Discussion 3

CSE 131

overview

- pointers
- function pointers
- type casts
- address-of
- function overloading (extra credit)

pointer declaration

```
int **x;                // pointer to pointer to int
typedef float* PTR;     // alias PTR is a pointer to float
PTR y;                  // y is a pointer to float
PTR *z;                 // z is a pointer to a pointer to float
```

pointer usage

```
*myPtr = 3;
myStructPtr->myStructField;
new myPtr;
delete myPtr;
myPtr = nullptr;
if (myPtr != nullptr && myPtr != myPtr) { /* stuff */ }
```

dereference

- only accepts arguments of PointerType
- result STO
  - has the pointed to type
  - is addressable

arrow operator

- left side must be a pointer to a struct
- right size must be some field in the struct

new & delete

- only accepts arguments of PointerType
- new is like calloc()
  - but no actual allocation in project 1
- delete is like free()
  - but no actual deallocation in project 1

assignability

- think polymorphism
- check if the base types are equivalent
- int **[5] assignable to int **[5] ?
  - [5] equivalent to [5]? yep
  - * equivalent to *? yep
  - * equivalent to *? yep
  - int equivalent to int? yep

assigning array to pointers

- base type of the array must be *equivalent* to the base type of the pointer
  - TYPE* <= TYPE[5]

```
typedef float[2] FARR;
FARR farr;
float *fp = farr; // ok because base types are equivalent (float and float)
```

nested arrays

```
typedef float[2] FA2;
typedef FA2[3] FA2A3;
FA2A3 fa2a3;

// fails because base types are not equivalent (float** <= float[2][3])
float **fpp = fa2fa3;

// ok (float[2]* <= float[2][3])
FA2 *fa2p = fa2fa3;
```

function pointers

- specific and unique type of pointer
- they do not use * to dereference
  - instead, implicitly dereferenced using parentheses
- similar assignability and comparison rules to normal pointers
  - can compare and assign nullptr to them
  - can only assign functions and function pointers to function pointers, using their identifiers directly

function pointers

```
typedef funcptr : int (int x, int y) TWOINTFUNC;
TWOINTFUNC ptr1, ptr2;

function : int addition(int x, int y) { return x + y; }
function : int subtraction(int x, int y) { return x - y; }

function : int main() {
     if (ptr1 == nullptr) ptr1 = addition;

     ptr1(4, 6);          // 10
     ptr2 = subtraction;
     ptr2(5, 2);          // 3
     ptr2 = ptr1;
     ptr2(5, 2);          // 7
     ptr2 = nullptr;

     return 0;
}
```

type casts

- pretty straightforward
  - take the operand STO and return an appropriate STO (e.g. ExprSTO or ConstSTO) with the type specified in the type cast
- some work for casting constants
  - need to convert the value of the constant appropriately
- result STO is always an r-val

address-of

- operand must be addressable
- take the operand and make a PointerType wrapping the type of the operand
  - should be an ExprSTO which is set as an r-val
- if you dereference the result of an address-of, you will get a modifiable l-val
  - regardless of whether the original was a modifiable l-val or not

address-of

```
int x, y;
int *z;
const int w = 77;

z = &x;         // &x is simply an r-val
&x = nullptr;   // error, not a modifiable l-val
y = *&x;        // *&x basically just x, so ok
*&x = y;        // the * reverses the &x, making it a modifiable l-val
*&w = y;        // the * reverses the &w, making it a modifiable l-val
                // even though w was originally a constant
&*z = z;        // error, result of address-of is not a modifiable l-val
```

address-of

```
function : int foo() { return 0; }
typedef funcptr : int() MYFP;
MYFP myFuncPtr;

myFuncPtr = foo;
myFuncPtr();              // this will be a call to foo

MYFP* myFuncPtrPtr;
myFuncPtrPtr = &foo;      // error, foo is a constant r-val
myFuncPtrPtr = &myFuncPtr // totally fine
(*myFuncPtrPtr)();        // this will be a call to foo
```

overloading (extra credit)

```
function : void foo(float x) ...
function : void foo(int x) ...
function : void foo(int x, float y) ...
function : void foo(float x, int y) ...

function : int main() {
     foo(1);      // maps exactly to second one
     foo(1.7);    // maps exactly to first one
     foo(4, 8.8)  // maps exactly to third one
     foo(5, 6)    // error, no perfect match
     foo(1, 2, 3) // error, no perfect match

     return 0;
}
```

implementation

- starter code puts FuncSTO on to the symbol table using the name as the identifier
- two possible ways to allow overloading
  - name mangling
  - function lookup table

name mangling

- the idea is to incorporate the parameter types in the name
  - foo(float x, int y) becomes foo_float_int in the symbol table
  - when you call foo(3.2, 7) you can look up the FuncSTO by searching for foo_float_int

function lookup table

- create some sort of table that stores all functions of the same name
- when you look up a name
  - get a list of functions with the same name
- to find the matching function for a call
  - look through all functions with the same name and match to the calling arguments

next steps

- finish project 1
- write test programs to verify correctness
- come to lab hours and ask questions
- after you get everything working, consider working on the extra credit