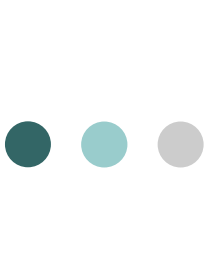

Classes and Types

Classes and Types Topics

- Storage Classes
- Type Specifiers

Storage Classes

- auto
  - *auto* keyword is new with the ANSI standard
  - *auto* is the default for locally declared variables
  - *auto* variables may only be declared within a compound statement
  - storage for an *auto* variable is allocated each time the variable's defining block is entered, and deallocated each time its block is exited

Storage Classes

- auto
  - Example

```
int main( int argc, char **argv )
{
   auto  int       inx      = 0;
   int             test     = 0;
   for ( inx = 0 ; inx < MAX ; ++inx )
   {
         double  test      = 3.14;
         . . .
   }
   return 0;
}
```

Storage Classes

- extern
  - an *extern* declaration may occur anywhere, but is most commonly found in the global declaration area
  - *extern* is the default for globally declared variables
  - *extern* references are resolved by the linker
  - for any *extern* variable there must be exactly one defining declaration

Storage Classes

- extern
  - for any *extern* variable there may be zero or more referencing declarations
  - a defining declaration omits the *extern* keyword, and should give the defined variable an initial value
  - a referencing declaration includes the *extern* keyword, and must not give the declared variable an initial value
  - the result of having two defining instances of a single *extern* variable is unpredictable, and environment dependent.

Storage Classes

- extern
  - Example

```
/*** Source module A ***/
int  control_var = 0;
int func1( . . . )
{
     . . .
}
```

```
/*** Source module B ***/
extern int  control_var;
int  func2( . . . )
{
     . . .
}
/*** Source module C ***/
extern int  control_var;
int   func3( . . . )
{
     . . .
}
```

Storage Classes

- static
  - a *static* declaration may occur anywhere
  - a *static* declaration in the global area overrides the default *extern* storage class of a variable or function
  - storage for a *static* variable is allocated once, when the containing program is executed, and deallocated when the program terminates

Storage Classes

- static
  - Example

```
static IMAGE_DATA_t   image_parms;
void display_image( void )
{
   static BOOL_t          init       = FALSE;
   int                    inx        = 0;
   if ( !init )
   {
          init_image(  &image_parms  );
          init = TRUE;
           . . .
   }
}
```

Storage Classes

- register
  - a *register* declaration is a special kind of *auto* declaration
  - declaring a variable to be *register* is a "hint" to the compiler that the variable should be store in a hardware register
  - the compiler may ignore the register hint, may limit the number of allowed register variables, and may limit the types of variables that can be store in a register (*int* is always allowed)

Storage Classes

- register
  - Example

```
void      test_funk(  void  )
{
   register          int        inx        = 0;
                . . .
   for  (  inx  =  0 ;  inx < 100000 ; ++inx  )
   {
                . . .
   }
}
```

Storage Classes

- typedef
  - a *typedef* declaration indicates that the declared identifier is a type rather than a variable or function
  - does not allocate storage … syntactic convenience
- Example

```
typedef int BOOL_t;
int  main(  int  argc,  char  **argv  )
{
    BOOL_t    status = TRUE;
    int           rcode = 1;
        . . .
    if ( !status )
      rcode = 0;

    return rcode;
}
```

Type Specifiers

- **const**
  - a *const* specifier for a variable or parameter indicates that the value of the variable or parameter will not be changed
  - the compiler will provide minimal protection for *const* variables and parameters, but cannot guarantee that a *const* storage location will not be modified

Type Specifiers

- **const**
  - **Example**
    - const int parm1 = 32;
      declares parm1 *to be a constant* int
    - const int *parm1_p;
      declares parm1_p *to be a pointer to a constant* int
    - int *const parm1_p = &someIntVariable;
      declares parm1_p *to be a constant pointer to an* int
    - const int *const parm1_p = &parm1;
      declares parm1 *to be a constant pointer to a constant* int

Type Specifiers

- **volatile**
  - a *volatile* specifier informs the compiler that an object's value may be altered in unpredictable ways, hence references to the object should not be optimized

- Example

```
extern  volatile  int key_input;
void funk_e(  void  )
{
    int  inx  =  0;
    for  (  inx  = 0                          ;
          inx  < MAX && !key_input            ;
          ++inx
        )
      execute_big_proc( inx );
}
```

Type Specifiers

- “`Volatile`, in particular, is a frill for esoteric applications, and much better expressed by other means. Its chief virtue is that nearly everyone can forget about it. `Const` is simultaneously more useful and more obtrusive; you can't avoid learning about it, because of its presence in the library interface “
  - Dennis Ritchie