Module 06: Programming in Modern C++

Constants and Inline Functions

Intructors: Abir Das and Sourangshu Bhattacharya

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

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in C++

by **Prof. Partha Pratim Das**

Module Objectives

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

Module Outline

- const-ness and cv-qualifier
  - Notion of const
  - Advantages of const
    - Natural Constants – π, 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

Program 06.01: Manifest constants in C

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

Source Program

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

#define TWO 2             // Manifest const
#define PI 4.0*atan(1.0) // Const expr.

int main() { int r = 10;
    double peri = TWO * PI * r;
    cout << "Perimeter = " << peri << endl;
}
```

Perimeter = 62.8319

- `TWO` is a manifest constant
- `PI` is a manifest constant as macro
- `TWO` & `PI` look like variables

Program after CPP

```
// Contents of <iostream> header replaced by CPP
// Contents of <cmath> 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; // By CPP
    cout << "Perimeter = " << peri << endl;
}
```

Perimeter = 62.8319

- CPP replaces the token `TWO` by `2`
- CPP replaces the token `PI` by `4.0*atan(1.0)` and evaluates
- Compiler *sees* them as constants
- `TWO * PI` = `6.28319` by constant folding of compiler

Notion of const-ness

- 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

Using `#define`

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

#define TWO 2
#define PI 4.0*atan(1.0)

int main() { int r = 10;
    // Replace by CPP
    double peri = 2 * 4.0*atan(1.0) * r;
    cout << "Perimeter = " << peri << endl;
}
```

Perimeter = 62.8319

- `TWO` is a manifest constant
- `PI` is a manifest constant
- `TWO` & `PI` *look like variables*
- Types of `TWO` & `PI` may be indeterminate
- `TWO` * `PI` = `6.28319` by constant folding of compiler

Using `const`

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

const int TWO = 2;
const double PI = 4.0*atan(1.0);

int main() { int r = 10;
    // No replacement by CPP
    double peri = TWO * PI * r;
    cout << "Perimeter = " << peri << endl;
}
```

Perimeter = 62.8319

- `TWO` is a `const` variable initialized to `2`
- `PI` is a `const` variable initialized to `4.0*atan(1.0)`
- `TWO` & `PI` *are variables*
- Type of `TWO` is `const int`
- Type of `PI` is `const double`

Advantages of const

- Natural Constants like π, e, Φ (*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;
}
```

Advantages of const

- Prefer `const` over `#define`

Using `#define`	**Using `const`**
Manifest Constant	**Constant Variable**
• Is not type safe • Replaced textually by CPP • Cannot be *watched* in debugger • Evaluated as *many times as replaced*	• Has its type • Visible to the compiler • Can be *watched* in debugger • Evaluated *only on initialization*

const and Pointers

- `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*

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; // Error: p points to a constant data. Its pointee cannot be changed
```

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: Example

What will be the output of the following program:

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

int main() {
const int a = 5;
int *b;
b = (int *) &a;
*b = 10;
cout << a << " " <<b<<" "<< &a <<" "<< *b <<"\n";
}
```

const and Pointers: Example

What will be the output of the following program:

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

int main() {
const int a = 5;
int *b;
b = (int *) &a;
*b = 10;
cout << a << " " <<b<<" "<< &a <<" "<< *b <<"\n";
}
```

Standard g++ compiler prints: `5 0x16b58f4ec 0x16b58f4ec 10`
b actually points to a
But when accessed through a the compiler substitutes the constant expression Technically the behavior is **undefined**

const and Pointers: *Constant Pointer*

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

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
```

Notion of `volatile`

- 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
- `const`
  - A `const` variable's value is set *only at initialization* – *can't be changed* afterwards
- `volatile`
  - In contrast, the value of a `volatile` variable can be modified by actions other than those in the user application.
  - Therefore, the volatile keyword is useful for declaring variables in shared memory that can be accessed by multiple processes for communication with interrupt service routines. It can be *changed by hardware, the kernel, another thread* etc.
  - When a name is declared as volatile, the compiler reloads the value from memory each time it is accessed by the program. This dramatically reduces the possible optimizations.
- `cv-qualifier`: A declaration may be prefixed with a qualifier – `const` or `volatile`

Using volatile

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

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

Source Program

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

#define SQUARE(x) x * x

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

    b = SQUARE(a);

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

Square = 9

- `SQUARE(x)` is a macro with one param
- `SQUARE(x)` looks like a function

Program after CPP

```
#include <iostream> // 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 << "Square = " << b << endl;
}
```

Square = 9

- CPP replaces the `SQUARE(x)` substituting `x` with `a`
- Compiler does not *see* it as function

Pitfalls of macros

Consider the example:

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

#define SQUARE(x) x * x

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

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

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

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

Continuing ...

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

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

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

    b = SQUARE(++a);

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

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

- An `inline` function is just a function like any other
- 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

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

Using macro

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main() {
    int a = 3, b;
    b = SQUARE(a);
    cout << "Square = " << b << endl;
}
```

Square = 9

- `SQUARE(x)` is a macro with one param
- Macro `SQUARE(x)` is efficient
- `SQUARE(a + 1)` fails
- `SQUARE(++a)` fails
- `SQUARE(++a)` does not check type

Using `inline`

```
#include <iostream>
using namespace std;
inline int SQUARE(int x) { return x * x; }
int main() {
    int a = 3, b;
    b = SQUARE(a);
    cout << "Square = " << b << endl;
}
```

Square = 9

- `SQUARE(x)` is a function with one param
- `inline SQUARE(x)` is equally efficient
- `SQUARE(a + 1)` works
- `SQUARE(++a)` works
- `SQUARE(++a)` checks type

Macros & `inline` Functions: Compare and Contrast

Macros	**`inline` Functions**
• Expanded at the place of calls	• Expanded at the place of calls
• Efficient in execution	• Efficient in execution
• Code bloats	• Code bloats
• Has *syntactic and semantic pitfalls*	• *No pitfall*
• *Type checking* for parameters is *not done*	• *Type checking* for parameters is *robust*
• Errors are *not checked during compilation*	• Errors are *checked during compilation*
• *Not available* to debugger	• *Available* to debugger in DEBUG build

Limitations of Function `inlineing`

- `inlineing` is a *directive* – compiler may not inline functions with large body
- `inline` functions may not be *recursive*
- Function body is needed for `inlineing` 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

- 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`