|  |  |
| --- | --- |
| **Program / Project Name:** | Modern C++ |
| **Checklist / Template Completed by:** | TBD |
| **Date Completed:** | Click here to enter a date. |

**C++ and Modern C++**

C++11, C++14

**Detailed Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Editor(s)** | **Description of change** |
| A.0 | 13-AUG-2020 | Sang-Gu Kang | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[1 변수 4](#_Toc48254445)

[1.1 초기화 4](#_Toc48254446)

[1.1.1 보편적 초기화 4](#_Toc48254447)

[2 자료형 4](#_Toc48254448)

[2.1 auto 4](#_Toc48254449)

[3 API 5](#_Toc48254450)

[3.1.1 String 5](#_Toc48254451)

# 변수

## 초기화

### 보편적 초기화

C++에는 초기화 방법이 너무 많아서 많은 혼동을 가져왔다. 각 초기화 방법들은 서로 다른 형식을 사용해 왔다. 그동안 혼란의 극치였던 초기화 방법을 통일한 것이 보편적 초기화 방법이다.

|  |
| --- |
| int i {100}; //int i = 100;과 동일  string s {“hello”}; //string s = “hello”;와 동일 |

# 자료형

## auto

C++ 이전에 auto는 단순히 자동 변수라는 의미만을 가지고 있었다. 즉 지역 변수가 함수 안에서 선언되었다가 함수가 종료되면 자동으로 소멸되는 것을 의미하였다.

double d = 1.0;  
변수 d를 double로 선언했지만, 컴파일러는 이미 초기값 1.0을 보고 변수 d가 double형이라는 것을 알 수 있다. 이런 경우 자료형을 생략할 때 사용하는 키워드가 auto이다. 이것을 자동 타입 추론(automatic type deduction)이라고 부른다.  
auto d = 1.0; // 1.0은 double형리터럴이어서 d의 자료형은 double이 된다.

|  |
| --- |
| auto add (int x, int y) {  return x + y;  }  int main()  {  auto sum = add(5, 6); // add()는 정수값을 반환하므로 sum은 int형이 된다.  return 0;  } |

앞으로 초기값은 줄 수 있지만 자료형이 생각나지 않으면 일단 auto를 사용해 보자.

# 클래스

## vector

push\_back()은 편리하기는 하지만 비효율적이다. 벡터의 크기가 1만큼 증가하게 되면 새로운 메모리 영역을 찾아서 벡터를 이동하여야 하기 때문이다. 따라서 미리 크기를 어느 정도 짐작할 수 있으면 고정된 크기를 사용하는 편이 낫다.

벡터의 처음은 v.begin()이나 v.front로 알 수 있다.(v는 만들어진 벡터. 예: vector<int> v) 벡터의 끝은 v.end() 또는 v.back()로 알 수 있다.

벡터에서 요소의 위치는 반복자(iterator)를 이용하여 표시한다. begin()과 end()는 반복자로서 벡터의 시작과 끝을 식별한다.

vector는 크기를 미리 결정하지 않아도 된다는 큰 장점을 가지고 있다. 하지만 성능이 중요한 애플리케이션을 작성할 때는 문제점이 있다. vector는 생성과 소멸을 하는데 상당한 시간이 소요된다. 따라서 vector의 장점이 많지만 성능 때문에 기존의 배열을 사용하는 경우도 많다. 이 문제를 해결하기 위해 C++11에서는 std::array를 새롭게 제시하였다.

# 생성자

## 초기화 리스트

|  |  |  |
| --- | --- | --- |
|  |  | C++14 / C++11 |
| Time(int h, int m) //생성자  {  hour = h;  minute = m;  } | -> | Time(int h, int m) : hour(h), minute(m)  {  }  Time(int h, int m) : hour{h}, minute{m}  {  }  초기화 리스트와 디폴트 인수를 동시 사용  Time(int h=0, int m=0) : hour{h}, minute{m}  {  } |

# 포인터

## nullptr

포인터를 선언했다면 반드시 초기화해 주어야 한다. 포인터가 아무것도 가리키고 있지 않을 때는 nullptr로 초기화한다.

이전에 NULL을 사용한 경우가 있으나 NULL은 정수 0이어서 문제가 발생할 수 있으니 사용하지 않아야 한다.

int \*p = nullptr;

# 접근자(getter) 와 설정자(setter)

-. 접근자는 앞에 get~~을 붙이고, 설정자는 앞에 set~~을 붙이는게 불문율이다.  
예) getNumber(), setNumber(int n)

-. 설정자는 매개 변수를 통하여 private 멤버를 설정 하는데 사용한다. 설정자를 사용하는 주된 이유 중 하나는 **매개변수를 통하여 잘못된 값이 넘어오는 경우 이를 사전에 차단**하는데 있다. 프로그램 중간 중간에 조건문을 달아서 지저분하게 구현하지 말고 설정자에서 걸러주면 **인터페이스와 구현이 분리**된다.

|  |
| --- |
| void setNumber(int n) {  if(n<0){  number = 0;  cout << “계좌번호 오류!”;  } else {  number = n;  }  } |

-. 접근자는 기본적으로 private 멤버 변수를 반환하는 구조로 이루어져 있으나, 필요할때마다 계산하여 반환하도록 할 수 있다. 프로그램 중간 중간에 멤버 변수를 변환하게 되면 복잡해지게 된다. 이 역시 인터페이스와 구현을 분리한 예가 된다.

|  |
| --- |
| void getNumber() {  return rand();  } |

# loop

## 범위 기반 루프(Todo)

# API

### String

|  |
| --- |
| to\_string() |
| 숫자를 문자열로 변환 |
| string s1 {apple};  string s2;  s2 = s1 + “ “ + to\_string(10) + “개”;  cout << s2 << endl; |