.

It's just a warning right? Surely it's safe to ignore warnings...
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x;
    x = malloc(sizeof(*x));
    while (*x < 10)
    { printf("x has value %d\n",*x);
        *x++;
    }
}
```

```
bash-3.2$ clang -std=c11 -Wall -o g1 g1.c
g1.c:9:2: warning: expression result unused [-Wunused-value]
    *x++;
    ^~~~
1 warning generated.
bash-3.2$ ./g1
x has value 0
x has value -1879048192
x has value 0
x has value -1879048192
x has value 0
x has value -1879048192
x has value 0
x has value -1879048192
x has value -1649213425
bash-3.2$
```

Running the code doesn't quite produce the output expected!
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x;
    x = malloc(sizeof(*x));
    while (*x < 10)
    {
        printf("x has value %d\n",*x);
        *x++;
    }
}
```

We wanted `*x++` to be interpreted as `(*x)++` to increment the value on the heap. Instead this is interpreted as `*(x++)` which means that the pointer is incremented (the address stored in `x` is advanced `sizeof(*x)` bytes) and then dereferenced. The dereferencing has no effect, hence the warning.
int *x = malloc(sizeof(int));
while (*x < 10) {
    printf("x has value %d\n",*x);
    *x=*x+1;
}
free(x);
return 0;
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x = malloc(sizeof(int));
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x = malloc(sizeof(int));
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}

bash-3.2$ clang -std=c11 -Wall -o g2 g2.c
bash-3.2$

Code compiles without warnings.
```
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x = malloc(sizeof(int));
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```

```bash
class-3.2$ clang -std=c11 -Wall -o g2 g2.c
class-3.2$ ./g2
x has value 0
x has value 1
x has value 2
x has value 3
x has value 4
x has value 5
x has value 6
x has value 7
x has value 8
x has value 9
```

Code produces expected result.

It must be correct, right?
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *y = malloc(sizeof(int));
    *y = -2;
    free(y);
    int *x = malloc(sizeof(int));
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```

What if we allocate, initialize, and free some memory before we allocate space for our int?

It will compile with no errors, no warnings. Just like before.

But look at the values printed!

What would happen if *y = 10?
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x = malloc(sizeof(int));
    *x = 0;
    while (*x < 10) {
        printf("x has value %d\n",*x);
        *x=*x+1;
    }
    free(x);
    return 0;
}
```

We should initialize all memory before we use it!
int main()
{
    int *x = malloc(sizeof(int));
    *x = 0;
    while (x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *x = malloc(sizeof(int));
    *x = 0;
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *x = malloc(sizeof(int));
    *x = 0;
    while (x < 10) {
        printf("x has value \%d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```

bash-3.2$ clang -std=c11 -Wall -o g3 g3.c
```
g3.c:8:12: warning: ordered comparison between pointer and integer ('int *' and 'int')
    while (x < 10) {
        ~^~~
1 warning generated.
bash-3.2$
```

Compilation produces a warning.

It's just a warning right? Surely it's safe to ignore warnings...the last time was just bad luck.
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *x = malloc(sizeof(int));
    *x = 0;
    while (x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```

```
bash-3.2$ clang -std=c11 -Wall -o g3 g3.c
1 warning generated.
```

```
bash-3.2$ ./g3
Oh bother!
We should have compared *x to 10, not x. x is a pointer, and so holds an address, whose value is >= 10, so the loop body never executes.
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *x = malloc(sizeof(int));
    *x = 0;
    while (x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}

bash-3.2$ clang -std=c11 -Wall -o g3 g3.c
bash-3.2$ ./g3

Oh bother!
We should have compared *x to 10, not x. x is a pointer, and so holds an address, whose value is >= 10, so the loop body never executes.
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *x = malloc(sizeof(int));
    *x = 0;
    while (*x < 10) {
        printf("x has value %d\n", *x);
        *x = *x + 1;
    }
    free(x);
    return 0;
}
```

Changing this to *x < 10 fixed the problem.

```
bash-3.2$ clang -std=c11 -Wall -o g3 g3.c
bash-3.2$ ./g3
x has value 0
x has value 1
x has value 2
x has value 3
x has value 4
x has value 5
x has value 6
x has value 7
x has value 8
x has value 9
bash-3.2$
```
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int * px;
    px = malloc(sizeof(*px));
    if ( px == NULL ) {
        exit(EXIT_FAILURE);
    }
    *px = 0;
    while (*px < 10) {
        printf("*px has value %d\n",*px);
        *px = *px + 1;
    }
    free(px);
    return(EXIT_SUCCESS);
}
```

Here's my code.

I often (but not always) name my pointer variables starting with a 'p', to remind me of the indirection.

It's a good idea to check whether memory allocation was successful or not.

In printf statement the output says "*px has value..." to accurately reflect what's happening in the program.

Although it would not be a problem to not free the memory allocated using malloc in *this* program (since it exits right after the free call) you should get into the habit of always freeing malloced memory when it is no longer needed.

This code also shows the use of the library-defined constants EXIT_FAILURE and EXIT_SUCCESS. They are preferable to 1 and 0. Also, exit can be used to terminate a program even if not in main. Because this is main return(EXIT_SUCCESS) and exit(EXIT_SUCCESS) will have an equivalent effect.
Compiler versions

`timberlake.cse.buffalo.edu`

- `/usr/bin/gcc` 4.4.7
- `/util/bin/gcc` 6.4.0
- `/util/gcc-7.2.0/bin/gcc` 7.2.0
- `/util/llvm/bin/clang` 3.5.0
Compiler versions

We want to use the more up-to-date versions.
Compiler versions

timberlake.cse.buffalo.edu

- /util/gcc-7.2.0/bin/gcc 7.2.0
- /util/llvm/bin/clang 3.5.0

Make sure you ssh to timberlake from the lab machines:

ssh timberlake.cse.buffalo.edu
Compiler versions

timberlake.cse.buffalo.edu

- /util/gcc-7.2.0/bin/gcc 7.2.0
- /util/llvm/bin/clang 3.5.0

Make sure your path is updated so that you can use these versions. Update in your ~/.cshrc file.
Compiler documentation

https://gcc.gnu.org/onlinedocs/7.2.0/

https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Standards.html#C-Language

http://releases.llvm.org/3.5.0/tools/clang/docs/UsersManual.html

http://releases.llvm.org/3.5.0/tools/clang/docs/UsersManual.html#c
Common options

-std set language standard
-o set output file name
-g debugging
-c compile/assemble do not link
-Wall report "all" warnings
-L library path
-I include path