CSC209: Software Tools and Systems Programming

Week 2: C Programming, Arrays, and Pointers

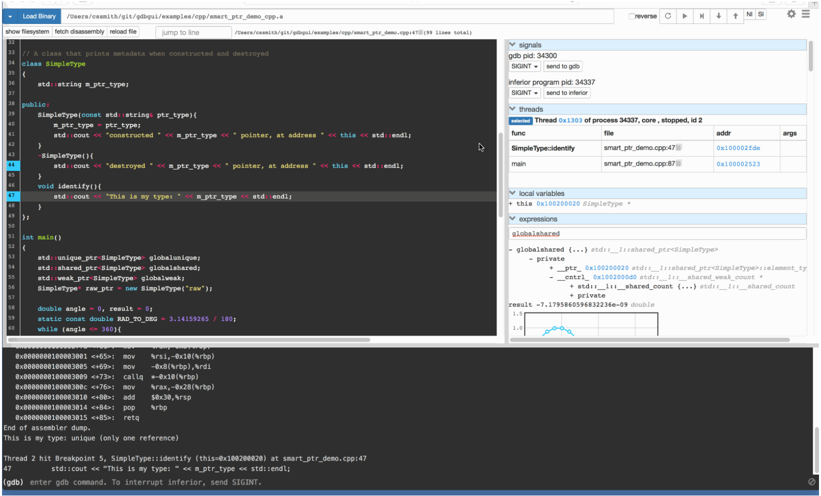
Slides based on material from Andi Bergen

Important Tasks Coming Up

Task	Date
First graded lab	This week (due Friday)
PCRS Week 3: Dynamic memory	Monday
Milestone 1	Next Friday

Visualizing and Debugging

- pythontutor.com/c.html
- Investing time to learn gdb will pay off handsomely
- [gdbgui (gdbgui.com)](https://www.gdbgui.com/: very powerful for generating visualizations



/Users/cssmith/git/gdbgui/examples/cpp/smart_ptr_demo_cpp.a

/Users/cssmith/git/gdbgui/examples/cpp/smart_ptr_demo.cpp:47 (99 lines total)

```cpp

// A class that prints metadata when constructed and destroyed
class SimpleType
{
    std::string m_ptr_type;

public:
    SimpleType(const std::string& ptr_type){
        m_ptr_type = ptr_type;
        std::cout << "constructed " << m_ptr_type << " pointer, at address " << this << std::endl;
    }
    ~SimpleType(){
        std::cout << "destroyed " << m_ptr_type << " pointer, at address " << this << std::endl;
    }
    void identify(){
        std::cout << "This is my type: " << m_ptr_type << std::endl;
    }
};

int main()
{
    std::unique_ptr<SimpleType> globalunique;
    std::shared_ptr<SimpleType> globalshared;
    std::weak_ptr<SimpleType> globalweak;
    SimpleType* raw_ptr = new SimpleType("raw");

    double angle = 0, result = 0;
    static const double RAD_TO_DEG = 3.14159265 / 180;
    while (angle <= 360){
```

gdb pid: 34300

inferior program pid: 34337

Thread 0x1303 of process 34337, core , stopped, id 2

func	file	addr	args
SimpleType::identify	smart_ptr_demo.cpp:47	0x100002fde	
main	smart_ptr_demo.cpp:87	0x100002523	

```
+ this 0x100200020 SimpleType *
```

globalshared

```
- globalshared {...} std::__1::shared_ptr<SimpleType>
    - private
        + __ptr_ 0x100200020 std::__1::shared_ptr<SimpleType>::element_ty
        - __cntrl_ 0x1002000d0 std::__1::__shared_weak_count *
            + std::__1::__shared_count {...} std::__1::__shared_count
            + private
result -7.17958605968322236e-09 double
```

```
   0x0000000100003001 <+65>:  mov    %rsi,-0x10(%rbp)
   0x0000000100003005 <+69>:  mov    -0x8(%rbp),%rdi
   0x0000000100003009 <+73>:  callq  *-0x10(%rbp)
   0x000000010000300c <+76>:  mov    %rax,-0x28(%rbp)
   0x0000000100003010 <+80>:  add    $0x30,%rsp
   0x0000000100003014 <+84>:  pop    %rbp
   0x0000000100003015 <+85>:  retq
End of assembler dump.
This is my type: unique (only one reference)

Thread 2 hit Breakpoint 5, SimpleType::identify (this=0x100200020) at smart_ptr_demo.cpp:47
47          std::cout << "This is my type: " << m_ptr_type << std::endl;
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

Programming in C: Return Values

```
while (scanf(...) != EOF) { ... }
```

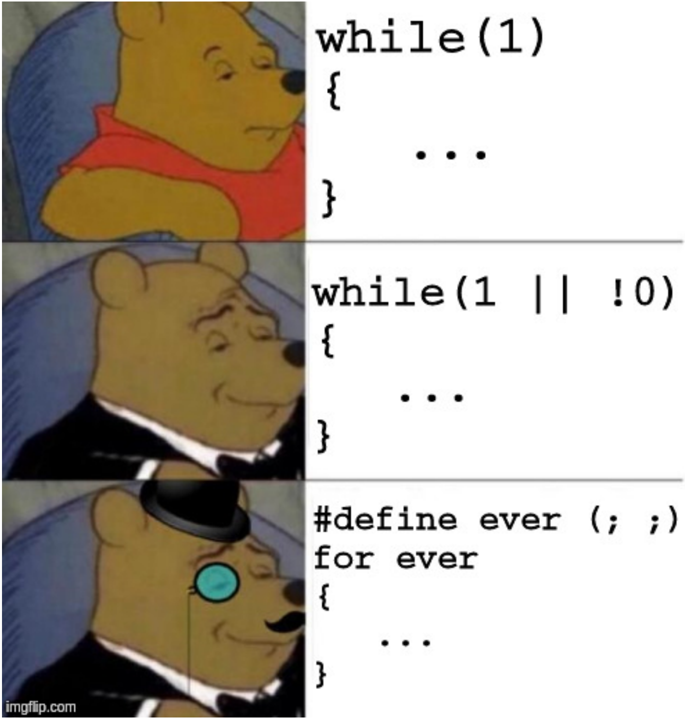
- Almost every library call has a return value
- Always check return values
  - C does not throw exceptions like Java or Python
  - *Be paranoid* about whether or not each library call completes successfully

What does the above code do? Check `man 3 scanf` and scroll to `RETURN VALUE`

Programming in C: Macros

- Return values are often defined as *macros*, e.g., `EOF`
  - These typically “expand” to integer constants
  - Typically defined in `.h` files
  - Already saw an example of this in PCRS:

```
#define DAYS 365
```



```
while(1)
 {
     ...
 }
```

```
while(1 || !0)
 {
     ...
 }
```

```
#define ever (; ;)
for ever
{
    ...
}
```

Compiler Warnings (and Errors) are Your Friends

Common `gcc` compiler flags (all explained in `man gdb`):

- `-g`: Include debugging symbols in compiled program (`gdb` and `valgrind` make use of these)
- `-Wall`: Warn about highly-questionable code
- `-Wextra`: More warnings (sometimes helpful)
- `-Wpedantic`: All possible warnings
- `-Werror`: Treat all warnings as errors

C: Memory (un)Safety

- C assumes that you know what you're doing
  - A perilous assumption: 70% of security vulnerabilities in Microsoft products are due to **avoidable mistakes that C/C++ allow you to make**
- Example of unsafe code that will compile and run:

```c
int arr[10];
arr[-1] = 123;
```

- Use gcc flag `-fsanitize=address` to catch memory safety bugs

Project Requirement

Your code must compile with:

```
-g -Wall -Wextra -Werror
-fsanitize=address,leak,object-size,
bounds-strict,undefined
-fsanitize-address-use-after-scope
```

These flags make all warnings into errors and check several common memory errors

C: Undefined Behaviour

- *Undefined behaviour* is any operation for which the C standard imposes no requirements
- Example: The contents of uninitialized variables are **undefined**
  - The following code will likely print **garbage values**, but **it will compile and run** without memory checks:

```c
int a;
printf("%d", a);
```

PCRS: Arrays

Declaring arrays:

```
int student_ids[400];
```

Writing to or reading from array elements:

```
student_ids[0] = 1001111111;
student_ids[399] = 1002222222;
int x = student_ids[0];
```

Questions about these?

Arrays Worksheet

PCRS: Pointers and the & Operator

> *A pointer is a variable that contains the memory address of another variable*

1. Assume `x` is an integer and `px` is a pointer
2. Then, `px = &x` stores the *address* of `x` in `px`

The `&` operator expects its operand to be a variable or array element, so constructs such as `&(x+1)` are illegal

PCRS: Pass-by-value and Pass-by-reference

```
int x = 10;
increment_int(x); // Cannot change x
increment2_int(&x); // Can change x
```

`increment()` takes an *integer* parameter, whereas `increment2()` takes an *address of an integer*

PCRS: Pointers and the ∗ Operator

> *The ∗ operator interprets its operand as an address, and fetches the memory contents at that address*

1. Assume that `y` is an integer and `px` is a pointer
2. The statement `y = *px` assigns to `y` the integer that `px` points to

The ∗ operator is said to *dereference* its operand

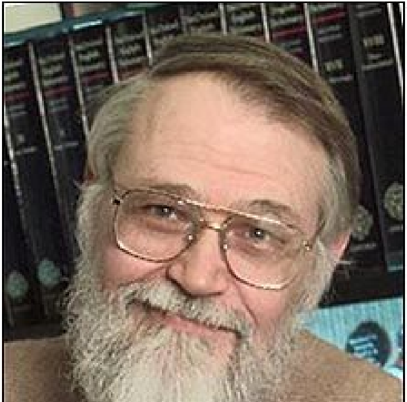
PCRS: Declaring Pointers

Pointer declarations are intended as a mnemonic, so:

```
int *px;
```

means that `*px` is equivalent to a variable of type `int`. Thus, `px` is a pointer containing the address of an `int`.

Why? Because K&R.(See p. 90)



Notes on Pointers and Addresses

- A pointer is not an array
  - But it can contain the address of an array
- An array is not a pointer
  - But the compiler interprets the name of an array as the address of its zeroth element, so the following statements are equivalent

```
int *x = &a[0];
int *y = a; // Assuming "a" is an array.
```

A Common Error With Pointers

What does this do:

```
int *x;
*x = 10;
```

What about this:

```
int *x;
printf("%d", *x);
```

Never dereference uninitialized (or `NULL`) pointers!

Pointers Worksheet

Extra Slides

Common Size of C Primitives

Type	`sizeof` (bytes)	bits
`char`	1	8
`int`	4	32
`long int`	8	64
`long long int`	8	64

GNU C compiler (`gcc`) default values (`std=gnu11`) on a 64-bit system. See GNU C Reference Manual.

Note: Compiler and machine dependent.

Pointer Size

- On modern systems, pointers are 64 bits (8 bytes)
  - e.g., `0xFFFFFFFFFFFFFFFF`
- In memory diagrams, pay attention to whether each “cell” represents a single byte or multiple bytes

Hexadecimal, Decimal, Octal, and Binary

- A hexadecimal digit corresponds to 4 binary digits
  - `0x` prefix indicates hex, e.g., `0xFF`
  - `b` prefix indicates binary, e.g., `0b11`
- You may also encounter octal notation
  - `0` prefix, e.g., `012`
  - `\` prefix followed by up to 3 digits, e.g., `\111`
- Try declaring `int x` and assigning values in hex, decimal, octal, and binary
- Tutorial on binary, decimal, and hexadecimal notation