MA517M-Basic Programming Laboratory

Laboratory 3 : Selection

Panchatcharam Mariappan[1]

[1]Associate Professor
Department of Mathematics and Statistics
IIT Tirupati, Tirupati

August 18, 2025

Assignment Operator

Assignment Operator

- An Assignment statement has three components: lvalue, equal sign (=), rvalue
- It assigns the literal value of the RHS to a variable of the LHS:
- example `x = 5;`
- or It evaluates an expression of the RHS and assigns the result to a variable of the LHS. `c = a + b;`
- The expression on the RHS is evaluated first to produce an rvalue, and the result is assigned to the variable in the LHS or lvalue

Assignment Operator

- **Note:** The LHS must be a variable. It must not contain any expressions. The following are invalid assignment statements
- `a + b = c`
- `5 = x`
- **Note:** The = sign in mathematics is different than in programming.
- `x = x + 1` is illegal in mathematics, but valid in C++
- `a + b = c` is legal in mathematics, but invalid in C++
- `a + b = c + d` is legal in mathematics, but invalid in C++

Overflow/Underflow

Overflow/Underflow

- You will often encounter underflow and overflow in your research or programming in the lab.
- It is a programmer's responsibility to check the overflow and underflow

Overflow occurs while you assign a value less than the maximum permissible value.

Overflow

```
#include<iostream>
using namespace std;
int fact(int n)
{
 if(n==0)
   return 1;
 else
    return n*fact(n-1);
}
```

```
int main()
{
 int n;
 cout<<"Enter n"<<endl;
 cin>>n;
 cout<<n<<"! = "<<fact(n)<<endl;
 return 0;
}
```

Give the input as 5, and you will get proper input. If you change the input to 12, you will still get the correct answer. But when you use 20 as the input, you will get a negative number or a wrong answer from 13 onward. Why?

Overflow

- When the number exceeds `INT_MAX` which we have found from Example 2 of Lab2 shows that, it is 2147483647. However, 12!= 479001600, whereas 13!=6227020800 which exceeds `INT_MAX`. Therefore, it is an overflow.
- If you change int as long int, then you will get the correct answer for 13 to 20!. However, this also has a limitation. Check!

Underflow

Underflow occurs while you assign a value less than the minimum value. Consider the following example,

```
#include<iostream>
using namespace std;
int main()
{
  int n = -2147483640;
  int m = 50000;
  cout<<n-m<<endl;
  return 0;
}
```

`n-m` should be negative, but when you execute the program, you will get a positive number, because it is an underflow error. Since `INT_MIN` is -2147483647, we get this error.

Compound Assignment Operators

Compound Assignment Operators

Apart from the simple assignment operator, we can combine the arithmetic operators with the assignment operator as follows:

Operator	Description	Example	Result
+=	varName+=expression	varName = varName + expression	`x+=2;`
-=	varName+=expression	varName = varName + expression	`x-=2;`
*=	varName+=expression	varName = varName * expression	`x*=2;`
/=	varName+=expression	varName = varName * expression	`x/=2;`
%=	varName=expression	varName = varName % expression	`x%=2;`

Relational Operators

Relational Operators

It checks the relationship between two operands, returns 1 if the relation is true and returns 0 if the relation is false. For the example, we have `int x=2,y=3;`

Operator	Syntax	Description	Example	Result
>	Expression1>Expression2	Greater than	`x>y`	False (0)
<	Expression1<Expression2	Less than	`x<y`	True (1)
>=	Expression1>=Expression2	Greater than or Equal to	`x>=y`	False (0)
<=	Expression1<=Expression2	Less than or Equal to	`x<=y`	True (1)
==	Expression1==Expression2	Equal to	`x==y`	False (0)
!=	Expression1!=Expression2	Not Equal to	`x!=y`	True (1)

Note: `a + b = c + d` is not valid in C++, but `a + b == c + d` is valid in C++. The first one is an assignment operator, whereas the second one is a relational operator that verifies whether the relation between *lvalue* and *rvalue* is true or not.

Logical Operators

Logical Operators

It checks the relationship between two operands, returns 1 if the relation is true, and returns 0 if the relation is false. For example, we have

```
int x=2,y=3,z=4;
```

Operator	Syntax	Meaning	Example
`&&`	Logical AND	`x>y && x<z`	False (0) AND TRUE (1) = FALSE (0)
		`x<y && x<z`	TRUE (1) AND TRUE (1) = TRUE (1)
		`x<y && x>z`	TRUE (1) AND FALSE (0) = FALSE (0)
		`x>y && x>z`	FALSE (0) AND FALSE (0) = FALSE (0)
`||`	Logical OR	`x>y || x<z`	FALSE (0) OR TRUE (1) = TRUE (1)
		`x<y || x<z`	TRUE (1) OR TRUE (1) = TRUE (1)
		`x<y || x>z`	TRUE (1) AND FALSE (0) = TRUE (1)
		`x>y || x>z`	FALSE (0) OR FALSE (0) = FALSE (0)
`!`	Logical NOT	`!(x==y)`	NOT FALSE (0) = TRUE (1)
		`!(x<y)`	NOT TRUE (1) = FALSE (0)

Selection/Decision Making: if Condition

if condition

It checks the relationship between two operands, returns 1 if the relation is true, and returns 0 if the relation is false. For the example, we have `int x=2,y=3;` When there are multiple choices available to select from, we can use decision-making to select based on the requirement

- It is useful if you want to execute the code only if the given condition is true

Syntax

```
// if-then
if ( booleanExpression )
{
  true-block ;
}
```

Example

```
#include<iostream>
using namespace std;
int main()
{
    int x=8;
    if (x%2==0)
    {
        cout<<"The given number is even"<<endl;
    }
    return 0;
}
```

Example

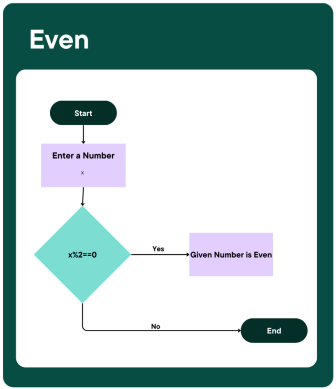


Figure 1: Flowchart

if..else condition

1. It is useful if you want to execute a code block when the condition is true and another code block if the condition is false
2. This is a fundamental control structure in programming that allows for decision-making based on conditions
3. This is also known as **Binary Decision Making**
4. Short-Circuit Evaluation: In a condition with logical operators (`&&`, `||`), C++ uses short-circuit evaluation

if..else condition

Syntax

```
// if-else
if ( booleanExpression )
{
  true-block ;
}
else
{
  false-block
}
```

Example

```
#include<iostream>
using namespace std;
int main()
{
    int x=5;
    if (x%2==0)
    {
      cout<<"The given number is even"<<endl;
    }
    else
    {
      cout<<"The given number is odd"<<endl;
    }
    return 0;
}
```

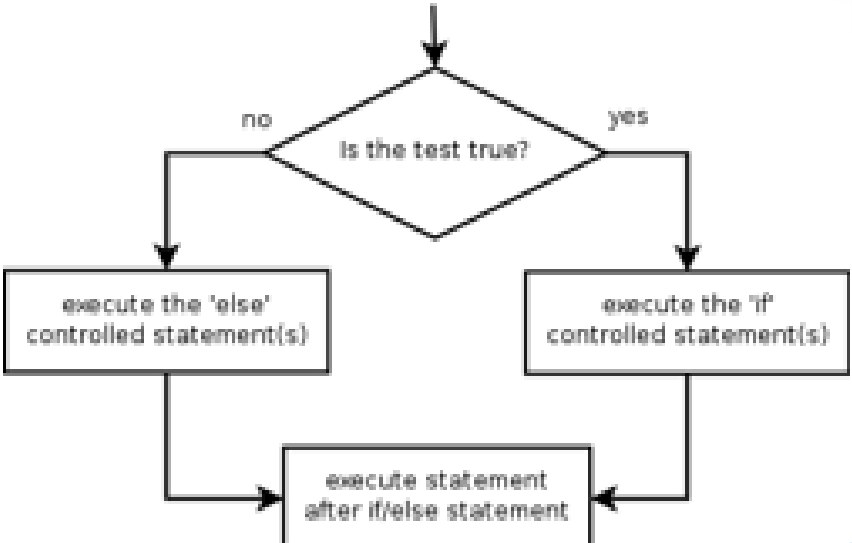
Example

For given two integers, find which one is smaller than the other

```
#include<iostream>
using namespace std;
int main()
{
  int a,b;
  cout<<"Enter two integers, let me tell you their relations"<<endl;
  cout<<"Enter the first integer: "<<endl;
  cin>>a;
  cout<<"Enter the second integer: "<<endl;
  cin>>b;

  if(a<b)
  {
    cout<<a<<" is smaller than "<<b<<endl;
  }
  else
  {
    cout<<b<<" is smaller than "<<a<<endl;
  }
  return 0;
}
```

Example


Is the test true?
no
yes
execute the 'else' controlled statement(s)
execute the 'if' controlled statement(s)
execute statement after if/else statement

Ternary Conditional Operator

Example

- The ternary conditional operator `?:` can sometimes replace a simple if-else statement for compact code.
- It is a compact form of an if-else statement

Syntax

```
booleanExpression? true-block: false-block;
```

Example

```
#include<iostream>
using namespace std;
int main()
{
    int x=5, y=8;
    int max=(x>y)?x:y; //Ternary Conditional Operator
    cout<<"max("<<x<<","<<y<<" = "<<max<<endl;
    return 0;
}
```

Benefits and Warning

- It makes your code more concise, especially for simple conditional assignments.
- **Readability**: It can improve readability when used in simple situations
- Reduces the amount of code needed
- Don’t use it for complex expressions or nested ternary operations
- If you need to perform more complex logic, use multiple `if-else` statements

Nested if

Nested if

It is a `if` statement placed inside another `if` or `else` statement. This allows you to test multiple conditions in a hierarchical manner. **Syntax**

```
if ( boolExpr-1)
{
  // Block of code executed if  boolExpr-1 is true
  if ( boolExpr-2)
  {
    // Block of code executed if  boolExpr-1 & boolExpr-2 are both true
    if ( boolExpr-3)
    {
      // Block of code executed if  boolExpr-1,  boolExpr-2, and  boolExpr-3 are all true
    }
    else
    {
      // Block of code executed if boolExpr-1 and boolExpr-2 are true, but boolExpr-3 is false
    }
  }
  else
  {
    // Block of code executed if boolExpr-1 is true, but boolExpr-2 is false
  }
}
else
{
  // Block of code executed if boolExpr-1 is false
}
```

Example

```
bool Valid = true;
int correctPIN = 1234, yourPIN;
double Bal = 50000.0, withdraw;
cout << "Welcome to the ABC Bank ATM!\n Insert Your ATM Card!\n";
if (Valid) {
  cout << "Please enter your PIN: ";
  cin >> yourPIN;
  if (yourPIN == correctPIN)
  {
    cout << "Enter amount to withdraw: ";
    cin >> withdraw;
    if (withdraw <= Bal)
    {
      Bal -= withdraw;
      cout << "Please collect your cash. Remaining balance: " << Bal << "\n";
    }
    else {
      cout << "Insufficient funds. Transaction canceled.\n";
    }
  }
  else {
    cout << "You have Entered a Wrong PIN. Transaction canceled.\n";
  }
}
else
{
  cout << "Invalid card. Please contact your bank.\n";
}
```

Example

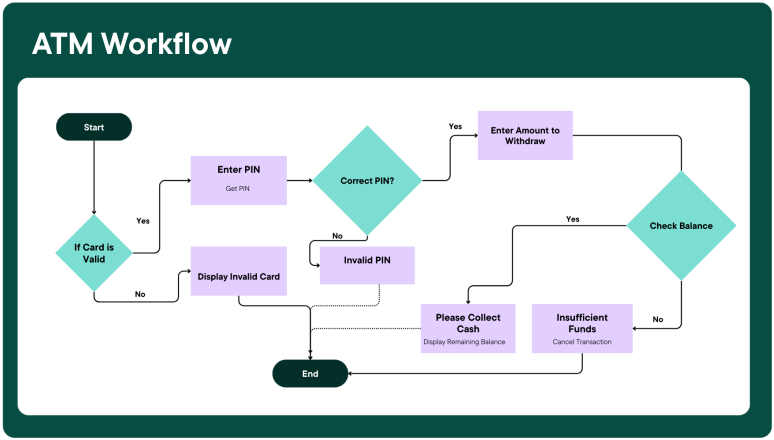


Figure 3: Flowchart

Benefits and Warning

- Hierarchical Decision Making
- Readability
- Conditional Dependence
- Deeply nested `if` statements can make the code harder to read and maintain
- If the nesting becomes too deep, consider refactoring using `else if`
- Nested if statements can become difficult to follow, especially if the logic is complex

if..else if .. else

if..else if .. else

This allows for executing different blocks of code based on multiple conditions. It is useful if you want to execute a

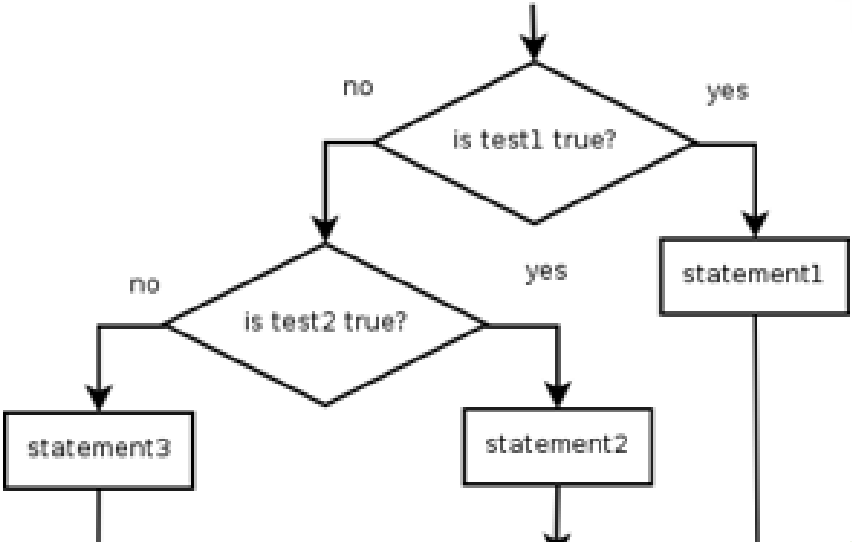
1. first code-block when first condition is true
2. second code-block if the second condition is true but the first condition is false
3. third code-block if the third condition is true but the first two conditions are false
4. kth code-block if the kth condition is true but the previous $k-1$ conditions are false
5. else block, if none of the conditions are true.

if..else if .. else

It is a `if` statement placed inside another `if` or `else` statement. This allows you to test multiple conditions in a hierarchical manner. **Syntax**

```
// if..else if .. else
if ( booleanExpr-1 ) {
    block-1 ;
}
else if ( booleanExpr-2 ) {
    block-2 ;
}
else if ( booleanExpr-3 ) {
    block-3 ;
}
.
.
.
else if ( booleanExpr-k ) {
    block-k ;
}
else {
    elseBlock ;
}
```

Example


no
yes
is test1 true?
no
yes
is test2 true?
statement1
statement3
statement2

Example

```
if (mark >= 90)
{
    cout << "S Grade" << endl;
}
else if (mark >= 80)
{
    cout << "A Grade" << endl;
}
else if (mark >= 70)
{
    cout << "B Grade" << endl;
}
else if (mark >= 60)
{
    cout << "C Grade" << endl;
}
else if (mark >= 50)
{
    cout << "D Grade" << endl;
}
else
{
    cout << "E Grade" << endl;
}
```

Example : Smallest of three integers

```
#include<iostream>
using namespace std;
int main(void)
{
   int a,b,c;
   cout<<"Enter three integers, let me find the smallest integer"<<endl;
   cout<<"Enter the first integer: "<<endl;
   cin>>a;
   cout<<"Enter the second integer: "<<endl;
   cin>>b;
   cout<<"Enter the third integer: "<<endl;
   cin>>c;

   if(a<b && a<c)
   {
      cout<<a<<" is the smallest integer"<<endl;
   }
   else if(b<a && b<c)
   {
      cout<<b<<" is the smallest integer"<<endl;
   }
   else
   {
      cout<<c<<" is the smallest integer"<<endl;
   }

   return 0;
}
```

Benefits and Warning

- Efficient Decision Making
- Readability
- Flexibility

Switch

switch

1. An alternate for long if ... else if ... else statements
2. Multiway branch statement
3. Useful to check a single variable for different conditions

switch

Syntax

```
switch (n)
{
  case constant1:
    //Code to be executed
    break;
  case constant2:
    //Code to be executed
    break;
  .
  .
  .
  case constantN:
    //Code to be executed
    break;
  default:
    //code to be executed if any of the above don't match
}
```

Example:Traffic Light Simulation

```
#include<iostream>
using namespace std;
int main()
{
  char sig;
  cout<<"Enter the Signal"<<endl;
  cin>>sig;

  switch(sig)
  {
    case 'R':
      cout<<"Stop"<<endl;
      break;

    case 'Y':
      cout<<"Ready"<<endl;
      break;

    case 'G':
      cout<<"Go"<<endl;
      break;
    default:
      cout<<"Invalid Signal"<<endl;
  }
  return 0;
}
```

Example: Simple Calculator

```
#include<iostream>
using namespace std;
int main()
{
  char op;
  double a,b,c;

  cout<<"Enter an operator (+, -, *, /):"<<
       endl;
  cin>>op;

  cout<<"Enter the first number: "<<endl;
  cin>>a;

  cout<<"Enter the second number: "<<endl;
  cin>>b;

    switch(op)
    {
        case '+':
            c = a + b;
            cout<<a<<" + "<<b<<" = "<<c<<endl;
            break;
    case '-':
      c = a - b;
      cout<<a<<" - "<<b<<" = "<<c<<endl;
      break;
    case '/':
      if (b==0)      {
        cout<<"Division by zero is not possible"<<endl;
      }
      else       {
        c=a/b;
        cout<<a<<"/"<<b<<" = "<<c<<endl;
      }
      break;
    case '*':
      c = a*b;
      cout<<a<<" * "<<b<<" = "<<c<<endl;
      break;

    // operator is doesn't match any case constant (+, -,
         *, /)
    default:
      cout<<"Invalid Operator"<<endl;
  }
  return 0;
}
```

Tips

While writing the code, always make proper indentation, so that you can differentiate between the loops, if, start, and end of each block will be readable

Tips

Dangline `else` Problem: When if-else statements are nested without braces {}, it can lead to ambiguity known as the dangling else problem, where it's unclear which if the else belongs to. Remedy: Always use braces {} to clearly define blocks.

Tips

Using operator == for assignment or using operator = for equality is a logical error.

Thanks

Doubts and Suggestions

panch.m@iittp.ac.in