Applied Programming

Performance Optimization Techniques

Program Optimization

"There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.
We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified."

Donald Knuth, Structured Programming with go to Statements, ACM Computing Surveys, Vol 6, No. 4, Dec. 1974

Optimization of C Programs

- Modern microprocessors are designed to perform many operations in parallel
- There are ***two main ways to optimize a program*** to take advantage of the of the processor architecture and the structure of the program:
  - Using Compiler Optimization
  - Optimizing the code “by hand”

Optimization of C Programs

There are many other things you can do to optimize your code… but before optimizing it you should "profile it"

Further References

- **Applied C Programming (*Steve Oualline*)**

  Chapter 15 (pp. 256-264)

- **Chapter 5: Optimizing Program Performance (in MyCourses)**

Optimization of C Programs

- **Loop ordering**
  - **Order nested loops with the simplest outermost**
- **Reduction in Strength**
  - **Use cheap operations (see table in next slide)**
- **Powers of 2 of Integers**
  - **Do powers of 2 by shifting bits (>> or <<)**
- **Pointers**
  - **Provide faster than indexing an arrays**
- **Macros**
  - **Eliminate overhead of function calls**

Optimization of C Programs

Operation	Rel. Strenght
`printf` and `scanf`	1000
`malloc` and `free`	800
trigonometric functions	500
floating point operations	100
integer division	30
integer multiplication	20
functional call	10
simple array index	6
shifts	5
integer add/subtract	5
pointer defererence	2
bitwise and or not	1
logical and or not	1

Table 15-1 (Practical C Programming)

Reference table for optimization by Reduction in strength

Compiler Optimization

- **`All modern compilers provide optimization features`**

- **`gcc`**, offers *3 levels of optimization* (**`1,2,3`**)
  - **`-O0`** No optimization (default, will **miss** some **warnings**)
  - **`-O or -O1`** Moderate optimization – finds more warnings
  - **`-O2`** Full optimization
  - **`-O3`** Full optimization + aggressive inlining + vectorization
  - **`-Os`** Space usage optimization

E.g: gcc -ansi mycode.c **-O2** **-o** mycode

Code Optimization

- Here we will discuss the simplest and most effective code optimization techniques

- This topic is discussed in further details in Chapter 3, sections 3.1 and 3.2 of G. Semeraro's Book (required reading)

Performance and Architecture

- To understand how to write programs that will execute quickly it is essential to understand the microprocessor architecture of the machine
  - (you will study this in detail in the **computer architecture** course)
- In modern microprocessors performance is all about **I**nstruction **L**evel **P**arallelism (**ILP**)

Instruction Level Parallelism

- **ILP** is a family of processor and compiler ***design techniques that speed up execution*** by causing individual machine operations to ***execute in parallel.***

- The ***greater the* ILP, *the higher the program execution rate***, *i.e.*, more instructions executed per "clock cycle"

Note: There is even a *Journal of Instruction-Level Parallelism* (www.jilp.org)

Increasing ILP

- Q: How do we increase the available **ILP** in our programs?
  - Using compiler optimization directives
  - ***Writing code to exploit the ILP***

- Within our programs there are basically two things that we can do
  - Write code that **interleaves many independent calculations** (“easy”)
  - Write code ***without long dependent operation chains*** (difficult, not pursued here)

Loop Unrolling

- ***Interleaving independent calculations*** can be easily done via *"loop unrolling"*

Loop unrolling:

- ***Replicate the code in the body of a loop*** multiple times and simultaneously reduce the number of iterations in the loop.

Benefit: Reduces loop overhead code while increasing the ILP.

Implementation: Compilers can do some loop unrolling, but usually programmers can do a much better job.

Example: Loop Unrolling

```
for (I = 0; I < 100; I++) {
   Sum [I] = VectorA [I] + VectorB [I];
} /* for (I) */
```

Analysis: Calculations at each iteration are:

- **Completely *independent* of all other iterations**
- **Could be *executed* by a different unit *in parallel***

- **We can increase ILP by *unrolling the loop***
  - **Modern compilers accomplish this to a certain degree**
  - **Programmers can in principle do better (since we know more about the *program and processor*)**

Example: Loop Unrolling

```
for (I = 0; I < 100; I += 4) {
   Sum [I+0] = VectorA [I+0] + VectorB [I+0];
   Sum [I+1] = VectorA [I+1] + VectorB [I+1];
   Sum [I+2] = VectorA [I+2] + VectorB [I+2];
   Sum [I+3] = VectorA [I+3] + VectorB [I+3];
} /* for (I) */
```

- **Loop unrolled four (4) times**
  - **Loop counter increment by four**
  - **Loop iterations *reduced by factor of 4***
  - **Loop code has *four times more lines***
- **General case**
  - **Can unroll *n* times (what is the optimal *n* ?)**

Loop Unrolling Analysis

- Q: Why does this increase ILP ?

  A: Because it ***interleaves (in this case) 4 independent calculations***

- Q: “How far should we go ?”, “How many iterations should we unroll ?”

  A: It depends on :

  - The ***type of calculations*** involved
  - The ***number of functional units*** in the processor, i.e., the # of calculations that the processor can perform in parallel

    (You will investigate this in the next homework)

Exercise: Loop Unrolling

- **Unroll the following loop (3) times**

```
for (i = 0; i < 41; i++) {
   z[i] = x[i]*y[i];
} /* for (i) */
```

- **Solution**

41-rem(41, 3)=39

```
for (i=0; i<39; i+=3) {
  z[i]=x[i]*y[i];
  z[i+1]=x[i+1]*y[i+1];
  z[i+2]=x[i+2]*y[i+2];
}
z[39]=x[39]*y[39];
z[40]=x[40]*y[40];
```

Homework Hints

- Each time you modify your code, be sure to re-run you tests multiple times (at least 5)
  - Caching WILL affect your results
  - Other system events WILL affect your results
- Be sure that you show your data for your optimization process.
  - Your analysis.txt should include detailed data for each step
  - Your overall conclusion must be supported by your data.

How to design functions

Mini Case Study:

Complex Arithmetic module

A Complex Numbers Module

- ***ANSI C89 does not* have standard libraries that *support complex numbers* (but C99 does)**
  - **Complex numbers arithmetic is often used in DSP applications (FFT algorithms)**

- ***C libraries* such *GSL* (GNU Scientific Library) and *FFTW* (Fastest FFT in the World) have *their own implementation* of complex numbers**

Here we will take a quick look at some details involved in designing data structures for complex numbers and the functions that support their arithmetic

Complex Representation 1/2

1. **`struct`** containing **`real`** and **`imaginary`**
   Multiplication & Division is time consuming

2. **`struct`** containing **`magnitude`** and **`angle`**
   ☹ Addition & Subtraction is time consuming

3. **`struct`** containing **`real`**, **`imaginary`**, and **`magnitude, angle`** components
   - ☺ All arithmetic performed in most natural way
   - ☹ Conversion to/from formats time consuming
   - ☹ it is difficult to know when conversion is necessary

Complex Representation 2/2

4. Array of 2 elements which can contain **real, imaginary** or **magnitude, angle**

☹ Programmer needs to keep track of what is stored in the array (same as using a **struct** with both formats or a **struct** of **unions** or **union** of **structs** for that matter)

5. Individual variables for the parts of each format

☹ Programs difficult to read, maintain and port.

Conclusion:

- **There is always a trade-off between competing objectives**
- **Design must achieve a compromise**

Complex Arithmetic

- ***Main issue:*** Complex functions must ***return more than a single value***
  - ☹ C does not support overloaded functions, nor does it support user-defined operators

- We need to decide how we will *define the interface* to the "complex math" functions.

- Assume that we use the "**`struct of doubles`**" to store ***complex numbers*** in rectangular form

```
typedef struct
  {
   double re, im;
  } dcomplex;
```

Structure Alternatives

```
/* 1. Take structs as input, returns a struct result */
dcomplex Cadd(dcomplex a, dcomplex b);


/* 2. Take structs as input, returns a pointer */
dcomplex* Cadd(dcomplex a, dcomplex b);


/* 3. Take structs as input and pointer to a
   struct to receive the result, returns an
   integer return code */
int Cadd( dcomplex* sum, dcomplex a, dcomplex b);
```

- What are the advantages and disadvantages of these choices?
- Reference for Complex Arithmetic
  http://en.wikipedia.org/wiki/Complex_number#Elementary_operations

Evaluation Criteria

1. Ease of implementation
2. Efficiency (how fast)
3. Chaining:
   e.g., `c*(a+b) -> cmul(c,cadd(a,b))`
4. Correctness
   1. Any “hidden” issues?

Complex Arithmetic, V1

```
/* 1. Take structs as input, returns the result *
 *    in a a struct */
dcomplex Cadd( dcomplex a, dcomplex b) {
/* Temporary variable for result */
   dcomplex sum;
/* Perform complex arithmetic */
   sum.re = a.re + b.re;
   sum.im = a.im + b.im;
/* return Result */
   return sum;
} /* Cadd() */
```

Complex Arithmetic, V1

```
dcomplex Cadd( dcomplex a, dcomplex b)
```

Analysis:

- Provides *correct* answers under all conditions
- Advantages

  (+) Straight-forward implementation

  - The functions take structs as inputs and return the result as a struct

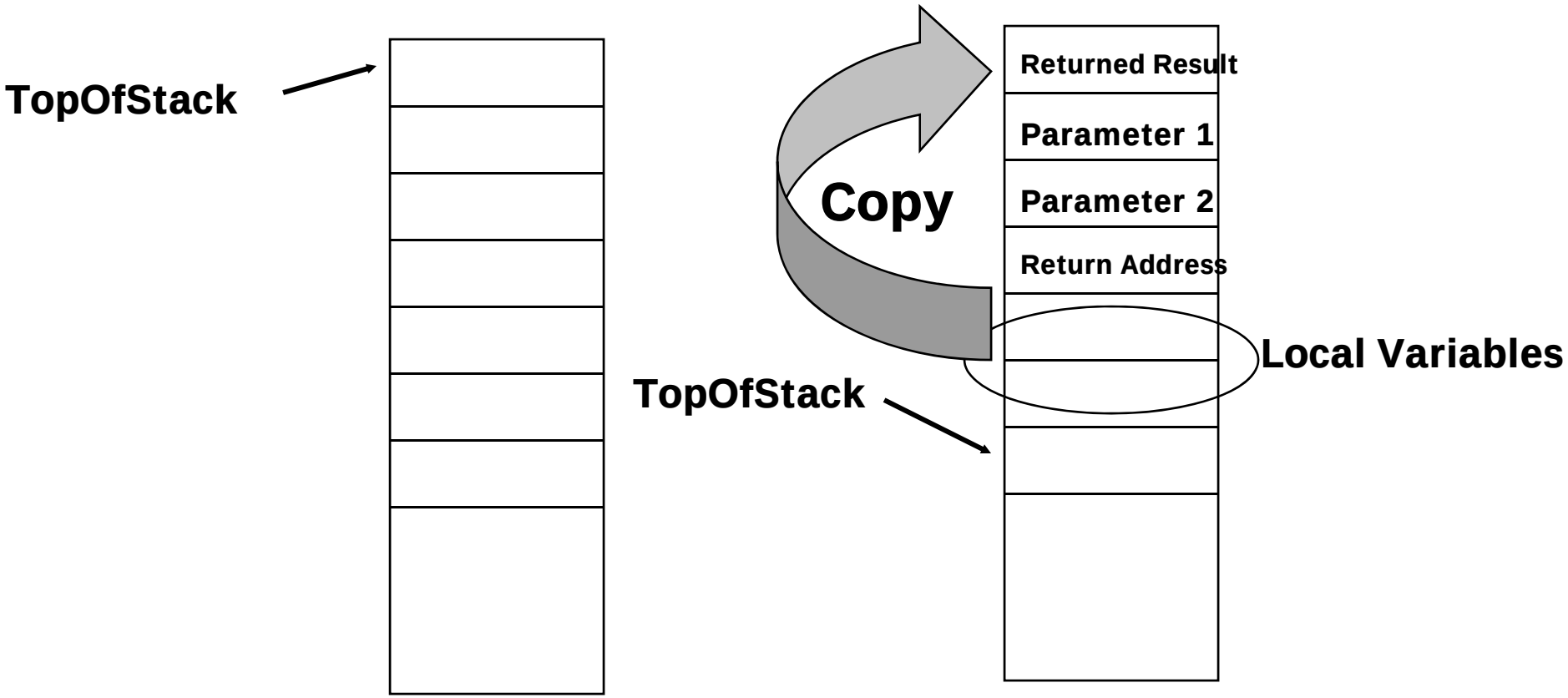
  (+) Allows "chaining" of operations (`d = a*b+c`)

```
d =  Cadd( Cmul(a,b),c );
```

- Disadvantages ?

  (-) "slow" since the *struct* that contains the *result must be copied* from the local variables section of the stack frame to the return variable section of the stack frame – see next slide

Stack Frame, V1



Complex Arithmetic, V2

```
/* 2. Take structs as input, returns a pointer to a   *
 * struct                                              */
   dcomplex* Cadd(dcomplex a, dcomplex b) {
      /* Temporary local automatic variable */
      dcomplex sum;
      /* Perform complex arithmetic */
      sum.re = a.re + b.re;
      sum.im = a.im + b.im;
      /* Return address of local
      return &sum;
     } /* Cadd() */
```

NBC / Via youtube.com

Complex Arithmetic, V2

```
dcomplex* Cadd(dcomplex a, dcomplex b)
```

Analysis:

- This method *does not* provide correct answers under all conditions !
- Advantages ?

  (+) Fast (no copying is needed)

  (+) *Appears* to allow "chaining" of operations, i.e., the following will usually produce the correct result

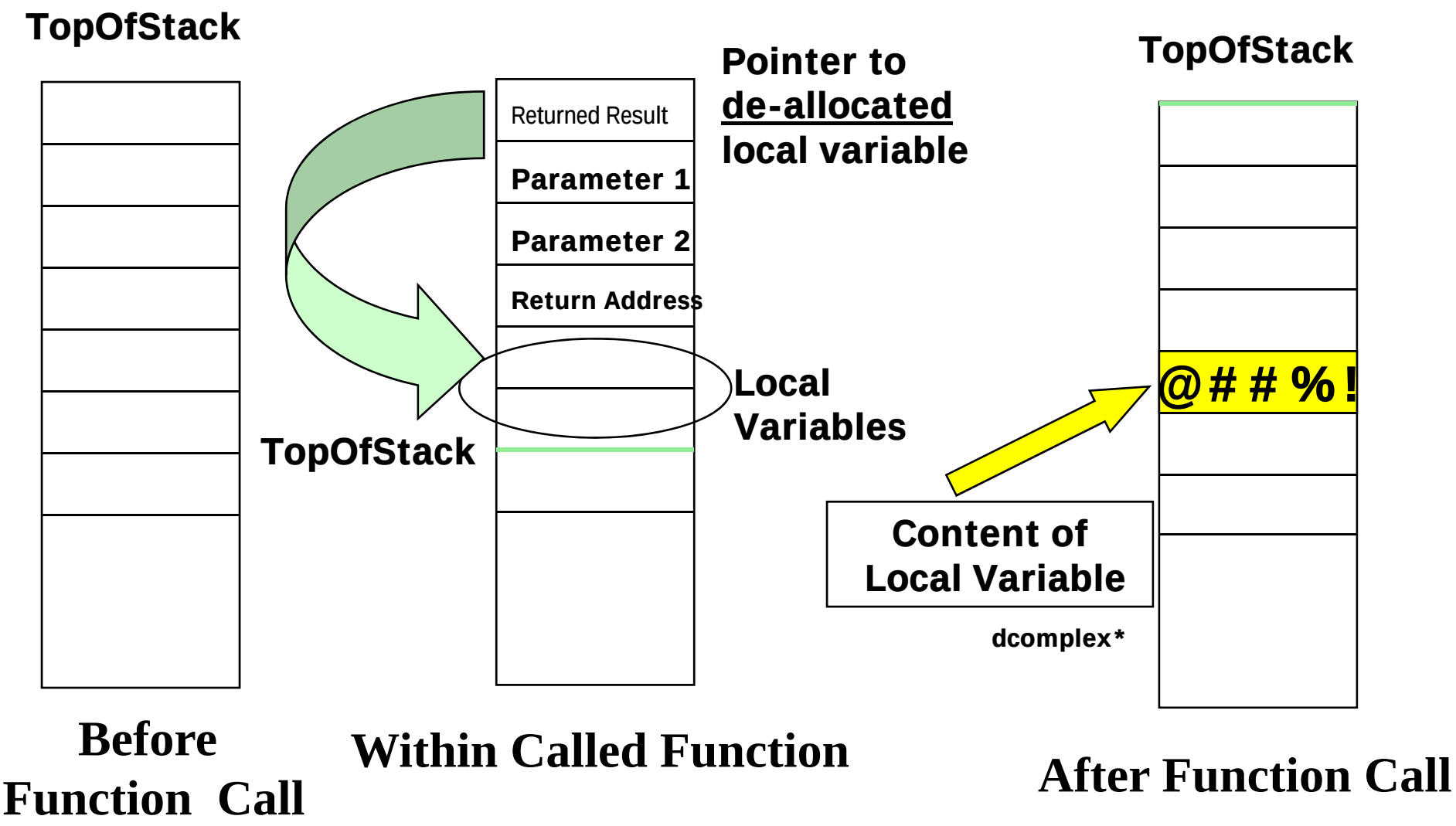
  ```
  d = ComplexAdd(*(ComplexMult(a, b)), c);
  ```

- Disadvantages ?

  (-) Does **not always** produce the **correct** result due to the way the stack is used (next side)

  (-) When chaining, copy is performed therefore it is not faster than Version 1

Stack Frame, V2



V2 Failure Mode

Analysis:

- Fails when:
  - We are executing multiple threads and the stack can be modified without our code making a function call

    (Recall that the variable that we are pointing to contains the correct result and as long as the memory doesn't get modified we will get the correct result when we de-reference the pointer)
  - We make a function call before the **last** use of the returned pointer
  - We call a function with variable number of arguments (such as **`printf()`** or **`scanf()`**) within or in the vicinity of the **`ComplexAdd()`** function

    (This fails because we cannot know how much of the stack will be modified and it can change from run to run of the program)
- Good News: **`gcc`** will *issue a warning*

  `warning: function returns address of local variable`
  - but most C compilers do not warn of this error !!

Complex Arithmetic, V3

```
/* 3. Take structs as input and address of a struct to
 *    return the result                                  */
   int Cadd(dcomplex* sum, dcomplex a,
                                   dcomplex b) {
   /* Perform complex arithmetic */
      (*sum).re = a.re + b.re;
      (*sum).im = a.im + b.im;

   /* Return Result */
      return 0;
   } /* Cadd() */
```

Complex Arithmetic, V3

- Analysis:
- The “best” solution (in terms of speed)

- Advantages ?

  (+) Straight-forward solution (although it uses pointers)
  - The functions take three structs and return the result in one of the parameters passed in

  (+) Fast since the function directly modifies the intermediate variable, i.e., the result is not copied

- Disadvantages ?

  (-) Does NOT allow “chaining” of operations

  There is no way to pass the result of one operation to another function without an intermediate variable

Summary: Complex Numbers

- Represent complex numbers using a **`struct`** containing its **`real`** and **`imaginary`** components.

```
typedef struct {
        double re, im;
} dcomplex;
```

- If speed not of primary concern, the best interface is to take **`structs`** as input and a **`struct`** to return the result

```
dcomplex Cadd(dcomplex a, dcomplex b);
```

- If *speed is of primary importance*, the best interface is to take structs as input and address of a struct to return the result

```
int Cadd( dcomplex* sum, dcomplex a, dcomplex b);
```

Complex Numbers in the GSL

- The **GNU Scientific Library** (GSL) represents complex numbers using the following:

```
typedef struct
{
double dat[2];
} gsl_complex;
```

- Real and imaginary parts are stored in contiguous elements of a ***two element array***.
  - This “eliminates padding between the real and imaginary parts”
  - Allows the `struct` to be mapped correctly onto packed complex arrays.

- Use `gsl_complex` to declare (**double precision**) GSL complex numbers

Complex Numbers in the GSL

- GSL provides functions and macros to manipulate complex numbers. (why macros ?)

- *Ex:* To *initialize* a complex GSL provides the functions

```
gsl_complex gsl_complex_rect (double x, double y)
gsl_complex gsl_complex_polar (double r, double theta)
```

and the **macros**

```
GSL_SET_COMPLEX (zp, x, y)
GSL_SET_REAL (zp,x)
GSL_SET_IMAG (zp,y)
```

Note: The argument ***zp*** denotes a pointer to a complex number

- Ex: To read the *real and imaginary part* of a complex number ***z*** GSL provides the **macros**

```
GSL_REAL (z)
GSL_IMAG (z)
```

Complex Numbers: ANSI C99

- **ANSI C99** *supports complex numbers* and basic complex arithmetic in **`complex.h`**

```
#include <complex.h>
 /* single precision */
complex z = 1.0 + 2.0*I;
 /* double precision */
double complex w = 1.0 + 2.0*I;
```

- **`I` is a macro** that defines the unit imaginary number

$$\mathrm{I} = i,\ i^2 = -1$$

Complex Numbers: ANSI C99

- Pointers and arrays of complex numbers are used in the same way as other primitive types

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
. . .
  int k;
  double complex b,c;
  /* dynamic array of complex numbers */
  double complex* a = malloc(10*sizeof(double complex))
  if (a == NULL){
    fprintf(stderr,"Memory allocation error");
    exit(EXIT_FAILURE);
  }
  /* Initialize array */
  for(k=0;k<10;k++){
    a[k] = (k+1) + (k+1)*I;
  }
  b = a[0]*a[9];
  c = a[1]/a[6]+a[3];
  printf("b = %5.2lf %+5.2lfI\n",creal(b),cimag(b));
. . .
```

Closing Remarks

- The ***ANSI C99*** implementation of **complex** is *clean and simple*, and leads to very readable programs.
  - Not all compilers support ANSI C99 (Microsoft does not)
  - Therefore the library **`<complex.h>`** may not be available

- The **GSL** (GNU Scientific Library) uses a **more traditional** (ANSI C89) way to implement complex numbers using structures.
  - It uses *macros for INLINING code*

- Other C libraries that require complex arithmetic (e.g., FFTW, **`www.fftw.org`**) offer their own implementation.

Code timing Problem 1

Will these macros properly measure code performance?
Why or why not?

```
START_TIMER(input);
BEGIN_REPEAT_TIMING(repeat_counter,10000);
 InputFile = fopen(argv[1], "r");
 do {
     if (fscanf(InputFile, "%lf %lf", &X, &Y) != EOF)
          {AddPoint(&DataSet, X, Y);  Done = 0; }
    else { Done = 1;   }
   } while (!Done);
END_REPEAT_TIMING;
STOP_TIMER(input);
fclose(InputFile);
```

Not a chance!
In pass 1 everything works as expected, BUT in pass 2 …N all the data has been read so NO MORE DATA IS READ giving a false measurement!
How do we fix this bug?

Code timing Problem 2

Will this properly measure code performance?

```
#include <stdio.h>
#include <time.h>
#define REPEAT 10000L
int main () {
  int        A, B, C, Index;
  clock_t  StartTime,  ElapsedTime;
  A=15; B=3;
  StartTime = clock();
       for (Index = 0; Index < REPEAT; Index++) { C=A*A/B; }
  ElapsedTime=  clock() - StartTime;
 printf ("   Total Time = %ld \"clock ticks\"\n", ElapsedTime);
 exit(0); }
```

Not a chance!
We are trying to measure a math operation with only 10,000 loops.
The answer is always ZERO!
How do we fix this bug?

REPEAT needs to be more like 10,000,000

Appendix

Complex Numbers: ANSI C99

- If you absolutely need (for whatever reason) ***to use*** **`I`** ***as a variable***,
  - ***Undefine*** **`I`** and **`complex`**
  - Use the alternative macros **`_Complex_I`** and **`_Complex`** as shown below

```
#include <complex.h>
#undef I
#undef complex
...
_Complex z = 1.0 + 2.0*_Complex_I;
...
```