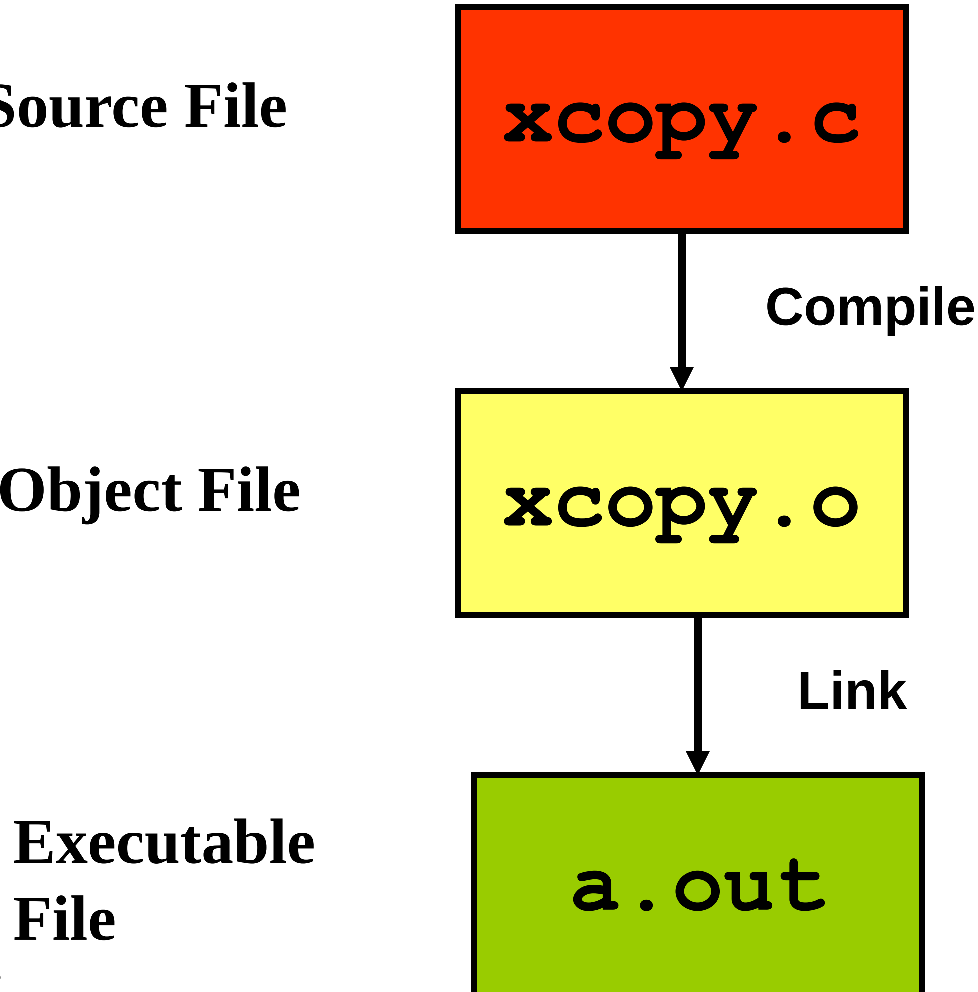
Chapter 6

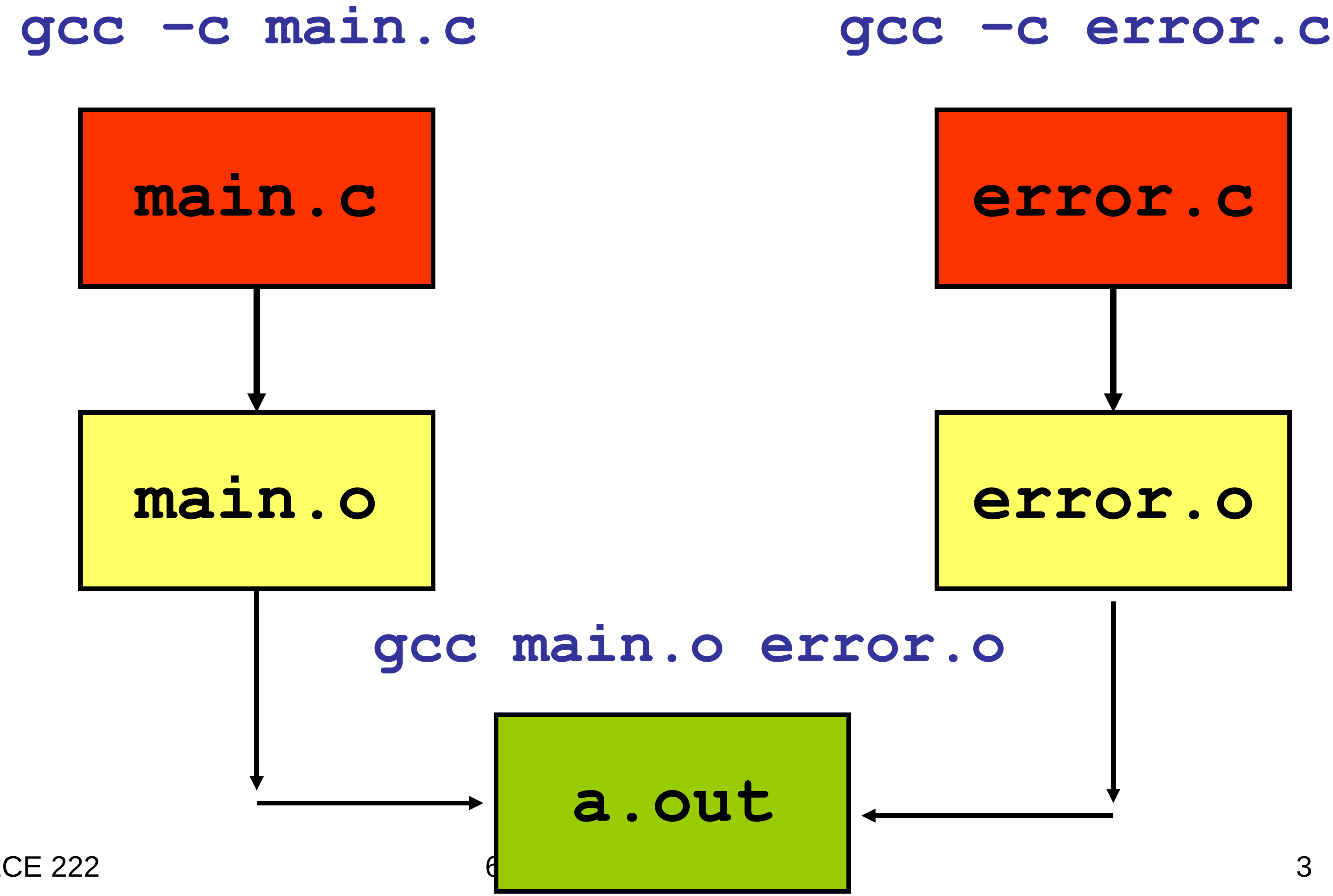
Program Management

Program Building

gcc xcopy.c

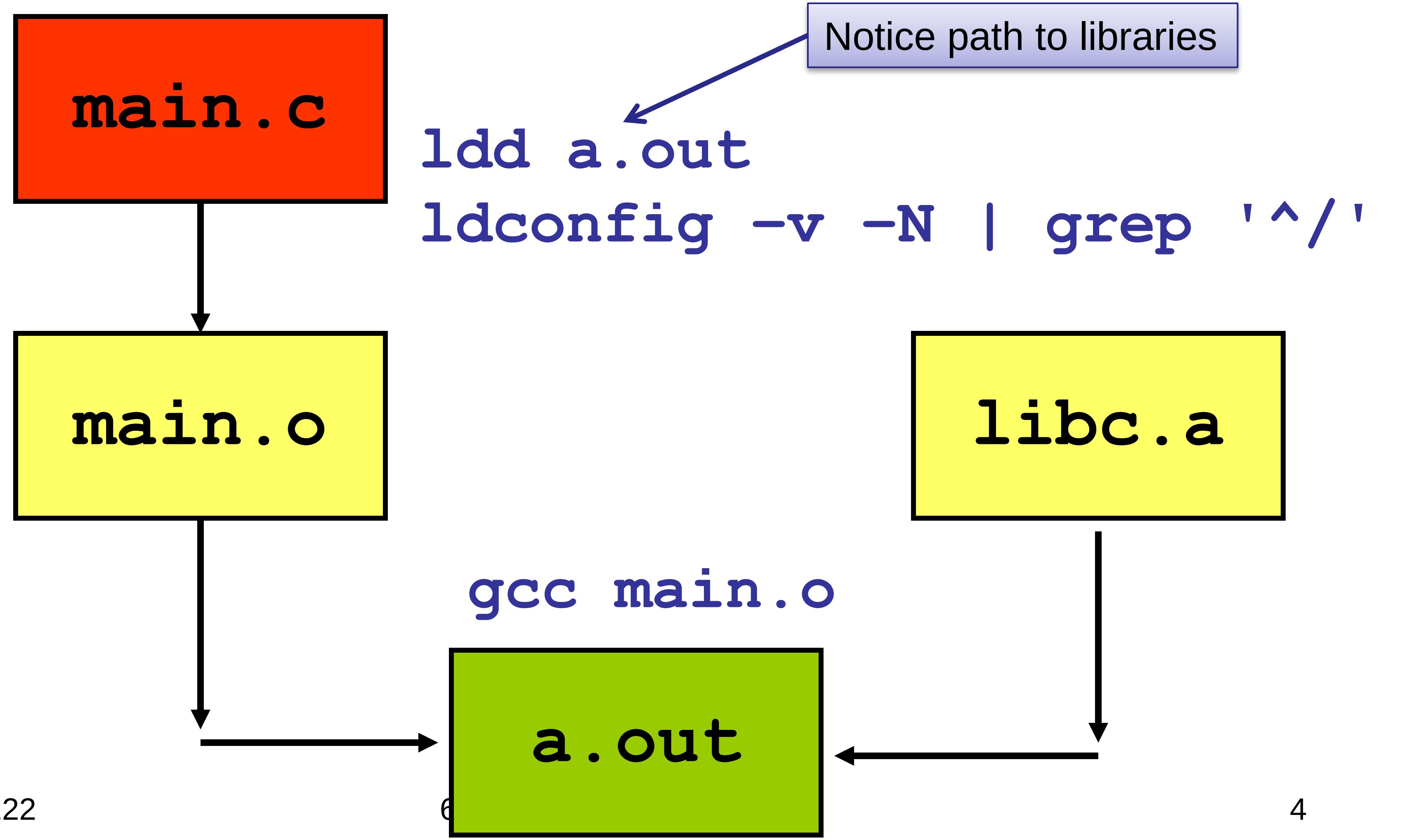


Compiling Object Code



Linking Libraries

`gcc -c main.c`



Libraries

Consider looking at the following library.

```
ls -all /usr/lib/libc.a
ar t /usr/lib/libc.a
ar t /usr/lib/libc.a | grep print
ar t /usr/lib/libc.a | grep string
```

What's the difference between these two statements?

```
gcc main.c
gcc -static main.c
```

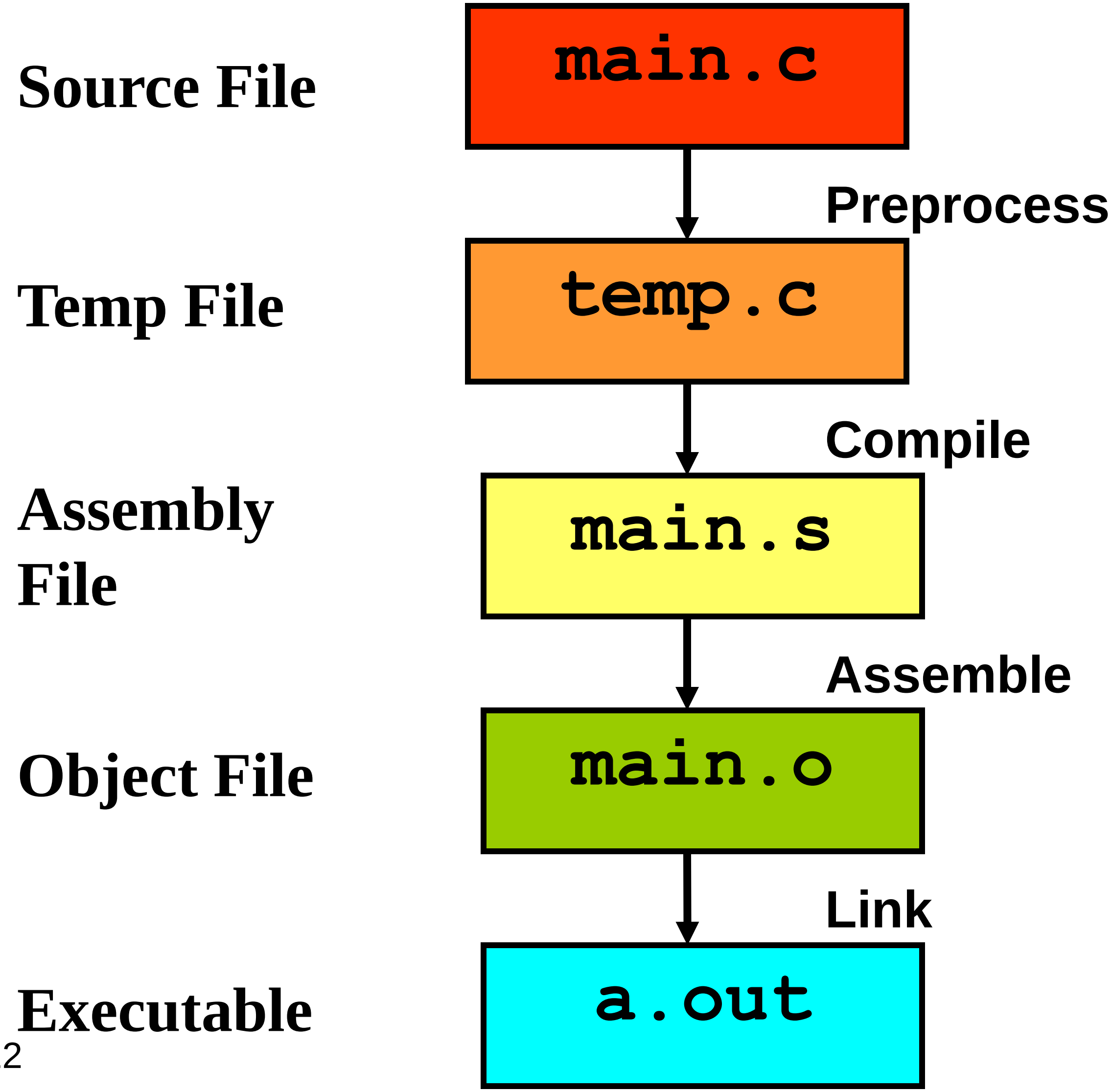
Linking

What are the differences between dynamic and static linking of libraries?

- **Dynamic linking produces smaller code.**
- **Static linking produces faster code.**

Compiling a Program

`gcc main.c`



Preprocessing

You can ask the compiler to stop after preprocessing and show the results using the `–E` switch.

```
gcc –E preprocess.c
```

or

```
gcc –E preprocess.c –o temp.c
```

What are some advantages of preprocessing?

Preprocessing Example

```
#define BIT_0    0x01
#define BIT_1    0x02
#define BIT_2    0x04
#define BIT_3    0x08
#define BIT_4    0x10
#define BIT_5    0x20
#define BIT_6    0x40
#define BIT_7    0x80

#define SetBit(x, y) x |= (0x01 << y)
#define ClrBit(x, y) x &= ~(0x01 << y)
#define TglBit(x, y) x ^= (0x01 << y)
#define TstBit(x, y) (x & (0x01 <<y))

int main(void)
{ unsigned char a = 0x5A;

  printf("a = 0x%02X\n", a);

  SetBit(a, 0);
  printf("a = 0x%02X\n", a);
...
```

define.c

Notice: Macros

Assembly Code

You can ask the compiler to stop after compiling stage using the `-S` switch.

The output has a `.s` extension.

```
gcc -S main.c
```

Object Code

You can ask the compiler to stop after assembly stage without linking using the `-c` switch. The output has a `.o` extension.

```
gcc -c main.c
```

Display Version

You can ask the compiler to display all the commands used to compile the file and provide the compiler and assembler versions by using the `-v` switch.

```
gcc -v main.c
```

6.2 Organizing Code

There are several methods for structuring and modularizing code in order to make it more readable and easier to debug.

- **Breaking up Code into Functions.**
- **Breaking up Code into Multiple Files.**
- **Commenting Code.**
- **Use Naming Conventions.**
- **Using Preprocessing Directives.**
- **Using `typedefs`.**

Multiple Modules

When breaking up code into multiple files, one must understand the various properties of variables.

Type – `char`, `int`, `float`, `double`, ***etc...***

Modifier – `signed`, `short`, `long`, ***etc...***

Qualifier – `const`, `volatile`, `register`, ***etc...***

Storage Class – `auto`, `static`, `extern`, **affecting the scope of a variable.**

Variable Scope - `auto`

`auto` – Default storage class, visible within function, i.e. "local" variable.

```
float hypsin(float y)
{ auto float x, z;
   ...
}
float hypcos(float x)
{ auto float y, z;
   ...
}
int main(void)
{ auto int i, float x, y;
   ...
```

`auto.c`

Variable Scope – `static` 1

`static` – (1) When used globally, the variable is visible to all functions within the file. (i.e. "global," but only to that file.)

```
static int i;
void hypsin(int i)
{ printf("%f\n", (exp(i) - exp(-i))/2};
}
int main(void)
{ for(i=0; i<10; i++)
   { hypsin(i); }
}
```

`static1a.c`

`static1b.c`

Variable Scope – `static` 2

`static` – (2) When used locally, the variable retains its value between function calls. (i.e. stored “globally” but only visible to the function.)

```
void hypsin(void)
{ static int j;
  j++;
  printf("%f\n", (exp(j) - exp(-j))/2};
}

int main(void)
{ int i;
  for(i=0; i<10; i++) hypsin();
}
```

static2.c

static3.c

static4.c

Variable Scope – `extern`

- `extern` – Variable is visible across all files.
  - The keyword `extern` can be used in header files to allow the user to have knowledge of variables and functions used in other modules.
- External and static variables are initialized to zero by default (automatic variables are not)

```
gcc extern1.c extern2.c –o extern
```

Preprocessing Directives

The `#define` directive can be used to aid programming in many ways.

Readability – Textual substitution can make code much easier to read.

```
#define BIT_0 0x01
#define BIT_1 0x02
  ...
#define HEATER BIT_1
#define SetBit(x,y) (x |= y)
  ...
SetBit(RelayByte, HEATER);
```

Notice: a Macro

Preprocessing Directives 2

The `#define` directive can be used to aid programming in many ways.

Scalability – Textual substitution allows for code to scale up or down easily.

```
#define ROWS  2
#define COLS 16
  ...
char DisplayValue[ROWS][COLS];
  ...
for (i=0; i<ROWS; i++)
{ DisplayValue[i][0] = '\0';
```

Preprocessing Directives 3

What is the difference between the `#define` directive and the `const` declaration below?

```
#define ROWS  2
#define COLS 16

const int ROWS 2
const int COLS 16
```

Preprocessing Directives 4

- The `#define` directive can be used to aid programming in many ways.
- Portability – Non-standard C types to be easily "ported" from one compiler or processor to another.

```
#define UCHAR  unsigned char
#define SINT16 signed short int
#define UINT32 unsigned int

UCHAR a, b; SINT16 i; UINT32 j;
```

Preprocessing Directives 5

The `#define` directive can be coupled with the `#ifdef` and `#ifndef` directives to do conditional compiling.

```
#ifdef DEBUG
printf("Value at 1 = %d", i)
#endif

#ifdef WINDOWS
#define sigsetjmp  setjmp
#define siglongjmp longjmp
#endif
```

Preprocessing Example

```
/*********************************************
 *   Copyright (C) 1999-2006 by  ZiLOG, Inc.
 *   All Rights Reserved
 *********************************************/
```

ez8.h

```
#ifndef EZ8_H
#define EZ8_H

#if defined(_Z8ENCORE_F642X) || defined(_Z8ENCORE_64K_SERIES)
#define _Z8F642
#endif

#if defined(_Z8ENCORE_F640X) || defined(_Z8ENCORE_640_FAMILY)
#define _Z8F640
#endif
…
#if defined(_Z8F640) || defined(_Z8F642)
#define EZ8_SPI
#define EZ8_ADC
#define EZ8_TIMER3
#define EZ8_PORT4
#define EZ8_I2C
#endif
```

`#define` Warning

Remember that `#define` is a pre-processing directive which simply does textual substitution before compiling.

Consider the following code. *What is the problem?*

```
#define DEBUG

#ifdef DEBUG
#define print(n) printf("%d", n); fprintf(fout, "%d", n);
#else
#define print(n) printf("%d", n);
#endif

/*…*/

for (i=0; i<5; i++) print(i)
```

```
{ printf("%d", i); fprintf(fout, "%d", i); }
```

Two statements

No { }

```
for (i=0; i<5; i++) printf("%d", i); fprintf(fout, "%d", i);
```

`#define` Warning 2

What is the difference between `f1` and `f2` below?

What is the output?

```
#define f1(x,y) (x*y)/(x+y)

float f2(float x, float y)
{ return (x*y)/(x+y); }

int main(void)
{ float a, b, c, d;
  a = 1;
  b = 2;
  c = f1(++a, b);
  printf("a = %f, c = %f\n", a, c);

  a = 1;
  b = 2;
  d = f2(++a, b);
  printf("a = %f, d = %f\n", a, d);
```

Incremented twice!

```
c = (++a*b) / (++a+b)
```

Typedefs

The `typedef` statement can also be used to make programs “portable.”

That is, they can be used the same was as `#define` to re-define types.

```
typedef unsigned char      UCHAR;
typedef signed short int SINT16;
typedef unsigned int        UINT32;
```

How is the above code different from the `#define` statements?

Typedefs 2

The `typedef` statement can also be used to simply make programs more “readable” reducing text in code.

```
struct T_Person
{ char name[80]; int age;
};
typedef struct T_Person Person;
typedef struct T_Person *pPerson;
...
pPerson John;
John = (pPerson)malloc(sizeof(Person));
```

Comments

Program – Contains program function, authorship, revision history and to-do list.

Function – Contains function usage along with description of parameters and return value.

Block – Describes sub-tasks.

Line – Describes purpose of each line to algorithm.

Variable – Describes purpose and usage of variable.

Variable Names

Variable names should describe the function or purpose of the variable.

In some cases it is acceptable to use arbitrary names like `i` for an index loop. But this should only be done if the usage is obvious and unambiguous.

A programmer should always inform the reader of the function of a variable by naming it properly. Well-chosen names make help make the code "**self-commenting.**"

Function Names

Function names should also describe the function or purpose of the variable.

Names of related functions should be consistent and structured.

```
LCD_Init()      KB_Init()
LCD_Write()     KB_GetChar()
LCD_Clear()     KB_Flush()
```

Again, well-chosen names help make the code "self-commenting."