Declarations (2)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Declaration specifiers
  - Type The Storage Class of a Function
  - Type Qualifiers
- Declarators
- Initializers
- Function specifiers

The Storage Class of a Function

- Function declarations (and definitions) may include a storage class.
- The only options are `extern` and `static`:
  - `extern` specifies that the function has external linkage, allowing it to be called from other files.
  - `static` indicates internal linkage, limiting use of the function's name to the file in which it's defined.
- If no storage class is specified, the function is assumed to have external linkage.

```
extern int f(int i); // external linkage
int h(int i); //external linkage
static int g(int i); //internal linkage
```

The Storage Class of a Function

- Using `extern` is unnecessary, but `static` has benefits:
- ***Easier maintenance.***
  - A `static` function isn't visible outside the file in which its definition appears, so future modifications to the function won't affect other files.
- ***Reduced "name space pollution."***
  - Names of `static` functions don't conflict with names used in other files.
  - The name of static functions can be reused in other files.
  - Especially important in large programs!

Storage Classes - Summary

- A program fragment that shows some possible ways to use storage classes in declarations of variables and parameters.

```
static int c;

void f(int d, int e){
      int h;
      static int i;
      register int k;
}
```

Storage Classes - Summary

- Of the four storage classes, the most important are `static` and `extern`.
- `auto` has no effect, and modern compilers have made `register` less important.

Type Qualifiers

declaration-specifiers declarators ;
`int i;`

- Declaration specifiers fall into three categories:
  - Storage classes
  - **Type qualifiers**
  - Type specifiers
- There are three ***type qualifiers:*** `const` and `volatile, restrict`
  - `volatile` is discussed in later when we introduce low-level programming.
  - `restrict` is used only for pointer variable, introduced in Advanced Uses of Pointers (5)
  - Focus on `const` in the following slides

Type Qualifiers - `const`

- `const` is used to declare "read-only" objects.
- Advantages of declaring an object to be `const`:
  - Serves as a form of documentation.
  - Allows the compiler to check that the value of the object isn't changed.
  - Alerts the compiler that the object can be stored in ROM (read-only memory).

```
const int n = 10;
const int tax_brackets[] = {750, 2250, 3750, 5250, 7000};
```

Type Qualifiers - `const`

- It might appear that `const` serves the same role as the `#define` directive, but there are significant differences between them.
- `#define` can be used to create a name for a numerical, character, or string constant, but `const` can create read-only objects of *any* type.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't.
- It's legal to apply the address operator (`&`) to a `const` object, since it has an address; a macro doesn't have an address.
- Programmers should decide which one to use based on the design of the programs.

Declarators

- In the simplest case, a declarator is just an identifier
- Declarators may also contain the symbols `*`, `[]`, and `()`.

```
int j; // simple declarator, an identifier
int *p; // begins with * represents a pointer
int a[10]; // ends with [] represents an array
float f(float); // ends with () represents a function
```

Declarators

- The brackets may be left empty if the array is a parameter, if it has an initializer, or if its storage class is `extern`
- In the case of a multidimensional array, only the first set of brackets can be empty.

```
int a[10]; // ends with [] represents an array
int b[] = {1, 2}; // initializer
extern int c[]; // extern storage class
```

Declarators

- A declarator that ends with `()` represents a function:
- C allows parameter names to be omitted in a function declaration:

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

Declarators

- Previous examples of declarators are simple.
- In fact, declarators in **actual** programs often combine the `*`, `[]`, and `()` notations.
- Here are some a bit more complicated examples of declarators. Please try to match the correct meaning to each declaration.

```
int *ap[10];

float *fp(float);

void (*pf)(int);
```

An array of 10 pointers to integers

A function that has a `float` argument and returns a pointer to a `float`

A pointer to a function with an `int` argument and a `void` return type

Deciphering Complex Declarations

convert (a text written in code, or a coded signal) into normal language. similar to decode, translate

- What about declarators like the one in the following declaration?
- It's not obvious whether `x` is a pointer, an array, or a function.

```
int *(*x[10])(void);
```

Deciphering Complex Declarations

- Rules for understanding declarations:
  - ***Always read declarators from the inside out.***
    - Locate the identifier that's being declared, and start deciphering the declaration from there.
  - ***When there's a choice, always favor*** **`[]`** ***and*** **`()`** ***over*** **`*`** **(asterisk).**
    - Parentheses can be used to override the normal priority of `[]` and `()` over `*`.
    - For example, if `*` precedes the identifier and `[]` follows it (e.g., `int *a[10]`), the identifier represents an array, not a pointer (e.g., `a` is an array)

Deciphering Complex Declarations

- Example 1:

  ```
  int *ap[10];
  ```

  `ap` is an *array* of *pointers*.

- Example 2:

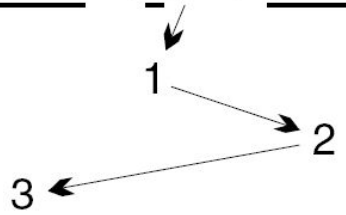
  ```
  float *fp(float);
  ```

  `fp` is a *function* that returns a *pointer*.

Deciphering Complex Declarations

- Example 3:

```
void (*pf)(int);
```

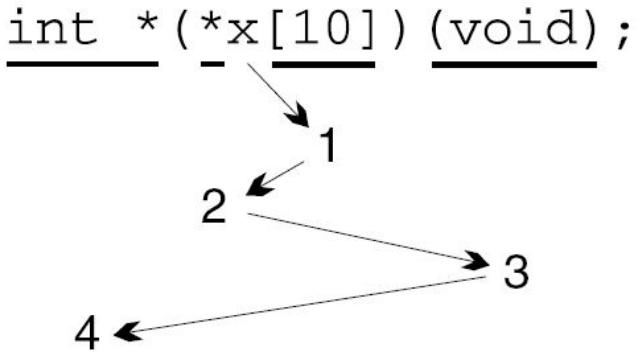


Type of `pf`:
1. pointer to
2. function with `int` argument
3. returning `void`

- Since `*pf` is enclosed in **parentheses**, `pf` must be a **pointer**.
- But `(*pf)` is followed by `(int)`, so `pf` must point to a function with an `int` argument.
- The word `void` represents the return type of this function.

Deciphering Complex Declarations

- Let's use the same rules to decipher the declaration earlier



Type of `x`:

1. array of
2. pointers to
3. functions with no arguments
4. returning pointer to `int`

Deciphering Complex Declarations

- Certain things can't be declared in C.
- Functions can't return arrays:

```
int f(int)[];       /*** WRONG ***/
```

- Functions can't return functions:

```
int g(int)(int);    /*** WRONG ***/
```

- Arrays of functions aren't possible:

```
int a[10](int);     /*** WRONG ***/
```

- In each case, pointers can be used to get the desired effect.
  - For example, a function can't return an array, but it can return a *pointer* to an array.

Initializers

- C allows us to specify initial values for variables as we're declaring them.
- To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer.
- Don't confuse the = symbol in a declaration with the assignment operator!
  - initialization is not the same as assignment

```
int i = 5 / 2;
float k = 0.0;
```

Initializers

- If the types don’t match, C converts the initializer using the same rules as for assignment
- The initializer for a pointer variable must be an expression of the same type or of type `void *`

```
int i = 5 / 2;
int j = 5.5;      /* converted to 5 */
int *p = &i; /* address of int object i */
```

Initializers

- The initializer for an array, structure, or union is usually a series of values enclosed in braces.
- We can also use designated initializers.

```
int a[5] = {1, 2, 3, 4, 5};

struct {
      int number;
      char name[NAME_LEN+1];
      int on_hand;
} part1 = {.number = 528, .name = "Disk drive", .on_hand = 10};
```

Initializers

- An initializer for a variable with static storage duration must be constant
- If `LAST` and `FIRST` had been variables, the initializer would be illegal.
  - However, we can use const to solve this error.

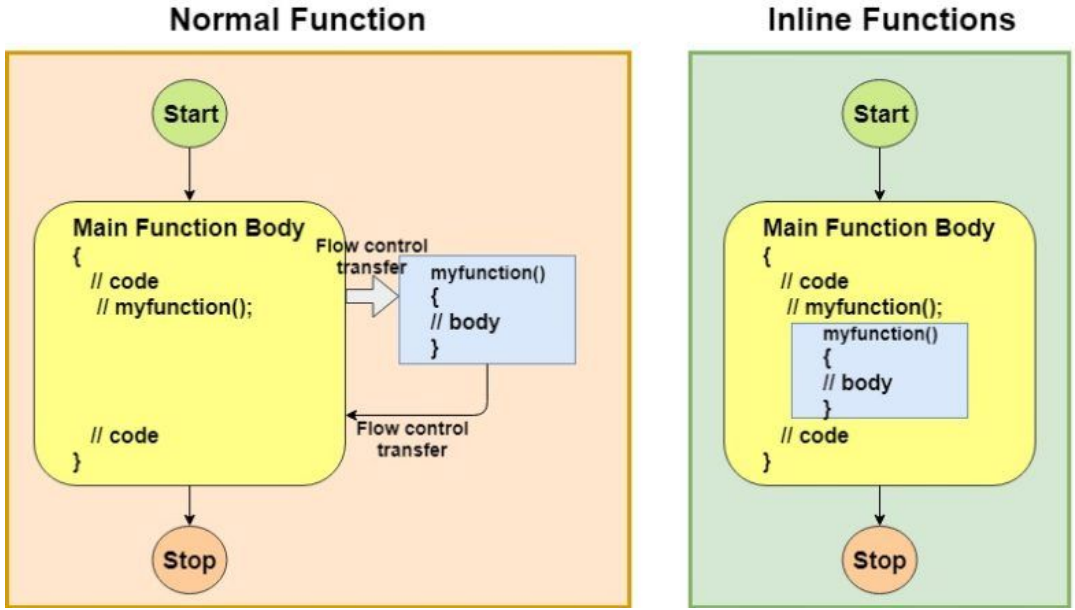
```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

Inline Functions

- **Declaration specifiers** fall into three categories:
  - Storage classes
  - Type qualifiers
  - Type specifiers
- C99 has a **fourth category**, ***function specifiers,*** which are used **only** in **function** declarations (will introduce later).
- The only keyword for function specifiers is: `inline`.
- `inline` is related to the concept of the “overhead” of a function call—the work required to call a function and later return from it.

Inline Functions

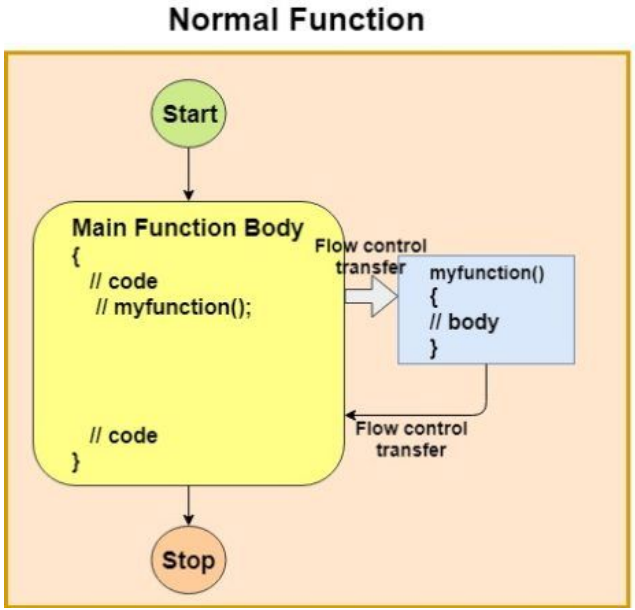
- `inline` is related to the concept of the "overhead" of a function call—the work required to call a function and later return from it.



https://simplesnippets.tech/inline-functions-in-cpp/

Inline Functions

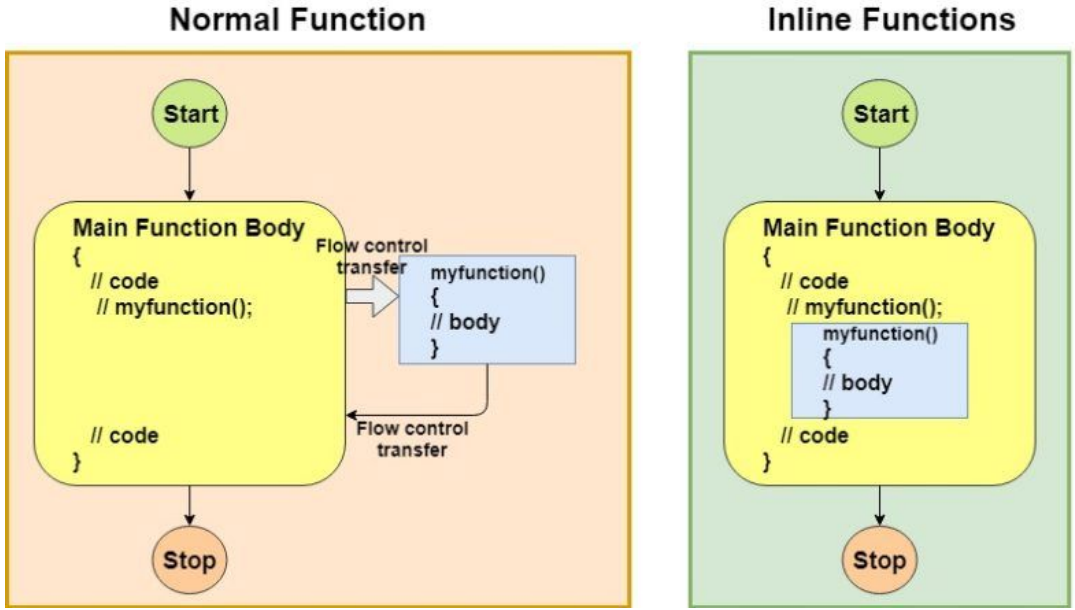
- Although the overhead of a function call slows the program by only a tiny amount, it may add up in certain situations (e.g., frequently calling the functions)



https://simplesnippets.tech/inline-functions-in-cpp/

Inline Functions

- C99 offers a better solution to this problem: create an ***inline function.***
- The word “inline” suggests that the compiler replaces each call of the function by the machine instructions for the function.



https://simplesnippets.tech/inline-functions-in-cpp/

Inline Functions

- An inline function has the keyword `inline` as one of its declaration specifiers
- Here is an example.
  - Most of the Inline functions are used for small and simple computations that are used frequently.

```
inline double average(double a, double b){
      return (a + b) / 2;
}
```

Inline Functions

- Declaring a function to be `inline` doesn't actually force the compiler to "inline" the function.
- It suggests that the compiler should try to make calls of the function as fast as possible, but the compiler is free to ignore the suggestion.
- It's also an advanced skill in C programming for optimization, so we will not go in deep into this topic in this introductory course.
- However, if you plan to use `inline`, please ensure that you understand its restriction totally!
- Please read this article and section 18.6 for more information
  - https://simplesnippets.tech/inline-functions-in-cpp/

Summary

- Declaration specifiers
  - Type The Storage Class of a Function
  - Type Qualifiers
    - `const`
- Declarators
  - Deciphering Complex Declarations
  - Rules for understanding declarations
- Initializers
- Function specifiers
  - `inline`