1. Meaningful names for variables

Hungarian naming convention: use an abbreviation for the type as part of the variable name

Example:

```
char ch;       /* all char begins with ch */
byte b;        /* all byte begins with b */

char* pch;       /* pointer to char type variable */
char** ppch;    /*pointer to pointer to char type variable */

void *pvNewBlock(size_t size) ;  /* allocate a memory of size "size" and return void type
                                    pointer of the starting address of the allocated memory */
```

2. Put Assert in functions

```
/* memcpy -- copy a nonoverlapping memory block */
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
   void* pbTo = (byte*)pvTo;
  void* pbFrom = (byte*)pvFrom;

   if(pvTo == NULL | | pvFrom == NULL)
   {
      fprintf(stderr, "Bad args in memcpy\n");
      abort();
   }

   while(size-->0)
       *pbTo++ == *pbFrom++;
   return(pvTo);
}
```

2.1. Debugging version vs. final version

```
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
   void* pbTo = (byte*)pvTo;
  void* pbFrom = (byte*)pvFrom;

  #ifdef   DEBUG
   if(pvTo == NULL | | pvFrom == NULL)
   {
      fprintf(stderr, “Bad args in memcpy\n”);
      abort();
   }
   #endif /*  #ifdef   DEBUG  */

    while(size-->0)
        *pbTo++ == *pbFrom++;
   return(pvTo);
}
```

2.2. Using Assert

An assertion specifies that a program satisfies certain conditions at particular points in its execution. In C, assertions are implemented with the standard *assert* macro. The argument to *assert* must be true when the macro is executed, otherwise the program aborts and prints an error message.

Example:

```
  assert( size <= LIMIT );
```

will abort the program and print an error message like this:

```
   Assertion violation: file test.c, line 34: size <= LIMIT
```

if size is greater than LIMIT.

```
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
  void* pbTo = (byte*)pvTo;
```

```
    void* pbFrom = (byte*)pvFrom;
    assert(pvTo != NULL && pvFrom != NULL);
    while(size-->0)
        *pbTo++ == *pbFrom++;
    return(pvTo);
}
```

assert is a debug-only macro that aborts execution if its argument is false. This macro is disabled if, at the moment of including <assert.h>, a macro with the name NDEBUG has already been defined. This allows for a coder to include as many assert calls as needed in a source code while debugging the program and then disable all of them for the production version by simply including a line like:

```
#define NDEBUG
```

at the beginning of its code, before the inclusion of <assert.h>.

3. Improving subsystems

When finishing writing a subsystem, ask yourself, "How are programmers going to misuse this subsystem, and how can I detect these problems automatically?".

3.1. Put scaffolding around the C routines in the form of cover functions

```
/* fNewMemory --   allocate a memory block.*/

int fNewMemory(void** pv, size_t size)
{
    byte** ppb = (byte**)ppv;
    *ppb = (byte*)malloc(size);
    return(*ppb != NULL);
}
```

3.2. Introducing mechanisms to catch exception, report error

```
if (fNewMemory(&pbBlock, 32))
    successful -- pbBlock  points to the block
else
    unsuccessful -- pbBlock  is NULL
```

4. Coding with Style

4.1. Documenting the code

Writing comments to explain how clients should interact with the code.

4.2. Choosing meaningful names for variables and functions

4.3. Using Language Features with Style

Using Language Features with Style

Use Constants: The language offers constants to give a symbolic name to a value that doesn't change.

Example:

const int kAveragePriceOfCheeseInNewBrunswick = 24;

Take Advantage of const Variables: when we do not want to change value of the variable

Example:

void wontChangeString(const char* inString);

// tells the caller that it will not change the content of the C-style string that is passed in.

4.4. Formatting

The Curly Brace Alignment and Indentation

```
void someFunction()
{
   if (condition())
   {
      cout << "condition was true" << endl;
   }
   else
   {
      cout << "condition was false" << endl;
   }
}
```

5. C struct

A struct is an aggregate of elements of arbitrary type.
Example:

```
struct address {
    int      number;
    char    *street;
    char    state[2];
    int      zip;
};
```

defines a new type called address. Individual members of the variable of type address can be accessed using the .(dot) operator.
Example:

```
address jd;
jd.number = 61;
jd.street = "South St.";
```

Structure objects can also be accessed through pointers using the -> operator.
Example:

```
address jd, *pjd;
pjd = &jd;
 pjd->number  = 60;

void print_addr(address *paddr)
{
    cout<<"street name "<< paddr->street <<endl;
}
```

6. Preprocessing and Macros

6.1.Macro definition and expansion

A preprocessing directive of the form

#define *identifier token-string*

cause the preprocessor to replace subsequent instances of *identifier* with the given sequence of *tokens*.

Example:

```
#define SIDE 8
```

the declaration

```
char chessboard[SIDE][SIDE];
```

after macro expansion becomes

```
char chessboard[8][8];
```

"Function like" macro definition

#define identifier(identifier, .., identifier) token-string

Example:

```
#define index_mask 0XFF00
#define extract(word,mask) word & mask
```

the call

```
index = extract(packed_data, index_mask);
```

expands to

```
index = packed_data&0XFF00;
```

6.2 Advantages of using macro and preprocessing directive

- Reduce length of the code
- Saves time spent by the compiler for calling functions (a speed-up in performance)
- Make code easy to maintain

7. Tricks for writing efficient code

7.1.Dead Code Removal

Before	After

var = 5;
printf("%d", var);
exit(0);
printf("%", var*2);
	var = 5;
	printf("%d", var);
	exit(0);

7.2 Using cheap arithmetic

Raising one value to the power of another, or dividing, is more expensive than multiplying.

Before	After
x = pow(y,2.0); a=c/2.0;	x = y*y; a=c*0.5;

7.3 Eliminating common subexpression

Before	After

d = c*(a/b);
e= (a/b)*2.0;
	adivb = a/b;
	d = c*adivb;
	e= adivb*2.0;

The subexpression (a/b) occurs in both assignment statements. There is no need to calculate it twice.

7.4 Renaming variables

Before	After

x=y*z;
q=r+x*2.0;
x=a+b;
	x0=y*z;
	q=r+x0*2.0;
	x=a+b;;

The original code has an output dependency, while the new code doesn't.

7.5 Avoid global variables

// if possible, do not use variables like

```
extern int a;
```

7.6 Use qualifiers

```
const //Define constants
const int N = 10;
int t[N];
N = 20; // compilation error. Const can not be modified.

//Protecting a parameter
void sayHello(const string& who){
    cout<<"Hello, "+who<<endl;
    who = "new name";  // compilation error
}
```

7.7 Use **NULL** for empty code block

```
if(5 == x)
{
     NULL;
}
```

7.8 Variable Length Array(VLA)

DO NOT:

```
void foo(int n) {
    int values[n]; //Declare a variable length array
}
```

This can cause memory issue when `n` is large.

7.9 Copy and Move

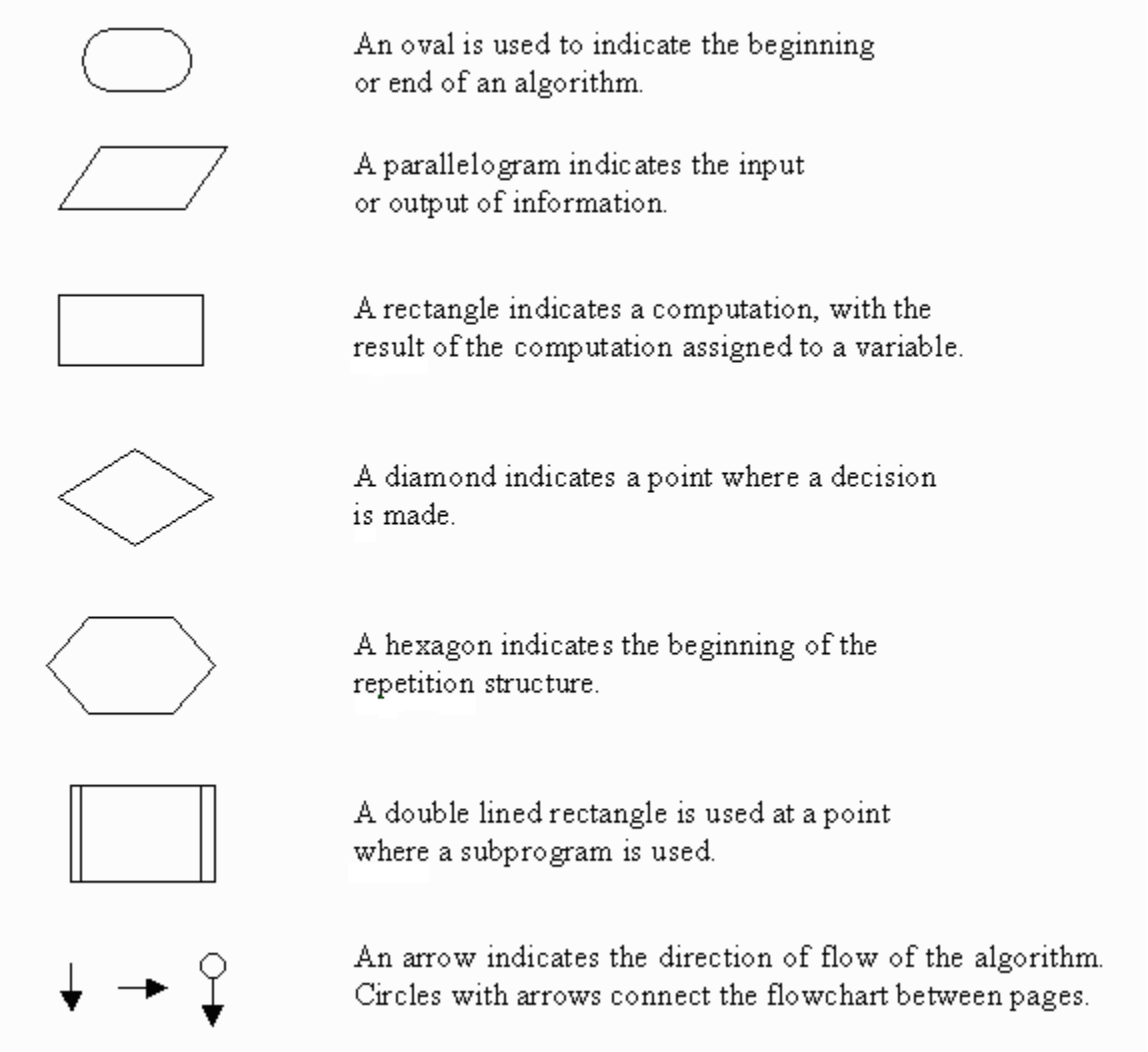
```
#define sz 100
double elem[sz], a[sz];
…
for(int I = 0; I < sz; I++)
   elem[I] = a.elem[I];
```

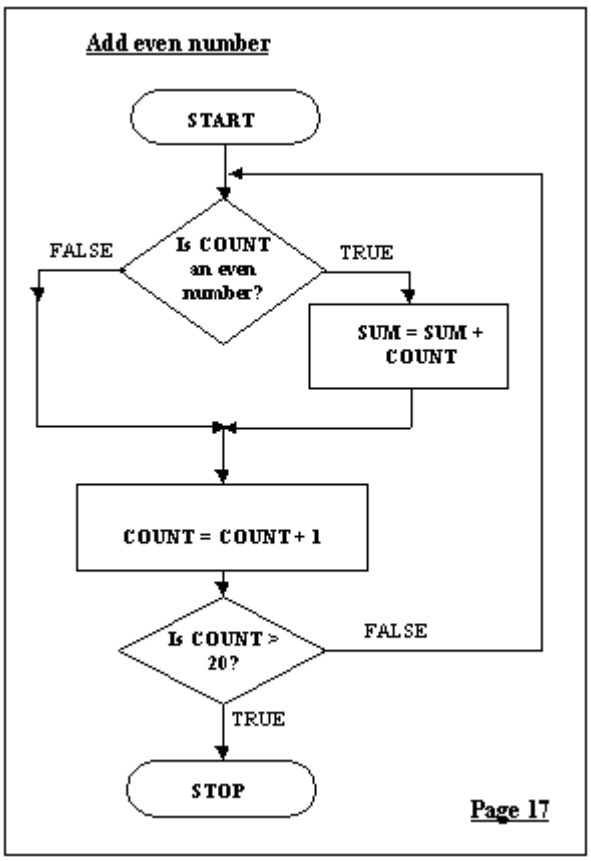
Above implementation is NOT efficient. Use `memcpy()` or `std::copy()` for fast memory copying.

When copy is expensive (long vector, large C++ object, e.g.) and the original data is no longer needed, use C++ move semantics. See Section 17.5.2 of "The C++ programming language".

8. Program Flow Chart

http://www.programiz.com/article/flowchart-programming





Reference:

1. Steve Maguire, Writing Solid Code
2. Bruce Eckel, Thinking in C++
3. Bjarne Stroustrup, The C+ + Programming Language
4. Nicholas A. Solter, and Scott J. Kleper, Professional C++