More about function pointers

Function name = pointer to itself

A function name (label) is converted to a pointer to itself.

```c
// add x + y
void add(int x, int y) {
  printf("%d + %d = %d\n", x, y, x + y);
}
// all are the same
void (*p_add1)(int, int) = add;     // OK
void (*p_add2)(int, int) = *add;    // OK
void (*p_add3)(int, int) = &add;    // OK
void (*p_add4)(int, int) = **add;   // OK
void (*p_add5)(int, int) = ***add; // OK
```

Function name = pointer to itself

And execution

```
(*p_add1)(1, 2); // OK
(*p_add2)(1, 2); // OK
(*p_add3)(1, 2); // OK
(*p_add4)(1, 2); // OK
(*p_add5)(1, 2); // OK

p_add1(1, 2); // OK
p_add2(1, 2); // OK
p_add3(1, 2); // OK
p_add4(1, 2); // OK
p_add5(1, 2); // OK
```

Function name = pointer to itself

And execution

```
(*p_add1)(1, 2);     // OK
(**p_add1)(1, 2);    // OK
(***p_add1)(1, 2);   // OK
(****p_add1)(1, 2);  // OK

(&p_add1)(1, 2); // NOT OK - ERROR
```

Function arguments

```c
// prototype of function that takes
// undetermined number of arguments
void foo();

// prototype of function that takes
// no arguments
void foo(void);
```

Function arguments

```
// add x + y
void add(int x, int y) {
  printf("%d + %d = %d\n", x, y, x + y);
}
// add 1 + 2
void add1and2() {
  printf("1 + 2 = 3\n");
}
```

Function arguments

```
int main() {
    void (*p_add)(); // num args undetermined

    p_add = add;
    p_add(1, 2);

    p_add = add1and2;
    p_add();
    p_add(3, 4); // OK, but prints “1 + 2 = 3”

}
```

Function arguments

```
int main() {
    void (*p_add)(int, int);

    p_add = add;
    p_add(1, 2);

    p_add = add1and2; // OK
    p_add(); // ERROR – too few arguments
    p_add(3, 4); // OK, but prints “1 + 2 = 3”
}
```

Function arguments

```c
// add x + y
void add(int x, int y) {
  printf("%d + %d = %d\n", x, y, x + y);
}
// add 1 + 2
void add1and2(void) {
  printf(“1 + 2 = 3\n");
}
```

Function arguments

```
int main() {
    void (*p_add)(); // num args undetermined

    p_add = add;
    p_add(1, 2);

    p_add = add1and2; // OK
    p_add();
    p_add(3, 4); // OK, but prints “1 + 2 = 3”

}
```

Function arguments

```
int main() {
    void (*p_add)(int, int);

    p_add = add;
    p_add(1, 2);

    p_add = add1and2; // WARNING - incompatible
                      //           types
    p_add(); // ERROR – too few arguments
    p_add(3, 4); // OK, but prints “1 + 2 = 3”
}
```

Function arguments

```
int main() {
   void (*p_add)(void);

   p_add = add; // WARNING - incompatible types
   p_add(1, 2); // ERROR - too many arguments
   p_add(); // OK, but prints garbage

   p_add = add1and2;
   p_add(); // OK
   p_add(3, 4); // ERROR - too many arguments
}
```

Uninitialized function pointers

```
int main() {
   void (*p_add)(void);

   p_add(); // WARNING – uninitialized
            // when used
}
```

=> Most likely will cause the program to crash

Object Oriented like programming in C
“a CPP preview”

Emulating methods for OOP in C

Look at the following code, it looks like we are using some String class, and setting its value to “hello”.

```
String s1 = newString();
s1->set(s1, "hello");
```

How can this be done in C?

Dog – simple example

```c
typedef struct Dog Dog;
struct Dog{
   int weight;
   void (*eat)(Dog *this, int pounds);
   void (*sleep)(Dog *this, int hours);
};
void dog_eat(Dog *d, int pounds) { //eat
   d->weight += pounds/2;
}
void dog_sleep(Dog *d, int hours) { //sleep
   d->weight -= hours/3;
}
```

Dog – simple example

```c
void init_dog(Dog *d) {
    if (d != NULL) {
        d->weight = 1;
        d->eat = dog_eat;
        d->sleep = dog_sleep;
    }
}
```

Dog – simple example

```
int main() {
    Dog beethoven;
    init_dog(&beethoven);
    while(beethoven.weight < 200) {
        beethoven.eat(&beethoven, 10);
        beethoven.sleep(&beethoven, 10);
    }
    return 0;
}
```

Shapes – example with polymorphism

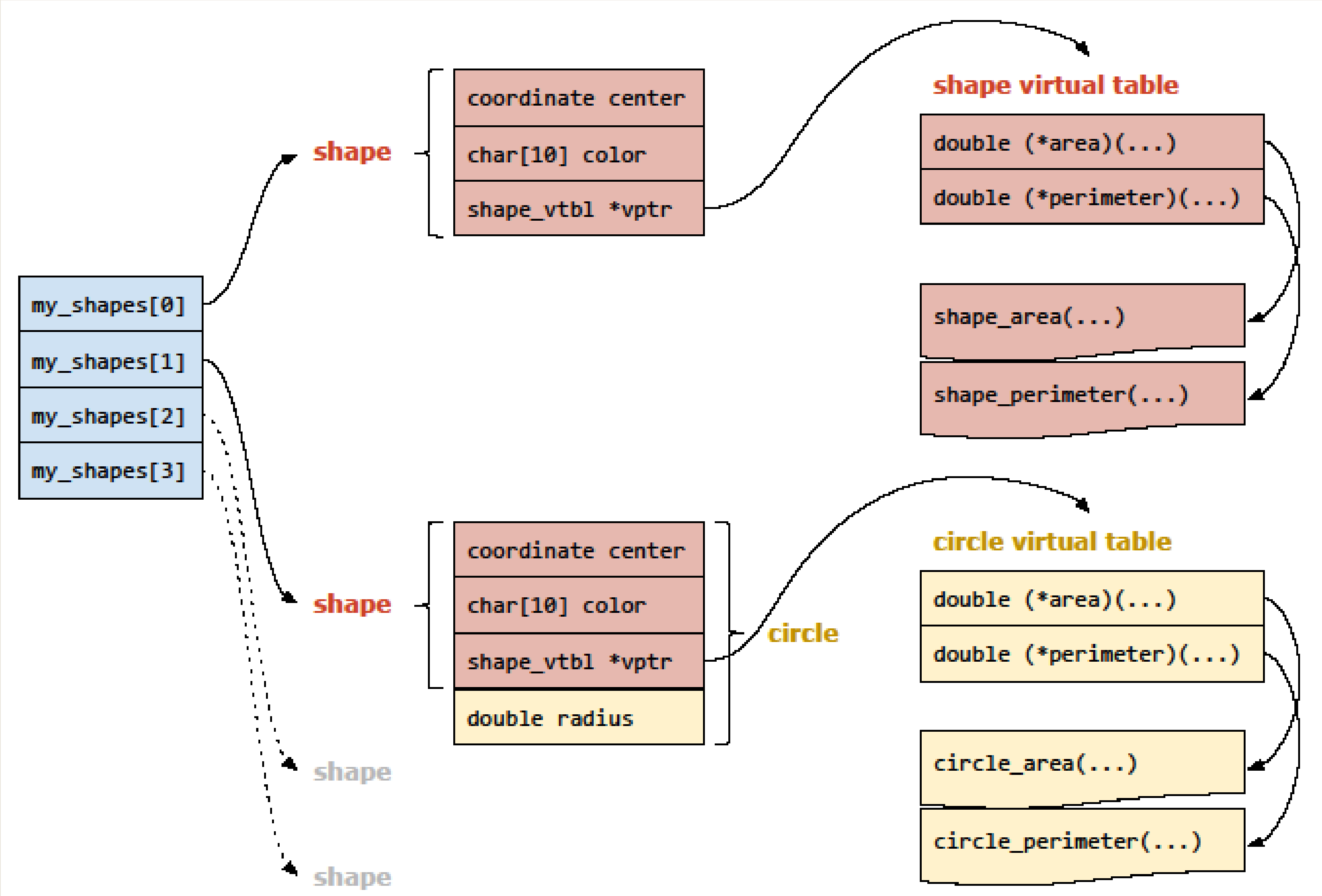
```
See code in course site… (oop_code.zip)
```

Example adopted from

http://www.embedded.com/electronics-blogs/27/Programming-Pointers

By **Dan Saks**

Shapes – example with polymorphism



Shapes – example with polymorphism

- How would you change the code such that the following would work in main (without any memory leak)?
- What would you need to do if the color field was dynamically allocated?

```
int main(void) {
   shape* my_shapes[4];
   my_shapes[0] = new_shape((coordinate){.x=0,.y=0}, "red");
   my_shapes[1] = (shape*)new_circle((coordinate){.x=0,.y=0}, "blue", 1.0);
   my_shapes[2] = (shape*)new_rectangle((coordinate){.x=0,.y=0}, "yellow", 3.0, 4.0);
   my_shapes[3] = new_shape((coordinate){.x=0, .y=0}, "green");
   for (int i = 0; i < 4; ++i) {
      my_shapes[i]->vptr->area(my_shapes[i]);
      my_shapes[i]->vptr->perimeter(my_shapes[i]);
   }
   for (int i = 0; i < 4; ++i) {
      my_shapes[i]->vptr->delete(my_shapes[i]);
   }
   return 0;
}
```

Syntax – a bit more

Statements - conditional

```
if (condition)
//statement or block
else
//statement or block

switch (integer value)
{
   case 3:
         //statement or block
   break; //optional. Otherwise fall-through until the '}'!
   case 5:
         //statement or block
   default:
         //statement or block
}
```

Statements - conditional goto

```
goto bla;
whee:
  //code here
bla:
  //code there
```

DANGER: SPAGHETTI CODE!

Avoid whenever possible.

Q: When to use goto?

A: There are cases. e.g. break from a lot of nested loops, forward jump to error handler

New in C99

- Compound Literals
- Designated Initializers
- Variable-Length Array (VLA)

Compound Literals

```c
struct foo {int a; char b[2];} my_struct;

// initializing my_strcut
struct foo temp = {x + y, 'a', 0};
my_struct = {x + y, 'a', 0}; // ERROR not allowed
my_struct = temp; // OK

// using Compound Literals
// (looks like a cast containing an initializer)
my_struct = ((struct foo) {x + y, 'a', 0});
```

Compound Literals

```
// can also construct an array
char **foo = (char *[]) { "x", "y", "z" };
```

Be careful about duration!

Don’t use in CPP

Designated Initializers

- In C90 - the elements of an initializer need to appear in a fixed order (the same as the order of the elements in the array or structure being initialized)
- In C99 you can give the elements in any order, specifying the array indices or structure field names they apply to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
int a[6] = { [4] = 29, [2] = 15 }; // equivalent

struct point {int x, y; };
struct point p = { .y = yval, .x = xval };
struct point p = { xval, yval }; // equivalent
```

Variable-Length Array (VLA)

How to allocate an array with size defined only at run time (instead of compile time)?

ANSI C - use malloc:

```
int length = atoi(argv[ARR_LENGTH]);
int *a = (int *) malloc(length * sizeof(int));
```

C99 - introduce VLA:

```
int length = atoi(argv[ARR_LENGTH]);
int a[length];
```

Variable-Length Array (VLA)

Also…

```
void foo(int x, int y, int a[x][y]) {
    ...
}

int main() {
    int x, y;
    ...
    int a[x][y];
    foo(x, y, a);
    return 0;
}
```

VLA - disadvantages

- Underlying memory allocation is not specified: the VLA may be allocated on the stack, heap (or none…).
- For example, GNU C Compiler allocates memory for VLAs on the stack
- Not accepted by C++ standard (use `std::vector` instead)
- **Not allowed during the course - compile with -Wvla**

Unions

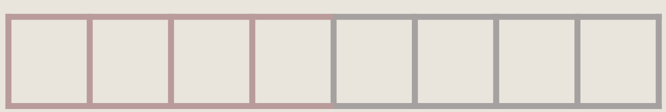
union

- A **union** is data type that enables you to store different data types in the same memory location
- The value of at most one of the data members can be stored in a union at any time
- Unions provide an efficient way of using the same memory location for multi-purpose
- The size of a union is sufficient to contain the largest of its data members
- Each data member is allocated as if it were the sole member of a struct

union

Union definition:

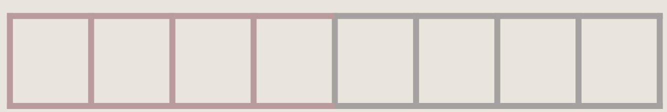
```
typedef union u_all
{
    int i_val;
    double d_val;
} u_all;
```



union

```
u_all u; //definition of the variable u
u.i_val= 3;//assignment to the int member of u
printf("%d\n", u.i_val);

u.d_val= 3.22; //this corrupts the previous
                  assignment
printf("%f\n", u.d_val);
```



It is your responsibility to remember which union member is currently assigned!

Using union for lexers

See also: http://stackoverflow.com/a/740686

You have different tokens, each related to different value

```
typedef enum {KEYWORD, INT_CONST, STRING_CONST} TokenType;
typedef struct Token {
  TokenType type;
  union {
    struct { int line; } noVal;
    struct { int line; int val; } intVal;
    struct { int line; struct string val; } stringVal;
  } data;
} Token;
```

It is your responsibility to remember which union member is currently assigned!

Using union for lexers

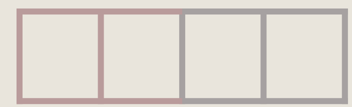
```
int main() {
  Token t = parse_a_token(...);
  switch (t.type) {
    case KEYWORD:
       ...
    case INT_CONST:
       printf("line %d, INT, v=%d\n",
              t.data.intVal.line, t.data.intVal.val);
       break;
    default:
       ....
  }
  return 0;
}
```

Remember: it is your responsibility to remember which union member is currently assigned!

Using unions for low-level coding

<u>If</u> supported by the compiler, you can write something like this

```
union float_cast
{
    unsigned short as_short[2];
    float f_val;
};
```



So you can look at the bits representing the float value, as two consecutive shorts (if not supported, behavior is undefined).

Bitwise operators & Bitfields

Bitwise operators

- C allows to perform operations on the bit representation of a variable
- Done with “bitwise operators”

Operator symbol	Operation
&	and
|	or
^	xor
~	One complement (not)
<<	Left shift
>>	Right shift

Bitwise operations to manipulate single bits

- Each char can represent 8 independent boolean variables
- For this we need to find a way to
  - Find the value of a single bit
  - Change the value of a single bit

Finding the value of a single bit

We can do that with the **<<** and **&** operators:

```
int isZero (int variable, int position)
{
    return (variable & (1 << position)) == 0;
}
```

Example:

- We have a variable with the bit-pattern: **11001**
- We want to test the value of the 3rd bit (from the least-significant bit, starting from place 0, this is place 2):

Construct a mask: `0001 << 2 == 00100`

Apply: `11001 & 00100 == 00000 == 0`

Toggling a single bit

Use the **<<** and **^** operators:

```
int toggleBit (int variable, int position)
{
    return variable ^ (1 << position);
}
```

Example:

- We have a variable with the bit-pattern: **11001**
- We want to change the value of the 3rd bit (from the least-significant bit, starting from place 0, this is place 2):

Construct a mask: `0001 << 2 == 00100`

Apply: `11001 ^ 00100 == 11101`

Bit fields

Define and manipulate single bits in a simple way:

Bitfields are defined **inside** a struct/union

```
struct
{
    type [member_name] : width ;
    …
};
```

- `type: int / signed int / unsigned int` determines how to interpret the bits (for > 1).
- `width:` the number of bits used ≤ #bits of `type.`

Bit fields

Define a struct of bits:

```
struct flags
{
    unsigned int _is_keyword : 1;
    unsigned int _is_extern : 1;
    unsigned int _is_static : 1;
};
```

usage:

```
struct flags f;
f._is_extern = f._is_static = 1;
struct flags * p = &f;
unsigned int * p_static = &(f._is_static) // error
```

Bit fields

Define a struct of bits:

```
struct flags
{
   unsigned int _is_keyword : 1;
   unsigned int _is_extern : 1;
   unsigned int _is_static : 1;
};
```

usage:

```
sizeof(struct flags) → 4
```

Bit fields

Define a struct of bits:

```
struct flags
{
    unsigned int _is_keyword : 1;
    unsigned int _is_extern : 31;
    unsigned int _is_static : 1;
};
```

usage:

```
sizeof(struct flags) → 8
```

Bit fields

Define a struct of bits:

```
struct flags
{
    unsigned int _is_keyword : 1;
    unsigned int _is_extern : 32;
    unsigned int _is_static : 1;
};
```

usage:

```
sizeof(struct flags) → 12
```

Variadic functions

Variadic functions

A variadic function is a function with **variable number of arguments.**

Example:

```
// 1 argument
printf("Hello world\n");
// 2 arguments
printf("The value of i is: %d\n", i);
// 3 arguments
printf("The value of i is: %d\t its address
        is: %p\n", i, &i);
```

Variadic function definition

```
<return type> <function name>(<first parameter>,...)
```

Examples:

```
//std lib. function
int printf(const char *format, ...)

//function we define
int countIntegers(int integerNum, ...)
```

The main challenge:

- Since we do not know the function parameters in advance, we can’t give them names.

- We have to develop a technique to access the variables based on their ***type and place*** relative to the first parameter.

stdarg.h

To define variadic functions we should include the header **stdarg.h**

- definition of the type: `va_list`
  - A type for iterating the arguments (most likely `void*`)

- definition of the macros:
  - `va_start` – start iterating the argument lists with `va_list` and the last named parameter of the function
  - `va_arg` – retrieve the current argument
  - `va_end` – free the `va_list`

Example

```
#include <stdarg.h>
int sumInts(int argsNum, …)
{
    va_list ap;
    int i, sum=0;
    va_start(ap, argsNum); // now we point to the place
                           // right after the first argument
    for(i = 0; i < argsNum; ++i) {
      sum += va_arg(ap,int); // access current argument and
                             // move ap to the next argument
    }
    va_end(ap); // free ap
    return sum;
}
//in main:
sumInts(5,1,1,2,3,4) //returns 11
sumInts(2,7,1) //returns 8
```

Note!

- Variadic functions must have <u>at least</u> one named parameter
  `void wrong(...);`
- There is no mechanism defined for determining the number or types of the unnamed arguments – we can use one of the following ways:
  - **format string** (like in printf)
  - **sentinel value** at the end of variadic arguments
  - **count argument** indicating the number of the variadic arguments

snprintf & vsnprintf

New in C99, **`snprintf`** and **`vsnprintf`** are nice examples of variadic functions

```
int snprintf (char *s, size_t n,
              const char * format, ...)

int vsnprintf(char *s, size_t n,
              const char * format, va_list arg)
```

Read example at:
http://www.cplusplus.com/reference/cstdio/vsnprintf/

Optimization

What smart people say about it...

Premature optimization is the root of all evil (or at least most of it) in programming

– Donald Knuth

(check the “humor” part in his wiki)

So, what to do?

- ✓ Check if you **need to optimize**
- ✓ Profile - check **where to optimize**
(gprof if you use gcc; need to add –g and –pg to the compilation line)
- ✓ Remember to **“turn off” debugging**
(gcc -DNDEBUG)
- ✓ Check what your compiler can do for you on your **specific hardware**
(-O3, -mcpu=arm7, etc…)

- ➢ We'll learn more about optimization in **labcpp**

Cache-friendly coding

- Fast memory is expensive, so we have very little of it
- Goal: achieve good performance with what we have
- Method: memory is organized in a hierarchic order:
  - Registers
  - Cache (several levels)
  - RAM
- Search from the fastest to the slowest memory
- Fetch important memory before it is used!
  How? When a given location is accessed, most likely its neighborhood will be accessed in the near future

Cache-friendly coding. Iterating 2D array

- C requires “row-major ordering”

```c
//efficient
int i,j;
int arr[ROWS_NUM][COLS_NUM];
for(i=0;i<ROWS_NUM;++i){
  for(j=0;j<COLS_NUM;++j){
    arr[i][j] = i*j;
  }
}
//not efficient
int arr[ROWS_NUM][COLS_NUM];
for(i=0;i<COLS_NUM;++i){
  for(j=0;j<ROWS_NUM;++j){
    arr[j][i] = i*j;
  }
}
```

New in C99

- `register` keyword
- `volatile` keyword

Registers

Register (אוגר) - is a small amount of storage available on the CPU

There are a few registers per CPU

Register Variables

The **register** keyword specifies that the variable is to be stored in a CPU register, **if possible.**

A <u>recommendation</u> to the compiler

```
register int i;
```

Provides quick access to **commonly used values**

Experience has shown that the **compiler usually knows much better than humans what should go into registers and when ☺**

Volatile - Motivation

Sometimes variables can be **modified externally** (not from the currently compiled piece of code)

The compiler cannot “guess” it!

Volatile example

Before optimization

```
void foo(void)
{
    int* addr;
    addr = INPUT_ADDRESS;
    *addr = 0;

    while (*addr != 255)
    ;
}
```

Volatile example

After optimization

```
void foo(void)
{
    int* addr;
    addr = INPUT_ADDRESS;
    *addr = 0;

    while (1)
    ;
}
```

Volatile example

After optimization using volatile

```
void foo(void)
{
   volatile int* addr;
   addr = INPUT_ADDRESS;
   *addr = 0;

   while (*addr != 255)
   ;
}
```

Volatile - Summary

Variables declared to be volatile **will not be optimized by the compiler**

This is due to our assumption that their value can change at any time

Note that this does not mean we can use volatile variables for synchronization – they are not atomic or impose any specific order of operations

Volatile - Summary

Use for:

- ✓ Memory-mapped peripheral registers
- ✓ Global variables (signal variables) modified by an interrupt service routine, or setjmp/longjmp
- ✓ Global variables accessed by multiple tasks within a multi-threaded application

New in C99

- `inline` functions
- `restrict` keyword

inline functions

- Tell the compiler to substitute calls for the function body by inline expansion - meaning, by inserting the function code
- This helps save the overhead of function invocation and return (placing data on stack and retrieving the result)
- However, this may result in a larger executable as the code for the function has to be repeated multiple times

inline functions

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}

// somewhere in the code ...
a = max(x, y);

// may actually be compiled as ...
a = (x > y) ? x : y;
```

inline functions

- The `inline` specifier is only a hint for the compiler to perform optimizations. Compilers can (and usually do) ignore the presence or absence of the `inline` specifier for the purpose of optimization
- Identifiers and macros used in the function refer to the definitions visible at the point of definition, not at the point of call
- Inline function must be defined in the same translation unit (so we usually just define the function in the header)
- If an external definition also exists in the program, it's unspecified whether the inline definition (if present in the translation unit) or the external definition is called
- Read more at home… http://en.cppreference.com/w/c/language/inline

inline functions vs macros

Traditionally inline expansion was accomplished at the source level using parameterized macros.

True inline functions provides several benefits over this approach…

http://en.wikipedia.org/wiki/Inline_function

inline functions vs macros

- Macro invocations do not perform type checking
- A macro cannot use the return keyword with the same meaning as a function would do (it would make the function that asked the expansion terminate, rather than the macro). In other words, a macro cannot return anything which is not the result of the last expression invoked inside it
- C macros use mere textual substitution, which may result in unintended side-effects and inefficiency due to re-evaluation of arguments and order of operations

http://en.wikipedia.org/wiki/Inline_function

inline functions vs macros

- Compiler errors within macros are often difficult to understand, because they refer to the expanded code, rather than the code the programmer typed
- Many constructs are awkward or impossible to express using macros, or use a significantly different syntax. Inline functions use the same syntax as ordinary functions, and can be inlined and un-inlined at will with ease
- Many compilers can also inline expand some recursive functions (up to some depth); recursive macros are typically illegal

http://en.wikipedia.org/wiki/Inline_function

inline functions vs macros

Bjarne Stroustrup, the designer of C++, likes to emphasize that macros should be avoided wherever possible, and advocates extensive use of inline functions.

http://en.wikipedia.org/wiki/Inline_function

restrict keyword

- A keyword that can be used in pointer declarations
- It says that for the lifetime of the pointer, only it or a value directly derived from it (such as `pointer + 1`) will be used to access the object to which it points
- This limits the effects of pointer aliasing, aiding optimizations
- If the object is accessed by an independent pointer, this will result in undefined behaviour

http://en.wikipedia.org/wiki/Restrict

restrict keyword

```
void updatePtrs(size_t *ptrA, size_t *ptrB,
                size_t *val) {
  *ptrA += *val;
  *ptrB += *val;
}
```

```
load R1 ← *val ; Load the value of val pointer
load R2 ← *ptrA ; Load the value of ptrA pointer
add R2 += R1 ; Perform Addition
set R2 → *ptrA ; Update the value of ptrA pointer
; Similarly for ptrB, note that val is loaded twice,
; because ptrA may be equal to val.
load R1 ← *val
load R2 ← *ptrB
add R2 += R1
set R2 → *ptrB
```

http://en.wikipedia.org/wiki/Restrict

restrict keyword

```
void updatePtrs(size_t *restrict ptrA, size_t *restrict ptrB,
                size_t *restrict val) {
  *ptrA += *val;
  *ptrB += *val;
}
```

The compiler is allowed to assume that `ptrA`, `ptrB`, and `val` point to different locations, and updating one pointer will not affect the others

http://en.wikipedia.org/wiki/Restrict

restrict keyword

```
void updatePtrs(size_t *restrict ptrA, size_t *restrict ptrB,
                size_t *restrict val) {
  *ptrA += *val;
  *ptrB += *val;
}
```

```
load R1 ← *val ; Load the value of val pointer
load R2 ← *ptrA ; Load the value of ptrA pointer
add R2 += R1 ; Perform Addition
set R2 → *ptrA ; Update the value of ptrA pointer
; Note that val is not reloaded,
; because the compiler knows it is unchanged.
load R1 ← *val
load R2 ← *ptrB
add R2 += R1
set R2 → *ptrB
```

http://en.wikipedia.org/wiki/Restrict