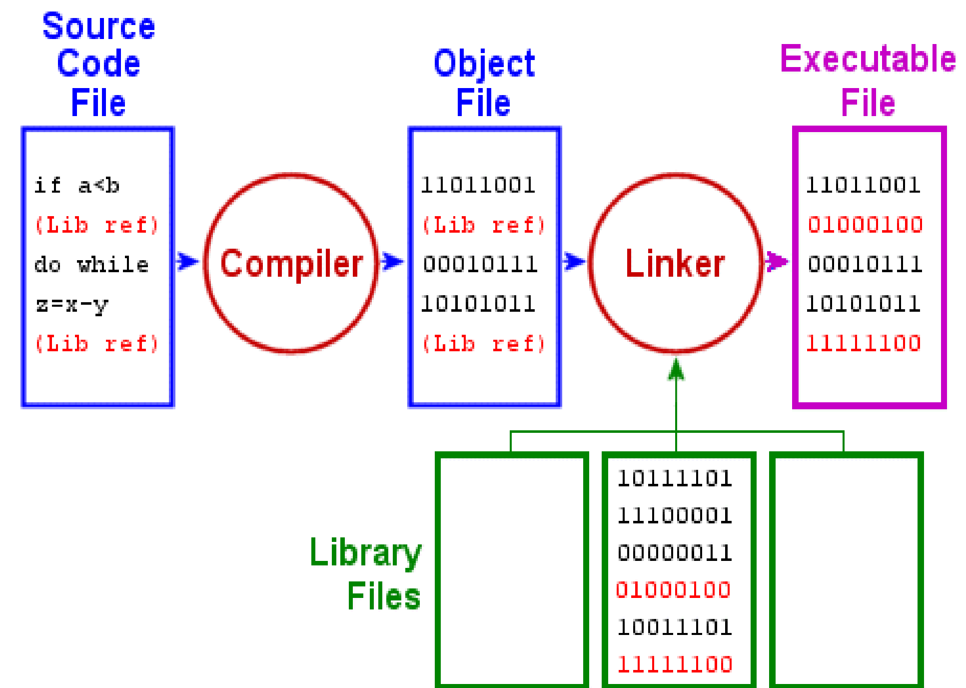
PROGRAMMING IN C++

KAUSIK DATTA
18-Oct-2017



Objectives

- Understanding constant
- Understanding inline function

Notion of const-ness

- A *const* variable must be initialized when defined
- 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;
```

Source: Programming in C++ by Prof. Partha Pratim Das

Notion of const-ness

- 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
```

Source: Programming in C++ by Prof. Partha Pratim Das

Compare #define and Constant

```
#include <stdio.h>
#include <math.h>

#define TWO 2
#define PI 4.0*atan(1.0)
int main()
{
    int r = 10;
    double peri = TWO * PI * r;
    printf("Perimeter = %f\n", peri);
    return 0;
}
```

```
#include <iostream>
#include <cmath>
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;
    cout << "Perimeter = " << peri <<
    endl;
    return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Advantages of const

- Prefer const over #define

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

Source: Programming in C++ by Prof. Partha Pratim Das

Pointer and const

- const-ness can be used with Pointers in one of the two ways:
  - **Pointer to Constant data** where the 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.
...
p = &m; // p points to m. *p is 4.
*p = 8; // m and *p are 8 now. n is 7.
```

Source: Programming in C++ by Prof. Partha Pratim Das

Pointer and const : Pointer to Constant 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
```

Interestingly,

```
*p = 8;  // Error: p points to a constant data
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
```

Source: Programming in C++ by Prof. Partha Pratim Das

Pointer and const : Constant 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
```

Source: Programming in C++ by Prof. Partha Pratim Das

Pointer and const : 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;
```

- What will be the Output?
- How to stop editing the name?
- How to stop changing the name?
- How to stop both?

Source: Programming in C++ by Prof. Partha Pratim Das

Pointer and const : Example

- 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
```

Source: Programming in C++ by Prof. Partha Pratim Das

Macros with Parameters

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main()
{
int a = 3;
int b = SQUARE(a);
cout << "Square = " << b << endl;
return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Pitfalls of Macro

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main()
{
int a = 3;
int b = SQUARE(a + 1); // Wrong macro expansion
cout << "Square = " << b << endl;
return 0;
}
```

Output is 7 instead 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);
```

Source: Programming in C++ by Prof. Partha Pratim Das

Pitfalls of Macro

- What would be the output of

```
SQUARE(++a);
```

- 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.

Source: Programming in C++ by Prof. Partha Pratim Das

inline function

- 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 called functions is avoided
- 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

Source: Programming in C++ by Prof. Partha Pratim Das

Example

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x

int main()
{
int a = 3;
int b = SQUARE(a + 1);
cout << "Square = " << b <<
endl;
return 0;
}
```

```
#include <iostream>
using namespace std;
inline int SQUARE(int x)
{ return x * x; }
int main()
{
int a = 3;
int b = SQUARE(a + 1);
cout << "Square = " << b <<
endl;
return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Macros & inline Functions: Compare and Contrast

Macro	Inline Function
Efficient in execution	Efficient in execution
Code get expanded in place of use	Code get expanded in place of use
Has syntactic and semantic pitfalls	No pitfall
Type checking for parameters is not done	Type checking for parameters is robust
Helps to write max / swap for all types	Needs *template* for the same purpose
Errors are not checked during compilation	Errors are checked during compilation
Not available to debugger	Available to debugger in DEBUG build

Source: Programming in C++ by Prof. Partha Pratim Das

REFERENCE AND POINTER

Outline

- Reference variable or Alias
  - Basic Notion
  - Call-by-reference in C++
- Example: Swapping two number in C
  - Using Call-by-value
  - Using Call-by-address
- Call-by-reference in C++ in contrast to Call-by-value in C
- Use of const in Alias / Reference
- Return-by-reference in C++ in contrast to Return-by-value in C
- Differences between References and Pointers

Source: Programming in C++ by Prof. Partha Pratim Das

Reference

- A reference is an alias / synonym for an existing variable

```
int i  = 15; // i is a variable

int &j = i; // j is a reference to i
```

```
      i    ←——   variable
      15   ←——   memory content
      200  ←——   address
      j    ←——   alias or reference
```

Source: Programming in C++ by Prof. Partha Pratim Das

Reference

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, &b = a; // b is reference of a
    // a and b have the same memory
    cout << "a  = " << a << ", b   = " << b << endl;
    cout << "&a = " << &a << ", &b = " << &b << endl;

    ++a; // Changing a appears as change in b
    cout << "a = " << a << ", b = " << b << endl;

    ++b; // Changing b also changes a
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Call by Reference

```
#include <iostream>
using namespace std;
void Function_under_param_test(int &b, int c);
int main()
{
    int a = 20;
    cout << "a = " << a << ", &a = " << &a << endl;
    Function_under_param_test(a, a);
    return 0;
}
void Function_under_param_test(int &b, int c)
{
    cout << "b = " << b << ", &b = " << &b << endl;
    cout << "c = " << c << ", &c = " << &c << endl;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Call by Reference

- Actual param a and formal param b get the same value in called function
- Actual param a and formal param c get the same value in called function
- Actual param a and formal param b get the same addresses in called function
- Actual param a and formal param c have different addresses in called function

Source: Programming in C++ by Prof. Partha Pratim Das

swap function : call by value

```
#include <stdio.h>
void swap(int, int);
int main()
{
    int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n", a, b);
    swap(a, b);
    printf("a= %d & b= %d on swap\n", a, b);
    return 0;
}
void swap(int c, int d)
{
    int t;
    t = c;
    c = d;
    d = t;
}
```

Source: *Programming in C++ by Prof. Partha Pratim Das*

swap function : call by address

```
#include <stdio.h>
void swap(int *, int *);
int main()
{
    int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n", a, b);
    swap(&a, &b);
    printf("a= %d & b= %d on swap\n", a, b);
    return 0;
}
void swap(int *c, int *d)
{
    int t;
    t = *c;
    *c = *d;
    *d = t;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

swap function : call by reference

Call by Value : Wrong

```
#include <stdio.h>
void swap(int, int);
int main()
{
    int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n", a, b);
    swap(a, b);
    printf("a= %d & b= %d on swap\n", a, b);
    return 0;
}
void swap(int c, int d)
{
    int t;
    t = c;
    c = d;
    d = t;
}
```

Call by Reference : Correct

```
#include <stdio.h>
void swap(int&, int&);
int main()
{
    int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n", a, b);
    swap(a, b);
    printf("a= %d & b= %d on swap\n", a, b);
    return 0;
}
void swap(int &c, int &d)
{
    int t;
    t = c;
    c = d;
    d = t;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Call by Reference

- A reference parameter may get changed in the called function
- Use const to stop reference parameter being changed

```
#include <iostream>
using namespace std;
int Ref_const(const int &x)
{
    ++x; // Not allowed
    return (x);
}
int main()
{
    int a = 10, b;
    b = Ref_const(a);
    cout << "a = " << a <<" and“ << " b = " << b;
    return 0;
}
```

Source: *Programming in C++ by Prof. Partha Pratim Das*

Return-by-reference

```
#include <iostream>
using namespace std;
int Function_Return_By_Val(int &x)
{
    cout <<"x = "<<x<<" &x = "<<&x<<endl;
    return (x);
}
int main()
{
    int a = 10;
    cout <<"a = "<<a<<" &a = "<<&a<<endl;
    const int& b = Function_Return_By_Val(a);
    cout << "b = " << b << " &b = " << &b <<
    endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int& Function_Return_By_Val(int &x)
{
    cout <<"x = "<<x<<" &x = "<<&x<<endl;
    return (x);
}
int main()
{
    int a = 10;
    cout <<"a = "<<a<<" &a = "<<&a<<endl;
    const int& b = Function_Return_By_Val(a);
    cout << "b = " << b << " &b = " << &b
    <<endl;
    return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Return-by-reference

```
#include <iostream>
using namespace std;
int& Return_ref(int &x)
{
    return (x);
}
int main()
{
    int a = 10;
    int b = Return_ref(a);
    cout << "a = " << a << " and b = " << b << endl;
    Return_ref(a) = 3; // Changes reference
    cout << "a = " << a;
    return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Pitfalls in reference

Wrong declaration	Reason	Correct declaration
int& i;	no variable to refer to; must be initialized	int& i = j;
int& j = 5;	no address to refer to as 5 is a constant	const int& j = 5;
int& i = j + k;	only temporary address (result of j + k) to refer to	const int& i = j + k;

Source: Programming in C++ by Prof. Partha Pratim Das

Function I/O

	Purpose	Mechanism
Value Parameter	Input	Call-by-value
Reference Parameter	In-Out	Call-by-reference
const Reference Parameter	Input	Call-by-reference
Return Value	Output	Return-by-value Return-by-reference

Source: Programming in C++ by Prof. Partha Pratim Das

Recommended Mechanisms

- Call
  - Pass parameters of built-in types by value
    - Recall: Array parameters are passed by reference in C
  - Pass parameters of user-defined types by reference
    - Make a reference parameter const if it is not used for output
- Return
  - Return built-in types by value
  - Return user-defined types by reference
    - Return value is not copied back
    - May be faster than returning a value
    - Beware: Calling function can change returned object
    - Never return a local variables by reference

Source: Programming in C++ by Prof. Partha Pratim Das

Reference Vs Pointer

- Pointer
  - Refers to an address
  - Pointers can point to NULL; int *p = NULL;
  - Pointers can point to different variables at different times

    ```
    int a, b, *p;
    p = &a; // p points to a
    p = &b // p points to b
    ```
  - NULL checking is required Makes code faster
  - Allows users to operate on the addresses
  - Array of pointers can be defined
- Reference
  - Refers to an address
  - References cannot be NULL

    ```
    int &j ; //wrong
    ```
  - For a reference, its referent is fixed
  - int a, c, &b = a;
  - &b = c // Error
  - Does not require NULL checking
  - Does not allow users to operate on the address. All operations are interpreted for the referent
  - Array of references not allowed

Source: Programming in C++ by Prof. Partha Pratim Das

DEFAULT PARAMETER AND FUNCTION OVERLOADING

Function with default parameter

```
#include <iostream>
using namespace std;
int IdentityFunction(int a = 10)
{
    return (a);
}
int main()
{
    int x = 5;
    int y = IdentityFunction(x); // Usual function call
    cout << "y = " << y << endl;
    y = IdentityFunction(); // Uses default parameter
    cout << "y = " << y << endl;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Function with 2 default parameters

```
#include<iostream>
using namespace std;
int Add(int a = 10, int b = 20)
{
    return (a + b);
}
int main()
{
    int x = 5, y = 6;
    int z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;
    z = Add(x); // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;
    z = Add(); // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Default Parameter: Highlighted Points

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function and not in the definition of the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters
- Default arguments may be expressions also
- All parameters to the right of a parameter with default argument must have default arguments
- Default arguments cannot be re-defined
- All non-defaulted parameters needed in a call

Source: Programming in C++ by Prof. Partha Pratim Das

Function Overloading

- The same function name may be used in several definitions
- Functions with the same name must have different number of formal parameters and/or different types of formal parameters
- Function selection is based on the number and the types of the actual parameters at the places of invocation
- Function selection (Overload Resolution) is performed by the compiler
- Two functions having the same signature but different return types will result in a compilation error due to attempt to re-declare
- Overloading allows Static Polymorphism

Source: Programming in C++ by Prof. Partha Pratim Das

Overload Resolution

- To resolve overloaded functions with one parameter
  - Identify the set of Candidate Functions
  - From the set of candidate functions identify the set of Viable Functions
  - Select the Best viable function through (Order is important)
    - Exact Match
    - Promotion
    - Standard type conversion
    - User defined type conversion

Source: Programming in C++ by Prof. Partha Pratim Das

Exact Match

- lvalue-to-rvalue conversion
  - Most common
- Array-to-pointer conversion

```
Definitions: int ar[10];
void f(int *a);
Call: f(ar)
```

- Function-to-pointer conversion

```
Definitions: typedef int (*fp) (int);
void f(int, fp);
int  g(int);
Call: f(5, g)
```

- Qualification conversion
- Converting pointer (only) to const pointer

Source: Programming in C++ by Prof. Partha Pratim Das

Promotion and Conversion

- Examples of Promotion
  - char to int; float to double
  - enum to int / short / unsigned int / ...
  - bool to int
- Examples of Standard Type Conversion
  - integral conversion
  - floating point conversion
  - floating point to integral conversion
  - pointer conversion
  - bool conversion

Source: Programming in C++ by Prof. Partha Pratim Das

Overload Resolution : One parameter

- In the context of a list of function prototypes:

```
int g(double);                     // F1
void f();                          // F2
void f(int);                       // F3
double h(void);                    // F4
int g(char, int);                  // F5
void f(double, double = 3.4); // F6
void h(int, double);               // F7
void f(char, char *);              // F8
```

- The call site to resolve is:

```
f(5.6);
```

- Resolution:
  - Candidate functions (by name): F2, F3, F6, F8
  - Viable functions (by # of parameters): F3, F6
  - Best viable function (by type double Exact Match): F6

Source: Programming in C++ by Prof. Partha Pratim Das

Overload Resolution Fails

- Consider the overloaded function signatures:

```
int fun(float a) {...}              // F1
int fun(float a, int b) {...}       // F2
int fun(float x, int y = 5) {...} // F3
int main()
{
    float p = 4.5, t = 10.5;
    int s = 30;
    fun(p, s); // CALL - 1
    fun(t); // CALL - 2
    return 0;
}
```

- CALL - 1: Matches F2 & F3
- CALL - 2: Matches F1 & F3
- Results in ambiguity

Source: Programming in C++ by Prof. Partha Pratim Das

Default Parameter and Function Overloading

- Function overloading can use default parameter
- However, with default parameters, the overloaded functions should still be resolvable

```
#include<iostream>
using namespace std;
int    Area(int a, int b = 10)  { return (a * b); }
double Area(double c, double d) { return (c * d); }
int main()
{
    int x = 10, y = 12;
    double z = 20.5, u = 5.0;
    int t = Area(x);       // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // t = 100
    double f = Area(z, y); // Binds double Area(double, double)
    cout << "Area = " << f << endl; // f = 102.5
    return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das

Default Parameter and Function Overloading

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);
int main()
{
    int x = 5, y = 6;
    f();      // Error C2668: 'f': ambiguous call to overloaded function
              // More than one instance of overloaded function "f"
              // matches the argument list:
              // function "f()"
              // function "f(int = 0)"
    f(x);     // int f(int);
    f(x, y); // int f(int, int);
    return 0;
}
```

Source: Programming in C++ by Prof. Partha Pratim Das