



## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

# Module 06: Programming in C++

## Constants and Inline Functions

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

## Module 06

Partha Pratim  
Das

### Objectives & Outline

`const-ness` &  
`cv-qualifier`

`const-ness`  
Advantages  
Pointers  
`volatile`

`inline`  
functions

Macros  
`inline`

Summary

- Understand `const` in C++ and contrast with Manifest Constants
- Understand `inline` in C++ and contrast with Macros



# Module Outline

## Module 06

Partha Pratim  
Das

## Objectives & Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- const-ness and cv-qualifier
  - Notion of const
  - Advantages of const
    - Natural Constants –  $\pi$ ,  $e$
    - Program Constants – array size
    - Prefer const to #define
  - const and pointer
    - const-ness of pointer / pointee. How to decide?
  - Notion of volatile
- inline functions
  - Macros with params
    - Advantages
    - Disadvantages
  - Notion of inline functions
    - Advantages



# Module 06: Lecture 08

## Module 06

Partha Pratim  
Das

## Objectives & Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- const-ness and cv-qualifier
  - Notion of const
  - Advantages of const
    - Natural Constants –  $\pi$ ,  $e$
    - Program Constants – array size
    - Prefer const to #define
  - const and pointer
    - const-ness of pointer / pointee. How to decide?



# Program 06.01: Manifest constants in C

## Module 06

Partha Pratim Das

Objectives & Outline

const-ness & cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- Manifest constants are defined by `#define`
- Manifest constants are replaced by CPP (C Pre-Processor)

Source Program	Program after CPP
<pre>#include &lt;iostream&gt; #include &lt;cmath&gt; using namespace std;  #define TWO 2 #define PI 4.0*atan(1.0)  int main() {     int r = 10;     double peri =         TWO * PI * r;     cout &lt;&lt; "Perimeter = "         &lt;&lt; peri &lt;&lt; endl;     return 0; }</pre>	<pre>// Contents of &lt;iostream&gt; header replaced by CPP // Contents of &lt;cmath&gt; header replaced by CPP using namespace std;  // #define of TWO consumed by CPP // #define of PI consumed by CPP  int main() {     int r = 10;     double peri =         2 * 4.0*atan(1.0) * r; // Replaced by CPP     cout &lt;&lt; "Perimeter = "         &lt;&lt; peri &lt;&lt; endl;     return 0; }</pre>
Perimeter = 314.159	Perimeter = 314.159
<ul style="list-style-type: none"> <li>● TWO is a manifest constant</li> <li>● PI is a manifest constant</li> <li>● TWO &amp; PI look like variables</li> </ul>	<ul style="list-style-type: none"> <li>● CPP replaces the token TWO by 2</li> <li>● CPP replaces the token PI by 4.0*atan(1.0)</li> <li>● Compiler sees them as constants</li> </ul>



# Notion of const-ness

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- The value of a const variable cannot be changed after definition

```
const int n = 10; // n is an int type variable with value 10
                // n is a constant

...
n = 5; // Is a compilation error as n cannot be changed

...
int m;
int *p = 0;
p = &m; // Hold m by pointer p
*p = 7; // Change m by p; m is now 7
...
p = &n; // Is a compilation error as n may be changed by *p = 5;
```

- Naturally, a const variable *must be initialized when defined*

```
const int n; // Is a compilation error as n must be initialized
```

- A variable of any data type can be declared as const

```
typedef struct _Complex {
    double re;
    double im;
} Complex;
const Complex c = {2.3, 7.5}; // c is a Complex type variable
                          // It is initialized with c.re = 2.3 and c.im = 7.5
                          // c is a constant

...
c.re = 3.5; // Is a compilation error as no part of c can be changed
```



# Program 06.02: Compare #define and const

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

Using #define	Using const
<pre>#include &lt;iostream&gt; #include &lt;cmath&gt; using namespace std;  #define TWO 2 #define PI 4.0*atan(1.0)  int main() {     int r = 10;     double peri =         TWO * PI * r;     cout &lt;&lt; "Perimeter = "         &lt;&lt; peri &lt;&lt; endl;     return 0; }</pre>	<pre>#include &lt;iostream&gt; #include &lt;cmath&gt; using namespace std;  const int TWO = 2; const double PI = 4.0*atan(1.0);  int main() {     int r = 10;     double peri =         TWO * PI * r; // No replacement by CPP     cout &lt;&lt; "Perimeter = "         &lt;&lt; peri &lt;&lt; endl;     return 0; }</pre>
Perimeter = 314.159	Perimeter = 314.159
<ul style="list-style-type: none"><li>• TWO is a manifest constant</li><li>• PI is a manifest constant</li><li>• TWO &amp; PI look like variables</li><li>• Types of TWO &amp; PI may be indeterminate</li></ul>	<ul style="list-style-type: none"><li>• TWO is a const variable initialized to 2</li><li>• PI is a const variable initialized to 4.0*atan(1.0)</li><li>• TWO &amp; PI are variables</li><li>• Type of TWO is const int</li><li>• Type of PI is const double</li></ul>



# Advantages of const

- Natural Constants like  $\pi$ ,  $e$ ,  $\Phi$  (*Golden Ratio*) etc. can be compactly defined and used

```
const double pi = 4.0*atan(1.0);           // pi = 3.14159
const double e = exp(1.0);                 // e = 2.71828
const double phi = (sqrt(5.0) + 1) / 2.0;  // phi = 1.61803
```

```
const int TRUE = 1;                        // Truth values
const int FALSE = 0;
```

```
const int null = 0;                        // null value
```

Note: NULL is a manifest constant in C/C++ set to 0.

- Program Constants like number of elements, array size etc. can be defined at one place (at times in a header) and used all over the program

```
const int nArraySize = 100;
const int nElements = 10;

int main() {
    int A[nArraySize];           // Array size
    for (int i = 0; i < nElements; ++i) // Number of elements
        A[i] = i * i;

    return 0;
}
```





# Advantages of const

## • Prefer const over #define

### Using #define

#### Manifest Constant

- Is not type safe
- Replaced textually by CPP
- Cannot be *watched* in debugger
- Evaluated as many times as replaced

### Using const

#### Constant Variable

- Has its type
- Visible to the compiler
- Can be *watched* in debugger
- Evaluated only on initialization

Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages

Pointers  
volatile

inline  
functions

Macros  
inline

Summary



# const and Pointers

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages

Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- const-ness can be used with Pointers in one of the two ways:
  - **Pointer to Constant data** where the pointee (pointed data) cannot be changed
  - **Constant Pointer** where the pointer (address) cannot be changed
- Consider usual pointer-pointee computation (without const):

```
int m = 4;
int n = 5;
int * p = &n; // p points to n. *p is 5
...
n = 6;        // n and *p are 6 now
*p = 7;       // n and *p are 7 now. POINTEE changes
...
p = &m;       // p points to m. *p is 4. POINTER changes
*p = 8;       // m and *p are 8 now. n is 7. POINTEE changes
```



# const and Pointers: *Pointer to Constant data*

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

## Consider pointed data

```
int m = 4;
const int n = 5;
const int * p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Okay
```

## Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6; // Okay
*p = 6; // Error: p points to a 'constant' data (n) that cannot be changed
```

## Finally,

```
const int n = 5;
int * p = &n; // Error: If this were allowed, we would be able to change constant n
...
n = 6; // Error: n is constant and cannot be changed
*p = 6; // Would have been okay, if declaration of p were valid
```



# const and Pointers: *Constant Pointer*

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages

Pointers  
volatile

inline  
functions

Macros  
inline

Summary

## Consider pointer

```
int m = 4, n = 5;
int * const p = &n;
...
n = 6; // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

## By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a 'constant' data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through \*

```
int n = 5;
int * p = &n;           // non-const-Pointer to non-const-Pointee
const int * p = &n;      // non-const-Pointer to const-Pointee
int * const p = &n;      // const-Pointer to non-const-Pointee
const int * const p = &n; // const-Pointer to const-Pointee
```



# const and Pointers: The case of C-string

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

Consider the example:

```
char * str = strdup("IIT, Kharagpur");  
str[0] = 'N';           // Edit the name  
cout << str << endl;  
str = strdup("JIT, Kharagpur"); // Change the name  
cout << str << endl;
```

Output is:

NIT, Kharagpur  
JIT, Kharagpur

To stop editing the name:

```
const char * str = strdup("IIT, Kharagpur");  
str[0] = 'N';           // Error: Cannot Edit the name  
str = strdup("JIT, Kharagpur"); // Change the name
```

To stop changing the name:

```
char * const str = strdup("IIT, Kharagpur");  
str[0] = 'N';           // Edit the name  
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

To stop both:

```
const char * const str = strdup("IIT, Kharagpur");  
str[0] = 'N';           // Error: Cannot Edit the name  
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```



# Module 06: End of Lecture 08

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- const-ness and cv-qualifier
  - Notion of const
  - Advantages of const
    - Natural Constants –  $\pi$ ,  $e$
    - Program Constants – array size
    - Prefer const to #define
  - const and pointer
    - const-ness of pointer / pointee. How to decide?



# Module 06: Lecture 09

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages

Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- Notion of Volatile
- Inline Function
  - Macros with parameters
    - Advantage
    - Disadvantage
  - Notion of inline
    - Advantage



# Notion of volatile

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
**volatile**

inline  
functions

Macros  
inline

Summary

- Variable Read-Write
  - The value of a variable can be read and / or assigned at any point of time
  - The value assigned to a variable does not change till a next assignment is made (value is persistent)
- **const**
  - The value of a **const** variable can be set only at initialization – cannot be changed afterwards
- **volatile**
  - *In contrast*, the value of a **volatile** variable may be different every time it is read – *even if no assignment has been made to it*
  - A variable is taken as **volatile** if it can be changed by hardware, the kernel, another thread etc.
- **cv-qualifier**: A declaration may be prefixed with a qualifier – **const** or **volatile**





# Using volatile

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
**volatile**

inline  
functions

Macros  
inline

Summary

Consider:

```
static int i;  
void fun(void) {  
    i = 0;  
    while (i != 100);  
}
```

This is an infinite loop! Hence the compiler should optimize as:

```
static int i;  
void fun(void) {  
    i = 0;  
    while (1);           // Compiler optimizes  
}
```

Now qualify `i` as `volatile`:

```
static volatile int i;  
void fun(void) {  
    i = 0;  
    while (i != 100);    // Compiler does not optimize  
}
```

Being `volatile`, `i` can be changed by hardware anytime. It waits till the value becomes 100 (possibly some hardware writes to a port).



# Program 06.03: Macros with Parameters

## Module 06

Partha Pratim Das

Objectives & Outline

const-ness & cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

- Macros with Parameters are defined by `#define`
- Macros with Parameters are replaced by CPP

Source Program	Program after CPP
<pre>#include &lt;iostream&gt; using namespace std;  #define SQUARE(x) x * x  int main() {     int a = 3, b;      b = SQUARE(a);      cout &lt;&lt; "Square = "         &lt;&lt; b &lt;&lt; endl;     return 0; }</pre>	<pre>// Contents of &lt;iostream&gt; header replaced by CPP using namespace std;  // #define of SQUARE(x) consumed by CPP  int main() {     int a = 3, b;      b = a * a; // Replaced by CPP      cout &lt;&lt; "Square = "         &lt;&lt; b &lt;&lt; endl;     return 0; }</pre>
Square = 9	Square = 9
<ul style="list-style-type: none"><li>• <code>SQUARE(x)</code> is a macro with one param</li><li>• <code>SQUARE(x)</code> looks like a function</li></ul>	<ul style="list-style-type: none"><li>• CPP replaces the <code>SQUARE(x)</code> substituting <code>x</code> with <code>a</code></li><li>• Compiler does not see it as function</li></ul>



# Pitfalls of macros

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

Consider the example:

```
#include <iostream>
using namespace std;

#define SQUARE(x) x * x

int main() {
    int a = 3, b;

    b = SQUARE(a + 1); // Wrong macro expansion

    cout << "Square = " << b << endl;

    return 0;
}
```

Output is 7 in stead of 16 as expected. On the expansion line it gets:

```
b = a + 1 * a + 1;
```

To fix:

```
#define SQUARE(x) (x) * (x)
```

Now:

```
b = (a + 1) * (a + 1);
```



# Pitfalls of macros

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

## Continuing ...

```
#include <iostream>
using namespace std;

#define SQUARE(x) (x) * (x)

int main() {
    int a = 3, b;

    b = SQUARE(++a);

    cout << "Square = " << b << endl;

    return 0;
}
```

Output is 25 in stead of 16 as expected. On the expansion line it gets:

```
b = (++a) * (++a);
```

and a is incremented twice before being used! There is no easy fix.



# inline Function

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

`const-ness` &  
`cv-qualifier`

`const-ness`  
Advantages  
Pointers  
`volatile`

`inline`  
functions

Macros  
`inline`

Summary

- An `inline` function is just another functions
- The function prototype is preceded by the keyword `inline`
- An `inline` function is expanded (inlined) at the site of its call and the overhead of passing parameters between caller and callee (or called) functions is avoided



# Program 06.04: Macros as inline Functions

## Module 06

Partha Pratim Das

Objectives & Outline

const-ness & cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline functions

Macros  
inline

Summary

- Define the function
- Prefix function header with `inline`
- Compile function body and function call together

Using macro	Using inline
<pre>#include &lt;iostream&gt; using namespace std; #define SQUARE(x) x * x int main() {     int a = 3, b;     b = SQUARE(a);     cout &lt;&lt; "Square = "          &lt;&lt; b &lt;&lt; endl;     return 0; }</pre>	<pre>#include &lt;iostream&gt; using namespace std; inline int SQUARE(int x) { return x * x; } int main() {     int a = 3, b;     b = SQUARE(a);     cout &lt;&lt; "Square = "          &lt;&lt; b &lt;&lt; endl;     return 0; }</pre>
Square = 9	Square = 9
<ul style="list-style-type: none"><li>● <code>SQUARE(x)</code> is a macro with one param</li><li>● Macro <code>SQUARE(x)</code> is efficient</li><li>● <code>SQUARE(a + 1)</code> fails</li><li>● <code>SQUARE(++a)</code> fails</li><li>● <code>SQUARE(++a)</code> does not check type</li></ul>	<ul style="list-style-type: none"><li>● <code>SQUARE(x)</code> is a function with one param</li><li>● <code>inline SQUARE(x)</code> is equally efficient</li><li>● <code>SQUARE(a + 1)</code> works</li><li>● <code>SQUARE(++a)</code> works</li><li>● <code>SQUARE(++a)</code> checks type</li></ul>



# Macros & inline Functions: Compare and Contrast

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

const-ness &  
cv-qualifier

const-ness  
Advantages  
Pointers  
volatile

inline  
functions

Macros  
inline

Summary

### Macros

- Expanded at the place of calls
- Efficient in execution
- Code bloats
- Has syntactic and semantic pitfalls
- Type checking for parameters is not done
- Helps to write `max` / `swap` for all types
- Errors are not checked during compilation
- Not available to debugger

### inline Functions

- Expanded at the place of calls
- Efficient in execution
- Code bloats
- No pitfall
- Type checking for parameters is robust
- Needs `template` for the same purpose
- Errors are checked during compilation
- Available to debugger in `DEBUG` build



# Limitations of Function `inline`ing

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

`const`-ness &  
`cv`-qualifier

`const`-ness  
Advantages  
Pointers  
`volatile`

`inline`  
functions

Macros  
`inline`

Summary

- `inline`ing is a directive – compiler may not inline functions with large body
- `inline` functions may not be recursive
- Function body is needed for `inline`ing at the time of function call. Hence, implementation hiding is not possible. *Implement inline functions in header files*
- `inline` functions must not have two different definitions





# Module Summary

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

`const-ness &  
cv-qualifier`

`const-ness  
Advantages  
Pointers  
volatile`

`inline  
functions`

`Macros  
inline`

Summary

- Revisit manifest constants from C
- Understand `const`-ness, its use and advantages over manifest constants
- Understand the interplay of `const` and pointer
- Understand the notion and use of `volatile` data
- Revisit macros with parameters from C
- Understand `inline` functions and their advantages over macros
- Limitations of `inlineing`



# Instructor and TAs

## Module 06

Partha Pratim  
Das

Objectives &  
Outline

`const-ness` &  
`cv-qualifier`

`const-ness`  
Advantages  
Pointers  
`volatile`

`inline`  
functions

Macros  
`inline`

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655