VARIADIC FUNCTIONS

Plan for Today

- Complex Declarations [Review]
- Variadic Functions
- Mixing C and C++ Code
- Uniform Initialization

Next week

- RAII, RVO, NRVO
- Rule of Three, Move semantics, Rule of Five, Rule of Zero

Precedence Rule: Example

- `char* const *(*next)();`
- `char *(*c[10])(int **p);`

Unscrambling Declarations by Diagram (1/2)

- Declarations in C are read boustrophedonically, i.e. alternating right-to-left with left-to-right
- Start at first identifier you find when reading from the left
- When a token in declaration is matched against diagram, erase it from further consideration
- At each point, look first to token to right, then to the left
- When everything has been erased, the job is done

Unscrambling Declarations by Diagram (2/2)

Step Number	Token to match	How to read
1. Go to leftmost identifier	Identifier	say *"identifier is"*
2. Look at next token to **right** if it is a square bracket.	[possible-size]	for each pair, say **"array of"**
3. or if it is an opening parenthesis	[possible-parameters]	Read up to closing parent-heses say **"function returning"**
4. If token to **left** is a opening parenthesis.	(stuff-already dealt with)	this is opening parentheses, grouping together part of the declaration we have already dealt with. read up to the balancing parenthesis. start again at step 2
5. If token to **left** is any of const, volatile or *	const volatile *	keep reading tokens to left until it is not one of these three: for const say "read-only" for volatile say "volatile" for * say "pointer to" start again at step 4
6. Tokens that remain form the basic type of declaration		read off tokens that remain, e.g. static unsigned int

C/C++ Functions

- So far, functions declared in our C/C++ programs have specified parameter lists
  - Fixed number of parameters
  - Every parameter has pre-specified type
- This is good
  - Compiler is able to use function declarations to detect deviations in numbers and type of arguments
- What about `printf` and `scanf`?

C-Style Variadic Functions

- `printf` and `scanf` are known as *variadic* functions
- How to declare such variadic functions?
- How can such functions access extra arguments passed when they don't know how many arguments there are or what types these arguments have?

```
printf("Hello world\n");
printf("Hello %s\n", fnam);
printf("Your name is %s %s\n", fnam, lnam);
printf("%d + %d = %d\n", n1, n2, n1+n2);

scanf("%s", fnam);
scanf("%s%s", fnam, lnam);
scanf("%s%s %d/%d/%d", fnam, lnam, &m, &d, &y);
```

Declaring C-Style Variadic Functions

- Provide *one* or more *defined* parameter declarations in parameter list followed by `...`

defined parameter declaration required!!!
That is parameter with name and type associated with it

```
int printf(char const *format_string, ...);
```

Ellipsis means "*and maybe some more parameters*"

Declaration of `printf` says it has a defined parameter of type `char const*` followed by variable number of additional parameters

How Do `printf` and `scanf` Know?

function to print to *standard output*

② function arguments

①

```
printf("Distance is %f\n", dist);
```

③ format string

④ print list

⑤

1) *Format specifier* or *conversion specifier* controls how output is printed to standard output
2) Literal characters are printed as is
3) Character following % is abbreviation for type of data it represents and ***must match*** with corresponding argument
4) `%f` means print floating-point value using fixed-point notation

What Can Go Wrong?

- Variadic functions *are* flexible
- However, compiler is unable to check:
  - Type safety of arguments being passed to function

```
printf("%s %s %d\n", age, lastname, firstname);
```

  - If number of arguments being passed matches semantics of function definition

```
printf("%s %s %d\n", firstname, lastname);
```

  - In both cases, we've undefined behavior

How Do `printf` and `scanf` Work?

- How does `printf` get access to unnamed, variable count of additional parameters following defined parameter?
- `<cstdarg>` provides a *type* and *set of macro definitions* that define how to step through variable parameter list

Type `va_list`

- Type `va_list` is name of opaque structure that will maintain information about entire variable parameter list

```
#include <cstdarg>

void foo(int defined_param, ...) {
  // argp will point to each unnamed
  // parameter in turn ...
  va_list argp;
  // other code follows
}
```

Macro `va_start`

- Makes variable `argp` of type `va_list` point to first unnamed parameter

```
void foo(int first_param, double second_param, ...) {
  va_list argp;
  // argp is pointing to first unnamed parameter
  va_start(argp, second_param);

  // other code follows
}
```

Macro va_arg

- Each call of va_arg returns one parameter and steps argp to next unnamed parameter

```
void foo(int first_param, double second_param, ...) {
  va_list argp;
  // argp is pointing to first unnamed parameter
  va_start(argp, second_param);

  // assuming first two unnamed parameters have
  // int type followed by double type
  int ival = va_arg(argp, int);
  double dval = va_arg(argp, double);

  // other code follows
}
```

Macro `va_arg`: Default Promotions

- Compiler performs default promotions on all parameters that match ellipsis
  - `char` and `short` arguments promoted to `int`
  - `float` values promoted to `double`
  - Therefore, doesn't make sense to pass types such as `char`, `short`, and `float`

Macro va_end

- Must be called to reset global variables and perform cleanup

```
void foo(int first_param, double second_param, ...) {
  va_list argp;
  // argp is pointing to first unnamed parameter
  va_start(argp, second_param);

  // assuming first two unnamed parameters have
  // int type followed by double type
  int ival = va_arg(argp, int);
  double dval = va_arg(argp, double);

  // extract other unnamed parameters

  va_end(argp);
}
```

Mixing C and C++ Code: First Steps

- Problem similar to cobbling together only C or only C++ program out of object files produced by more than one compiler
  - Size and alignment of fundamental data types such as `int`s and `double`s
  - Mechanism by which parameters are passed from caller to callee and who orchestrates the passing
- Make sure your C++ and C compilers generate compatible object files
- Then, four other things to worry about: *name mangling, initialization of statics, dynamic memory allocation, data structure compatibility*

Name Mangling (1/4)

- C++ must mangle names because of overloading
- Unnecessary in C
- If you stay within confines of C++, name mangling is not of concern
  - Function `draw_line` is mangled to `_Z9draw_linev` but you use name `draw_line` and you don't care about name mangling
- But if `draw_line` is C function, then object file will contain compiled version of `draw_line` called `draw_line`
- When you try to link with this C object file, you get an error

Name Mangling (2/4)

- You can tell your C++ compiler to suppress name mangling

```
// function implemented in non-C++ language
// and is meant to be imported by C++ linker
extern "C"
void draw_line(int, int, int, int);
```

- You can also tell your C++ compiler to suppress name mangling for certain C++ function names

```
// function implemented in C++ and is to be
// exported to clients using other languages
extern "C"
void draw_line(int, int, int, int);
```

Name Mangling (3/4)

- You can `extern "C"` set of functions like this:

```
extern "C" {
  // disable name mangling for following functions
  void draw_line(int, int, int, int);
  unsigned int twiddle_bits(unsigned int, unsigned int);
  void simulate_rope(int iterations);
}
```

Name Mangling (4/4)

- You want `extern "C"` when compiling for C++ but not for C
- Polyglot header files can be structured like this:

```
#ifdef __cplusplus
// disable name mangling for following functions
extern "C" {
#endif
  void draw_line(int, int, int, int);
  unsigned int twiddle_bits(unsigned int, unsigned int);
  void simulate_rope(int iterations);
#ifdef __cplusplus
}
#endif
```

Initialization of Statics (1/4)

- In C++ lots of code can get executed before and after `main`
  - Initialization of static class members, static class objects, and objects at file scope occurs before main gets executed
  - Objects created thro' static initialization must have their dtors called

Initialization of Statics (2/4)


```cpp
// C++ main looks like this:
int main() {
  // C++ implementation performs
  // static initialization here

  // statements in main go here

  // C++ implementation performs
  // static destruction here
}
```

Initialization of Statics (3/4)

- This means that when mixing C and C++ code, you must write `main` in C++
  - If you can't write `main` in C++, the program is toast
- This is true even if C program is calling C++ functions
  - Possible for C++ library to contain static objects (if not now, maybe in future)
- Rather than rewriting your C code, you could do this:

Initialization of Statics (4/4)


```
// rename C's main to real-main
extern "C"
int real_main(int argc, char *argv[]);

// write a new main in C++
int main(int argc, char *argv[]) {
  return real_main(argc, argv);
}
```

Dynamic Memory

- Simple but consistent rule: C++ parts of program always use `new` and `delete`; C parts use `malloc` (and its variants) and `delete`

Data Structure Compatibility

- No hope of making C functions understand C++ features
- So lowest common denominator is what C can do:
  - Can safely exchange normal pointers to C-style objects and pointers to non-member functions or static functions
  - Structures and variables of built-in types can also freely cross C/C++ border

Mixing C and C++ Code: Summary

- Make sure C++ and C compilers produce compatible object files
- Declare functions to be used by both language `extern "C"`
- Write `main` in C++
- Always use `delete` with memory from `new`; always use `free` with memory from `malloc`
- Limit what you pass between two languages to data structures that compile under C

Uniform Initialization

- Uniform initialization (since C++11) addresses:
  - Confusion of multiple syntaxes
  - Inability to cover all initialization scenarios

Pre-Modern C++ Initialization (1)

- Initialization of variables and objects in C++98 is confusing mess
  - Can happen with parentheses, braces, and/or assignment operator

```
int x(0);  // initializer is in parentheses
string s1;  // call default ctor
string s2("abc");  // call conversion ctor
vector<int> v(10);  // 10 elements each initialized to 0
vector<int> v(5, 6);  // 5 elements each with value 6
int y[] = {1,2,3};  // initializers are in braces
int z = 0;  // initializer follows “=“
```

Pre-Modern C++ Initialization (2)

- C++ initialization was divided into three different categories
  - Initialization of built-in types
  - Initialization of aggregate types
  - Initialization of data members in user-defined types

Pre-Modern C++ Initialization: Built-In Types (1)

- Initialization of non-static built-in types using equals symbol and parentheses

```
int u(12);  // initializer is in parentheses
int v = u;  // initializer follows "="
void* pv = nullptr;  // follows "="
char x('a');  // initializer is in parentheses
```

Pre-Modern C++ Initialization: Built-In Types (2)

- Use of = symbol for initialization misleads C++ novices into thinking assignment taking place
  - Not big deal for built-in types
  - But, important to distinguish for user-defined types because different functions are called

```
string s1;  // calls default ctor
string s2 = s1;  // not an assignment – calls copy ctor
s2 = s1;  // assignment – calls copy  operator=
```

Pre-Modern C++ Initialization: Built-In Types (3)

- Can also initialize non-static built-in types using syntax of explicit constructor

```
int z = int();  // initializer is 0
int z2 = int(11);  // initializer is 11
int y();  // NOT definition of variable y
```

Pre-Modern C++ Initialization: Built-In Types (4)

- This feature allows template code to ensure values of any type have certain default value:

```
template <typename T>
void f() {
 ...
 T x = T();
 ...
}
```

Pre-Modern C++ Initialization: Built-In Types (5)

- Then, we also have grammar ambiguity in C++ called *most vexing parse*

```
int x(); // declaration of function x
int y = int();  // initializer is 0
int z = int(11);  // initializer is 11
```

- Basically means "when a well-formed C++ statement can be interpreted as either declaration or something else (such as expression or definition), *favor declaration*"

Pre-Modern C++ Initialization: Built-In Types (6)

- Most vexing parse can also arise in other scenarios

```
struct Foo {
 Foo() { ... }
 Foo(int i) { ... }
// ...
};

Foo f1(10);  // call Foo ctor with argument 10
Foo f2();    // most vexing parse when
             // calling Foo with default ctor
```

Pre-Modern C++ Initialization: Built-In Types (7)


```
struct Foo {
 Foo() { ... }
// ...
};

struct Boo {
 Boo() { ... }
// ...
};
```

```
struct Coo {
 Foo f;
 Boo b;
 Coo(Foo const& ff, Boo const& bb)
    : f(ff), b(bb) { ... }
};
```

```
Coo x(Foo(), Boo());  // most vexing parse!
                      // NOT definition of object of type x
```

Pre-Modern C++ Initialization: Most Vexing Parse (1)

- What is meaning of following declaration?

```
struct S {
   S();
   // ...
};

S s(S());
```

- Is it "definition of object s of type S which is initialized with temporary object of type S (that is generated by default ctor of S)"

Pre-Modern C++ Initialization: Most Vexing Parse (2)

- Wrong!!!

```
S s(S());
```

- Declaration of function `s` returning object of type `S` taking pointer to function taking nothing and returning object of type `S`
  - Construct `S()` is considered an *abstract declarator* – a declarator without identifier
  - Abstract declaration is of function taking nothing and returning `S`
  - Implicitly converted to function pointer `S (*)()`

Pre-Modern C++ Initialization: Most Vexing Parse (3)

- Rule of thumb – Say what you mean!!!
- If you want to declare a function:

```
S s( S (*)() );
```

- If you want to declare an object `s`, add extra parentheses

```
S s( (S()) );
```

Pre-Modern C++ Initialization: Aggregates (1)

- Initialization of aggregates requires braces

```
int w[] = {1,2,3};  // initializers are in braces
char x[] = {'h','e','l','l','o','\0'};

struct S {  // C-style POD structure
  int a;
  float b;
};
S z = { 10, 10.1f };
```

Pre-Modern C++ Initialization: Aggregates (2)

- Exception is string literals

```
char x[] = {'a','b','\0'};

char y[] = "ab";  // same as previous
```

Pre-Modern C++ Initialization: Aggregates (3)

- Initialization of data members in user-defined types requires parentheses

```
struct S {
 int a;
 float b;
 S(int i, float f) : a(i), b(f) {}
};

S s1(10, 10.1f);  // initializers in parentheses
S s2 = { 20, 20.1f };  // error
```

Pre-Modern C++ Initialization: User-Defined Types (1)

- But for POD user-defined types, initialization of aggregates requires braces

```
struct T {  // C-style POD structure
 int a;
 float b;
};
T t1 = {10, 10.1f}; // initializers in braces
T t2(10, 10.1f); // error
                  // initializers in  parentheses
```

Pre-Modern C++ Initialization: User-Defined Types (12

- Even with several syntaxes, not possible in pre-modern C++ to express certain desired initializations
  - Not possible to initialize STL containers with group of values such as 1, 22, 33

Modern C++ Initialization (1)

- Modern C++ addresses:
  - Confusion of multiple syntaxes
  - Inability to cover all initialization scenarios

Modern C++ Initialization (2)

- Modern C++ introduces concept of *uniform initialization*
  - Means that for any initialization scenario, single initialization syntax of braces can be used
  - Also called *braced initialization*

```
int i {10};  // braced initializer

int j = {11};  // uses = and braces
```

Modern C++ Initialization (3)

- Braced initialization allows formerly inexpressible to be expressed
  - Specifying initial contents of container is simple and easy

```
int v[] {1, 2, 3};
vector<int> w {4, 5, 6, 7};
list<string> x {"a", "b", "c"};
complex<double> y {4.1, 5.2};
```

Modern C++ Initialization (4)

- Braces can also be used to specify default initialization values for non-static data members

```
class Str {
 ...
 size_t n{ 0 };
 char* p{ new char [n+1] };
};
```

Modern C++ Initialization (5)

- Note: Default initialization values for non-static data members can also be specified without using braced initialization

```
class Str {
 ...
 size_t n = 0 ; // fine
 char* p = new char [n+1]; // fine
};
```

Modern C++ Initialization (6)

- Prohibits *narrowing initializations*

```
int i1(1.2);        // ok but i1 has 1
int i2 = 1.2;       // ok but i2 has 1
int i3{1.2};        // error: narrowing
int i4 = {1.2}; // error: narrowing
```

Modern C++ Initialization (7)

- To check whether narrowing applies, compiler might consider current values, if available at compile time
  - If value representable exactly as target type, conversion is not narrowing

```
int i;
...
char c0{i};       // error – narrowing
char c1 = i;      // ok - conversion
char c2{7};       // ok – not narrowing
char c3 = 9999; // ok – but c3 has 15
char c4{9999};   // error – narrowing
```

Modern C++ Initialization (8)


```
double d1, d2, d3;
...
int sum1(d1+d2+d3);   // okay – but
                      // truncated to int
int sum2 = d1+d2+d3;// okay – but
                    // truncated to int
int sum3{d1+d2+d3};  // error – narrowing
```

Modern C++ Initialization (9)


```
vector<int> v1 {1,2,3}; // ok

// error – narrowing doubles to ints
vector<int> v2{1,2.3,3.4};
```

Modern C++ Initialization (10)

- Braced initialization immune to most vexing parse
- Recall that `f2` cannot be default constructed:

```
struct Foo {
 Foo() { ... }
 Foo(int i) { ... }
// ...
};

Foo f1(10);  // call Foo ctor with argument 10
Foo f2();    // most vexing parse when
             // calling Foo with default ctor
```

Modern C++ Initialization (11)

- Since functions can't be declared using braces for parameter list, we can default construct an object using braces:

```
struct Foo {
 Foo() { ... }
 Foo(int i) { ... }
// ...
};

Foo f1(10);  // call Foo ctor with argument 10
Foo f2();    // most vexing parse when
             // calling Foo with default ctor
Foo f2{};    // call default Foo ctor
```

Modern C++ Initialization (12)

- Empty braces force so-called *value initialization*
  - User-defined type initialized by default ctor
  - Non-static variable of built-in type initialized by zero (`nullptr`, for pointer type)

```
int x;        // x  has undefined value
int y{};      // y  is initialized to 0
int* px;      // px  has undefined value
int* py{};    // py initialized by nullptr
string s{}; // call default ctor
```

Plan for Next Week

- RAII, RVO, NRVO
- Rule of Three, Move semantics, Rule of Five, Rule of Zero
- Smart Pointers